

# Programujeme algoritmy z teórie grafov

Viera Palmárová



Nitra 2004

**Názov:** Programujeme algoritmy z teórie grafov  
**Autor:** Mgr. Viera Palmárová, KI FPV UKF v Nitre

Copyright © VP 2004

# Obsah

<b>Úvod .....</b>	<b>7</b>
<b>1 Reprezentácia grafu v programe.....</b>	<b>8</b>
1.1 Matice verzus zoznamy .....	8
1.2 Niektoré ďalšie reprezentácie grafu .....	13
1.3 Cvičenia .....	15
<b>2 Prehľadávanie grafu .....</b>	<b>17</b>
2.1 Hlavná myšlienka prehľadávania = označovanie vrcholov .....	17
2.2 Prehľadávanie grafu do šírky .....	19
2.3 Prehľadávanie grafu do hĺbky .....	22
2.4 Komponenty súvislosti .....	24
2.5 Cvičenia .....	25
<b>3 Topologické triedenie .....</b>	<b>27</b>
3.1 Topologické usporiadanie vrcholov acyklického grafu.....	27
3.2 Cvičenia .....	33
<b>4 Najkratšia cesta .....</b>	<b>34</b>
4.1 Dijkstrov algoritmus .....	34
4.2 Algoritmus Bellman-Ford .....	38
4.3 Algoritmus Floyd-Warshall .....	39
4.4 Cvičenia .....	41
<b>5 Minimálna kostra .....</b>	<b>43</b>
5.1 Kruskalov algoritmus .....	44
5.2 Primov algoritmus .....	47
5.3 Cvičenia .....	49
<b>6 Eulerovský ťah.....</b>	<b>51</b>
6.1 Hľadáme v grafe uzavretý eulerovský ťah .....	51
6.2 Cvičenia .....	56
<b>Literatúra .....</b>	<b>58</b>
<b>Prílohy .....</b>	<b>59</b>

# Úvod

Grafové štruktúry a grafové algoritmy predstavujú v programovaní efektívne nástroje na riešenie problémov. S použitím pojmov a metód teórie grafov možno často elegantne vyriešiť aj taký problém, ktorý sa zdá spočiatku „príliš komplikovaný“ (napr. výber optimálnej cesty medzi určenými miestami, vybudovanie najlacnejšej komunikačnej siete, zostavenie časového rozvrhu a pod.). Problémovú situáciu stačí často modelovať pomocou grafu a aplikovať pri riešení niektorý zo známych grafových algoritmov. A problém je vyriešený! Ale ako toto riešenie implementovať v programovacom jazyku? Ak si kladiete túto otázku, a máte aspoň základné vedomosti z teórie grafov a chuť učiť sa, tento učebný text je určený práve vám.

V jednotlivých kapitolách sa postupne dozviete, ako implementovať abstraktnú údajovú štruktúru graf v programe, ako graf prehľadávať a ako naprogramovať niektoré ďalšie známe grafové algoritmy (topologické triedenie, algoritmy riešiace problém najkratšej cesty, minimálnej kostry a Eulerovského ťahu) . Za každou kapitolou vás čakajú cvičenia, na ktorých si môžete overiť, či ste preberanej látke naozaj porozumeli a ste schopní tvorivo využiť svoje vedomosti pri riešení konkrétneho problému. Zdrojové texty programových ukážok sú v Pascale, všetky používané unity sú uvedené v prílohe.

Svoje pripomienky, námety, komentáre resp. akékoľvek otázky týkajúce sa predloženého materiálu posielajte na adresu [murphy@unitra.sk](mailto:murphy@unitra.sk). Ďakujem.

Príjemné štúdium a veľa programátorských úspechov!

# 1 Reprezentácia grafu v programe

Pri programovaní grafových algoritmov musíme najprv vhodne vybrať údajovú štruktúru, ktorou budeme graf reprezentovať v pamäti počítača.

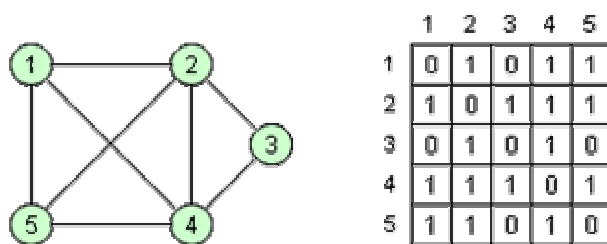
V tejto kapitole si ukážeme niektoré typické a v praxi bežne používané vnútorné reprezentácie grafu. Uvedieme možnosti, výhody i nevýhody jednotlivých reprezentácií. V celej kapitole pre jednoduchosť uvažujeme graf  $G = (V, E)$ , ktorý má  $N$  vrcholov a  $M$  hrán. Vrcholy označíme prirodzenými číslami od 1 po  $N$ , hrany číslami od 1 po  $M$ . Vždy pôjde o obyčajný graf t.j. bez slučiek a násobných hrán. Ak hovoríme o ohodnotenom grafe, máme na mysli hranovo ohodnotený graf.

Možností ako graf reprezentovať v programe je viac. Existujú však dve základné údajové štruktúry, ktoré sa v praxi používajú najčastejšie – matica susednosti a zoznamy susedov<sup>1</sup>.

## 1.1 Matice verzus zoznamy

### Matica susednosti

Neohodnotený graf (orientovaný i neorientovaný) môžeme jednoducho reprezentovať *maticou susednosti*<sup>2</sup>. Je to matica celých čísel  $A$  typu  $N \times N$ , v ktorej má prvok  $A[i, j]$  hodnotu 1, ak v grafe existuje hrana spájajúca vrchol  $i$  s vrcholom  $j$ . Ak taká hrana v grafe neexistuje, prvok  $A[i, j]$  má hodnotu 0. Namiesto čísel 1 a 0 môžeme použiť aj logické hodnoty *true* a *false* t. j. maticu s prvkami typu *boolean*.

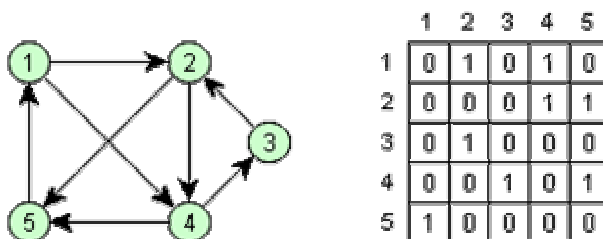


Obr. 1.1 Neorientovaný graf a jeho matica susednosti implementovaná ako dvojrozmerné pole

<sup>1</sup>V literatúre sa môžete stretnúť aj s iným pomenovaním napr. *matica príľahlosti* a *zoznamy okolí*.

<sup>2</sup>V programovaní zodpovedá matematickému objektu *matica* štandardná údajová štruktúra *dvojrozmerné pole*. V celej práci preto používame pojmy *matica* a *dvojrozmerné pole* ako synonymá.

Matica susednosti neorientovaného grafu je symetrická podľa hlavnej diagonály (pre všetky dvojice vrcholov  $i, j$  platí, že ak  $A[i, j] = 1$ , tak aj  $A[j, i] = 1$ ). V orientovanom prípade matica susednosti nemusí byť symetrická. Ak z vrcholu  $i$  vychádza orientovaná hrana a vchádza do vrcholu  $j$ , opačne orientovaná hrana nemusí v grafe vôbec existovať. Keďže v grafe nie sú slučky, všetky prvky na hlavnej diagonále majú hodnotu 0.



Obr. 1.2 Orientovaný graf a jeho matica susednosti implementovaná ako dvojrozmerné pole

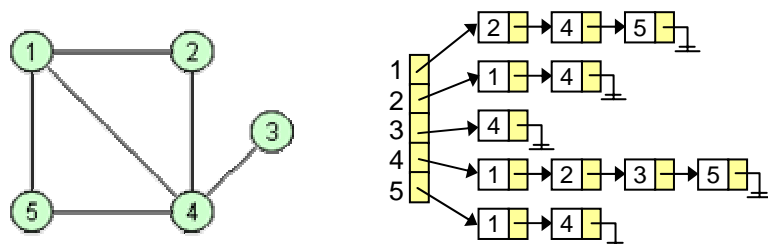
Obdobou matice susednosti pre ohodnotené grafy je *matica vzdialeností*. Prvky matice v tomto prípade obsahujú ohodnotenia jednotlivých hrán (napr. kladné celé čísla) resp. príznak, že príslušná hrana v grafe neexistuje (hodnotu, ktorá zaručene nemôže byť ohodnotením žiadnej hrany napr. 0 alebo nejaká iná dostatočne malá či dostatočne veľká konštanta).

Vďaka priamemu prístupu k prvkom matice prostredníctvom riadkových a stĺpcových indexov možno prítomnosť hrany v grafe overiť v konštantnom čase. Rovnako jednoduché je aj pridávanie a rušenie hrán. Pri tzv. *riedkych grafoch* s veľkým počtom vrcholov a relatívne malým počtom hrán môže ale maticová reprezentácia znamenať zbytočné plytvanie pamäte. Matica susednosti je pri týchto grafoch veľmi rozsiahla a obsahuje takmer samé nulové prvky. Riedke grafy možno oveľa úspornejšie reprezentovať pomocou zoznamov susedov.

### Zoznamy susedov

V tomto prípade si pre každý s  $N$  vrcholov pamätáme v jednosmernom lineárnom zozname všetky jeho susedné vrcholy (teda všetky hrany, ktoré s ním incidujú). V jednorozmernom poli s  $N$  prvkami uložíme pre každý vrchol ukazovateľ na „jeho“ zoznam susedov.

Ak chceme zistiť, či sa v grafe nachádza hrana  $(i, j)$ , prehl'adáme  $i$ -ty zoznam. Ak v ňom nájdeme vrchol  $j$ , hrana existuje. Táto operácia sa realizuje zrejme v čase  $O(d_i)$ , kde  $d_i$  je stupeň  $i$ -teho vrcholu.

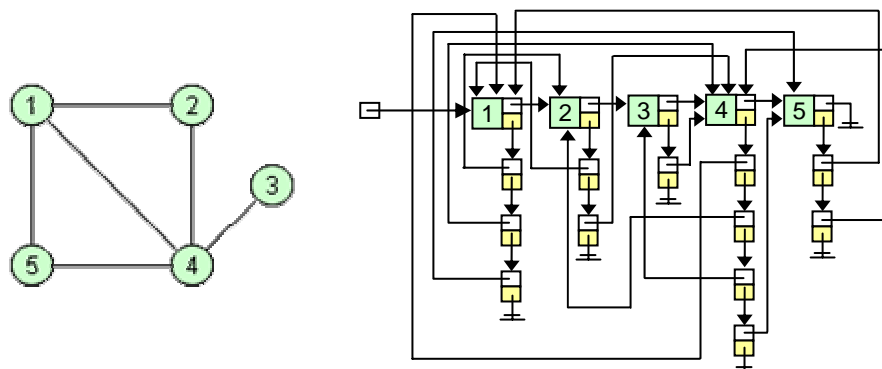


Obr. 1.3 Neorientovaný graf a jeho zoznamová reprezentácia implementovaná ako pole jednosmerných lineárnych zoznamov susedov

Takýmto spôsobom možno reprezentovať orientované aj neorientované grafy, ktoré môžu byť aj ohodnotené. V neorientovanom prípade sa každá hrana  $(i, j)$  eviduje dvakrát (v  $i$ -tom aj v  $j$ -tom zozname susedov).

Implementáciu tejto údajovej štruktúry možno samozrejme ešte viac prispôbiť riešenému problému. Ak zoznamy susedov v programe len prehl'adávame, postačia nám jednosmerné lineárne zoznamy. Ak však chceme hrany do grafu pridávať príp. ich aj rušiť, vhodnejšie budú obojsmerné lineárne zoznamy. Novú hranu by sme mohli pridať rovno na začiatok zoznamu. No pri rušení hrany je kvôli správne mu zret'azeniu užitočné poznať aj ukazovateľ na jej bezprostredného predchodcu v zozname. V neorientovanom grafe môžeme dokonca uchovávať aj ukazovateľ na druhú kópiu hrany – aj tú treba predsa z grafu odstrániť.

Stretnúť sa môžeme aj s lineárnym zoznamom všetkých vrcholov grafu, ktorý má v každom „vrchole“ uložený ukazovateľ na zoznam hrán vychádzajúcich z tohto vrcholu. Každá „hrana“ pritom obsahuje ukazovateľ na ten vrchol v hlavnom zozname vrcholov, s ktorým je táto hrana v grafe incidentná.



Obr. 1.4 Neorientovaný graf a jeho zoznamová reprezentácia implementovaná ako jednosmerný zoznam jednosmerných zoznamov hrán

Ak sa predsa len chceme vyhnúť manipulácii s ukazovateľmi a dynamickej alokácii pamäte (to by nás ale odradiť nemalo :-)) a vieme, že ide o riedky graf s maximálnym stupňom vrcholu  $D$ , mohli by sme použiť maticu typu  $N \times D$ . O každom vrchole si zapamätáme aj jeho stupeň a jeho susedov ukladáme v príslušnom riadku postupne od začiatku. Prehľadávanie „zoznamu“ susedov zvoleného vrcholu sa potom realizuje jednoduchým zvyšovaním indexu t. j. bez používania ukazovateľov.

### Čo je teda lepšie? Matice alebo zoznamy?

Správne zvolená údajová štruktúra môže riešenie problému významne zjednodušiť, sprehľadniť a zefektívniť. Pri rozhodovaní sa by sme si mali odpovedať na viaceré otázky:

**1.** S akým grafom chceme v programe manipulovať? Bude „veľký“ či „malý“? Hustý či riedky? Koľko bude mať vrcholov a koľko hrán – v priemernom a v najhoršom prípade?

Pre graf s 10 vrcholmi má matica susednosti len 100 prvkov, pre graf so 100 vrcholmi má ale matica susednosti až 10 000 prvkov a ak má graf 1 000 vrcholov je to až 1 000 000 prvkov atď. Pri riedkych grafoch (a práve s takými sa v reálnom svete stretávame oveľa častejšie) ide o zbytočné plytvanie. Matica susednosti je vhodná skôr pre „malé“ grafy alebo pre grafy s veľkým počtom hrán tzv. *husté grafy*. Pri hustých grafoch totiž použitím zoznamovej reprezentácie veľa neušetríme, veľkú časť pamäte obsadia aj samotné ukazovatele.

**2.** Aký algoritmus chceme implementovať? Ktoré operácie sa budú na grafe realizovať najčastejšie?

Pre niektoré problémy sú vhodnejšie matice (napr. hľadanie najkratších ciest medzi všetkými dvojicami vrcholov), pre iné problémy zase zoznamy (napr. algoritmy založené na prehľadávaní do hĺbky). Ak potrebujeme často zisťovať, či existuje nejaká hrana, víťazia opäť matice. Ak však chceme zistiť stupeň vrcholu, bude vhodnejšie používať zoznamy. Pre prehľadávanie grafu sú vo všeobecnosti tiež výhodnejšie zoznamové štruktúry.

**3.** Bude sa graf v priebehu vykonávania programu meniť? Ak áno, ako?

Pri opakovanom pridávaní no najmä rušení hrán sú oveľa praktickejšie zoznamy susedov. O grafe je okrem toho často potrebné uchovávať v priebehu výpočtu rôzne informácie (o vrcholoch aj o hranách). Väčšinou sa tieto údaje ukladajú ako ďalšie položky v záznamoch predstavujúcich jednotlivé vrcholy a hrany v príslušných lineárnych zoznamoch.



**Záver:** Každá z uvedených reprezentácií grafu má zrejme svoje „plus aj mínus“. No pre väčšinu „skutočných“ problémov sa v praxi ako výhodnejšie javia zoznamy.

V programoch v tomto učebnom texte budeme graf reprezentovať vo väčšine prípadov pomocou zoznamov susedov<sup>3</sup>. Na ich uloženie ale použijeme maticu. Môže sa zdať, že takto dobrovoľne kombinujeme nevýhodu lineárnych zoznamov (nutnosť vyhľadávať hrany) s nevýhodou matice (potrebuje priveľa miesta). Táto implementácia je však pre výklad dostatočne zrozumiteľná a jednoducho sa programuje. Navyše zväčša vieme odhadnúť maximálny stupeň vrcholu v danom grafe, ktorý je pri riedkych grafoch podstatne menší ako maximálny počet vrcholov. Definujeme potrebné údajové typy a deklaruje premennú typu (neohodnotený) graf:

```
const MAXV    = 100;      {maximálny počet vrcholov}
      MAXDEG  = 50;      {maximálny stupeň vrcholu - outdegree}

type graf = record
  adj: array [1..MAXV, 1..MAXDEG] of integer;  {zoznamy susedov}
  deg: array [1..MAXV] of integer;             {stupne vrcholov}
  n: integer;                                  {počet vrcholov}
  m: integer;                                  {počet hrán}
end;

var G: graf;                                  {premenná typu graf}
```

`G.adj[x]` je  $x$ -tý riadok matice `adj`, teda pole, v ktorom sú od indexu 1 počnúc uložené čísla vrcholov, do ktorých vedie hrana z vrcholu  $x$ . Hodnota `G.deg[x]` t. j. stupeň  $x$ -tého vrcholu určuje, koľko prvkov z poľa `G.adj[x]` je skutočne obsadených existujúcimi hranami. Prvky `G.adj[x,1]`, `G.adj[x,2]` až `G.adj[x,G.deg[x]]` teda predstavujú hrany vychádzajúce z vrcholu  $x$ .

Procedúra, pomocou ktorej túto štruktúru inicializujeme, môže vyzeráť napr. takto:

```
procedure inicializuj_graf (var G: graf);
var i: integer;
begin
  G.n:= 0;
  G.m:= 0;
  for i:= 1 to MAXV do G.deg[i]:= 0;
end;
```

Napišme ešte procedúru, ktorá zabezpečí načítanie grafu zo vstupného textového súboru. V súbore bude vždy v prvom riadku uvedený počet vrcholov grafu. V nasledujúcich riadkoch budú vymenované hrany – v každom riadku jedna dvojica

<sup>3</sup> Pri riešení niektorých problémov prispôsobíme reprezentáciu grafu algoritmu. Na použitie inej reprezentácie grafu čitateľa na príslušnom mieste upozorníme.

spojených vrcholov. Budeme predpokladať, že súbor popisujúci graf je vždy *korektný* t. j. že obsahuje údaje o grafe v dohodnutom formáte. Z programov tak budeme môcť vynechať ošetrovanie prípadných chýb na vstupe a zápis programu bude oveľa prehľadnejší.

```
procedure cita_j_graf (var G: graf; subor: string; orient: boolean);
var i,
    x, y: integer;      {vrcholy hrany (x, y) }
    t: text;             {vstupný textový súbor }
begin
    inicializuj_graf(G);
    assign(t, subor);
    reset(t);
    readln(t, G.n);
    while not eof(t) do begin
        readln(t, x, y);
        pridaj_hranu(G, x, y, orient);
    end;
    close(t);
end;
```

Všimnite si posledný parameter procedúry `pridaj_hranu`. Pomocou neho môžeme rozhodnúť, či treba vkladať jednu alebo až dve kópie tejto hrany, teda či ide o graf neorientovaný (*false*) alebo orientovaný (*true*). Problém sa dá elegantne vyriešiť rekurzívne:

```
procedure pridaj_hranu (var G: graf; x, y: integer; orient: boolean);
var i: integer;
begin
    inc(G.deg[x]);
    G.adj[x, G.deg[x]] := y;
    if (orient = false) then pridaj_hranu(G, y, x, true)
                           else inc(G.m);
end;
```

## 1.2 Niektoré ďalšie reprezentácie grafu

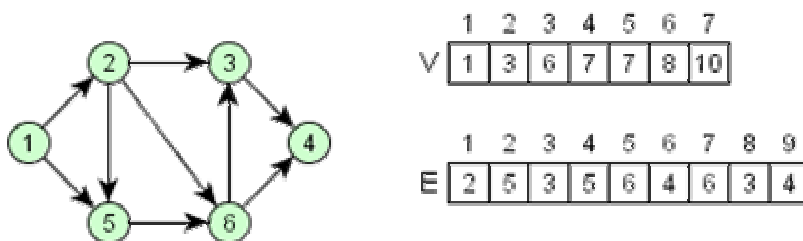
### Zoznam nasledovníkov

Zaujímavý spôsob ako uložiť jednoducho a úsporne graf predstavuje tzv. *zoznam nasledovníkov* jednotlivých vrcholov. Ku každému vrcholu budeme mať v programe k dispozícii zoznam tých vrcholov, do ktorých z neho vedie hrana. Použijeme dve jednorozmerné polia. V  $M$ -prvkovom poli  $E$  budú uložené čísla všetkých nasledovníkov – najprv vrcholy, do ktorých vchádza hrana z vrcholu 1, potom vrcholy, do ktorých vchádza hrana z vrcholu 2 atď.

Druhé pole  $V$  veľkosti  $N+1$  s prvkami typu  $1..M+1$  obsahuje indexy určujúce, od ktorého prvku v poli  $E$  začínajú nasledovníci príslušného vrcholu. Nasledovníci vrcholu s číslom  $i$  sú vrcholy s číslami  $E[V[i]]$ ,  $E[V[i]+1]$ , ...,  $E[V[i+1]-1]$ . Ak vrchol  $i$  nemá v grafe nasledovníka, položíme  $V[i] = V[i+1]$ .

Hodnoty  $N+1$  a  $M+1$  sa v deklarácii poľa  $V$  používajú čisto z praktických dôvodov – aby sme mohli nasledovníkov všetkých vrcholov vrátane posledného vrcholu s číslom  $N$  spracovávať jednotným spôsobom.

Túto reprezentáciu možno použiť pre orientované aj neorientované grafy (pole  $E$  musí mať v neorientovanom prípade samozrejme aspoň  $2 \cdot M$  prvkov). V prípade ohodnotených grafov bude pole  $E$  poľom záznamov s dvoma položkami: číslo nasledovníka a ohodnotenie príslušnej hrany.



Obr. 1.5 Orientovaný graf a jeho zoznam nasledovníkov implementovaný pomocou dvoch jednorozmerných polí

### Pole množín susedov

Pre každý vrchol  $i$  z množiny  $V$  existuje množina  $S_i$ , podmnožina množiny  $V$  taká, že  $S_i$  je množina vrcholov susediacich s vrcholom  $i$ . A túto množinu uložíme v prvku poľa s indexom  $i$ . Údajový typ *set* (množina) dovoľuje používať množinové operácie spôsobom známym z matematiky, manipulácia s grafom je preto veľmi jednoduchá. Nevýhodou ale je, že takto možno reprezentovať len grafy s najviac 256 vrcholmi a nie je možné uchovávať žiadnu informáciu o hranách.

### Matica incidencie

Pre riedke grafy sa niekedy používa aj *matica incidencie*. Ide o maticu typu  $M \times N$ . Každému riadku zodpovedá jedna hrana, každému stĺpcu jeden vrchol. V každom riadku sú hodnotou 1 priamo označené vrcholy, ktoré táto hrana spája. Ostatné prvky v tomto riadku majú hodnotu 0. Pre neorientované grafy môžeme použiť aj maticu s prvkami typu *boolean*. Pre orientované grafy potrebujeme ale rozlišovať tri hodnoty, preto použijeme maticu celých čísel. Hodnotou  $-1$  označíme vrchol, odkiaľ hrana vychádza, hodnotou  $+1$

označíme vrchol, do ktorého vstupuje. Ak ide o graf ohodnotený kladnými celými číslami, môžeme namiesto hodnoty  $+1$  uložiť príslušné ohodnotenie.

### Zoznam hrán

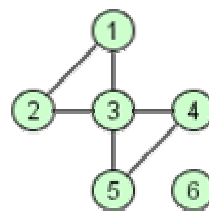
Zoznam hrán je veľmi jednoduchou aj úspornou reprezentáciou pre všetky typy grafov. Ľahko sa vytvorí (použije sa pole alebo lineárny zoznam záznamov, v ktorých si pre jednotlivé hrany pamätáme dva koncové vrcholy a prípadné ohodnotenia hrany). Niekedy však môže byť aj veľmi nepraktická. Ak napr. v programe potrebujeme vyhľadávať hrany, ktoré sú incidentné s nejakým vrcholom, v iných reprezentáciách takúto operáciu uskutočníme určite oveľa jednoduchšie a rýchlejšie.

## 1.3 Cvičenia

1. Napište samostatnú programovú jednotku s názvom *GrafUnit* pre prácu s neohodnoteným grafom. Okrem inicializácie, načítania grafu a pridania hrany (to už máme :-), pripravte tiež podprogramy na zrušenie hrany, overenie existencie hrany a výpis grafu.

2. Zapište („ručne“) uvedený graf v rôznych reprezentáciách:

- a) ako maticu susednosti
- b) ako maticu incidencie
- c) ako zoznam nasledovníkov
- d) ako pole jednosmerných zoznamov susedov



Ako sa jednotlivé reprezentácie zmenia, keď každú z hrán v grafe na obrázku nejako zorientujeme?

3. Definujte údajový typ pre neohodnotený graf ako

- a) maticu susednosti
- b) pole množín susedov
- c) zoznam nasledovníkov
- d) pole jednosmerných zoznamov susedov

Napište tiež procedúry resp. funkcie na inicializáciu grafu, prečítanie grafu zo vstupného súboru, overenie existencie hrany, zistenie stupňa vrcholu, pridanie a odstránenie hrany.

4. Upravte definície z cvičenia 3 tak, aby bola príslušná reprezentácia vhodná pre ohodnotený grafy.

5. Niekedy je v programe potrebné uchovávať informácie nielen o hranách ale aj o vrchoch grafu. Navrhnite údajovú štruktúru vhodnú na reprezentáciu grafu, v ktorom je každá hrana ohodnotená celým číslom (napr. dĺžka priamej cesty medzi príslušnými mestami) a každý vrchol jedným reťazcom (napr. názov mesta) a dvojicou celých čísel (napr. súradnice mesta na mape).
6. Napíšte program, ktorý zistí, či v danej cestnej sieti reprezentovanej maticou susednosti existujú medzi niektorými mestami jednosmerné cesty. Ak áno, vypíšte koľko a z ktorých miest do ktorých vedú.
7. Náhodne vygenerujte neorientovaný graf s  $N$  vrcholmi a  $M$  hranami. Vygenerovaný graf potom „zorientujte“ tak, že každej hrane náhodne zvolíte orientáciu. Graf reprezentujte poľom množín susedov (nezabudnite na to, že v pôvodnom – neorientovanom prípade sa každá hrana evidovala dvakrát).

## 2 Prehľadávanie grafu

Základným problémom pri riešení grafových úloh je *prehľadávanie grafu*. Stretneme sa s ním aj v mnohých iných grafových algoritmoch. Pri prehľadávaní postupne navštevujeme všetky vrcholy a hrany grafu. Toto prehľadávanie ale musí byť *systematické* (nechceme predsa nič vynechať) a *efektívne* (nechceme po grafe „blúdiť“ t. j. vracat' sa zbytočne tam, kde sme už predtým boli).

Po grafe by sa dalo „prechádzať“ viacerými spôsobmi. Existujú však dve hlavné a v praxi používané metódy:

- **prehľadávanie do šírky** (z angl. *Breadth First Search, BFS*)
- **prehľadávanie do hĺbky** (z angl. *Depth First Search, DFS*)

S oboma metódami ste sa už pravdepodobne stretli pri prehľadávaní binárnych stromov. Metóde DFS na stromoch zodpovedá metóda „preorder“; použiť metódu BFS na strome znamená postupovať pri prehľadávaní po úrovniach stromu. Strom je ale tiež len graf. Metódy prehľadávania používané pri stromoch teraz zovšeobecníme pre grafy.

### 2.1 Hlavná myšlienka prehľadávania = označovanie vrcholov

Podstatou prehľadávania grafu je postupné označovanie (farbenie) vrcholov. Ak navštívime nejaký vrchol prvýkrát, musíme si pre budúcnosť zapamätať, že o ňom už vieme. Označíme ho preto ako *objavený*. Keď vrchol objavíme, neznamena to ešte, že sme s ním skončili. Musíme totiž ešte preskúmať celé jeho okolie, teda všetky hrany, ktoré z tohto vrcholu vychádzajú. Takto objavíme všetky jeho doposiaľ neobjavené susedné vrcholy.

Každý vrchol sa tak pri prehľadávaní nachádza postupne v troch stavoch. Je najskôr *neobjavený* (biely) – na začiatku sú všetky vrcholy s výnimkou toho, z ktorého graf začíname prehľadávať, neobjavené. Potom *objavený* (žltý) – vrchol sme síce objavili, ale zatiaľ ešte nepreskúmali. A nakoniec *preskúmaný* (zelený) – vrchol sme úplne preskúmali.

Vrchol považujeme za *preskúmaný* až vtedy, keď sme preskúmali celé jeho okolie. Preto postupne preberieme všetky hrany, ktoré z neho vychádzajú. Ak niektorá vedie do zatiaľ neobjaveného vrcholu, označíme tento vrchol ako objavený a zapamätáme si, že ho v budúcnosti bude treba ešte preskúmať. Inak povedané: všetky práve objavené susedné

vrcholy skúmaného vrcholu dočasne odložíme do zoznamu „čo treba ešte preskúmať“. Neskôr ich z tohto zoznamu vyberieme a tiež preskúmame.

Algoritmy BFS a DFS sa líšia v poradí, v akom sa vrcholy grafu preskúmajú. Závisí to od použitej údajovej štruktúry pre zoznam „čo treba ešte preskúmať“. Algoritmus BFS používa *rad* (princíp FIFO), algoritmus DFS používa *zásobník* (princíp LIFO).

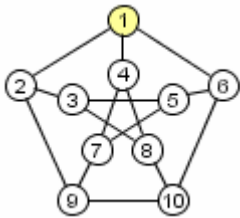
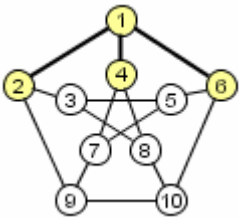
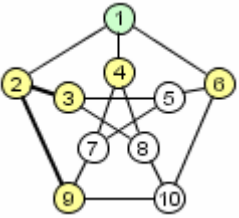
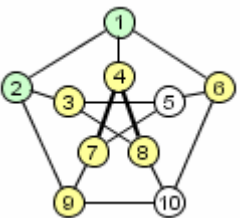
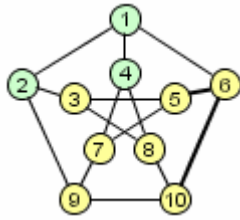
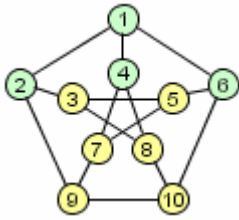
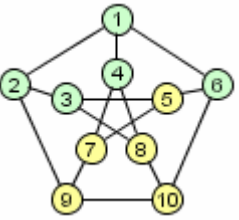
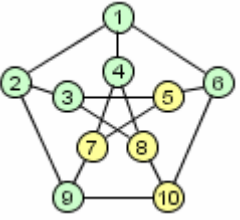
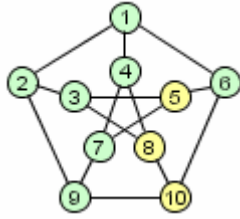
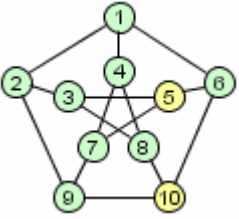
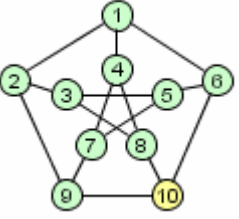
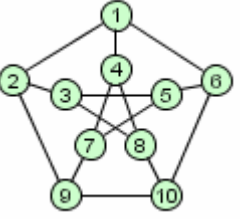
Pri algoritme BFS bude spomedzi objavených no zatiaľ nepreskúmaných susedných vrcholov ako prvý preskúmaný najskôr objavený vrchol (t. j. ten, ktorý „stojí prvý v rade“). Pri algoritme DFS je to naopak – prvý bude preskúmaný ten susedný vrchol, ktorý bol objavený ako posledný (t. j. ten, ktorý je na vrchu zásobníka). Prehľadávanie grafu skončí, keď sú všetky dostupné vrcholy grafu preskúmané.

Oba algoritmy prehľadávania sú správne – sú konečné (graf má konečný počet vrcholov a hrán) a naozaj takto navštívime všetky vrcholy grafu. Nie je totiž možné, aby po skončení prehľadávania existoval v grafe dostupný ale neobjavený vrchol  $x$ . Ak by taký vrchol existoval, musel by byť spojený hranou s nejakým objaveným vrcholom. Avšak pri preskúmaní tohto vrcholu objavíme všetky jeho susedné vrcholy – teda aj vrchol  $x$ .

Oba algoritmy prehľadávania možno použiť na orientované i neorientované grafy, oba pracujú s časovou výpočtovou zložitou  $O(N+M)$ . Musíme preskúmať každý vrchol grafu a každá hrana sa pri prehľadávaní overuje dvakrát (pre každý smer raz).

## 2.2 Prehľadávanie grafu do šírky

Prehľadávanie do šírky budeme demonštrovať na známom *Petersenovom grafe*:

			
Začíname vo vrchole 1. Označíme ho ako objavený.	Objavíme jeho tri susedné vrcholy a vložíme ich do radu.	Vrchol 1 je preskúmaný! Vyberieme z radu vrchol 2 a objavíme vrcholy 3 a 9. Vložíme ich do radu.	Vrchol 2 je preskúmaný! Vyberieme z radu vrchol 4 a objavíme vrcholy 7 a 8. Vložíme ich do radu.
Rad: prázdny	Rad: <b>2 4 6</b>	Rad: 4 6 <b>3 9</b>	Rad: 6 3 9 <b>7 8</b>
			
Vrchol 4 je preskúmaný! Vyberieme z radu vrchol 6 a objavíme vrcholy 5 a 10. Vložíme ich do radu.	Vrchol 6 je preskúmaný! Vyberieme z radu vrchol 3. Nemá neobjavených susedov.	Vrchol 3 je preskúmaný! Vyberieme z radu vrchol 9. Nemá neobjavených susedov.	Vrchol 9 je preskúmaný! Vyberieme z radu vrchol 7. Nemá neobjavených susedov.
Rad: 3 9 7 8 <b>5 10</b>	Rad: 9 7 8 5 10	Rad: 7 8 5 10	Rad: 8 5 10
			
Vrchol 7 je preskúmaný! Vyberieme z radu vrchol 8. Nemá neobjavených susedov.	Vrchol 8 je preskúmaný! Vyberieme z radu vrchol 5. Nemá neobjavených susedov.	Vrchol 5 je preskúmaný! Vyberieme z radu vrchol 10. Nemá neobjavených susedov.	Vrchol 10 je preskúmaný! <b>Koniec prehľadávania.</b>
Rad: 5 10	Rad: 10	Rad: prázdny	

Napišeme procedúru `bfs`, ktorá prehľadá graf  $G$  z počiatočného vrcholu  $s$ . V programe použijeme vopred pripravené unity pre prácu s grafom (*GrafUnit*) a s údajovou štruktúrou rad (*FifoUnit*)<sup>4</sup>.

<sup>4</sup> Zdrojový kód všetkých v práci používaných unitov čitateľ nájde v prílohe.



Stav jednotlivých vrcholov evidujeme počas prehľadávania v rovnomennom jednorozmernom poli. Užitočné tiež bude zapamätať si pre každý vrchol dve ďalšie informácie: „kto ho objavil“ t. j. jeho predchodcu (v poli *p*) a vzdialenosť od počiatočného vrcholu prehľadávania *s* (v poli *d*).

```

program BreadthFirstSearch;
uses GrafUnit, FifoUnit;
type stavý = (Neobjavený, Objavený, Preskumany);

var G: graf;
    stav: array [1..MAXV] of stavý;
    p: array [1..MAXV] of integer;
    d: array [1..MAXV] of integer;

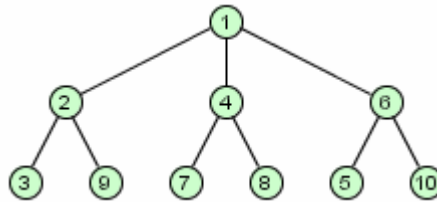
procedure bfs (var G: graf; s: integer);
var q: fifo;
    i, u: integer;
begin
    for i:= 1 to G.n do begin                                {inicializácia}
        stav[i]:= Neobjavený;
        p[i]:= -1;
        d[i]:= -1;
    end;
    stav[s]:= Objavený;                                         {prehľadávať začíname z vrcholu s}
    p[s]:= 0;                                                    {vrchol s nemá predchodcu}
    d[s]:= 0;                                                    {vzdialenosť z vrcholu s do s je rovná 0}
    nastav(q);                                                  {do pripraveného radu q vložíme prvý vrchol s}
    vlož(q, s);

    while not prázdný(q) do begin                                {kým nie je rad prázdny}
        u:= vyber(q);                                           {vyberieme z radu q vrchol u}
        vypis(u);
        for i:= 1 to G.deg[u] do                                {vyhľadáme jeho neobjavených susedov}
            if stav[G.adj[u,i]] = Neobjavený then begin
                stav[G.adj[u,i]]:= Objavený;                    {označíme objavený vrchol}
                p[G.adj[u,i]]:= u;                                {objavili sme ho z vrcholu u}
                d[G.adj[u,i]]:= d[u]+1;                          {je o 1 hranu ďalej ako vrchol u}
                vlož(q, G.adj[u,i]);                             {vložíme ho do radu q}
            end;
            stav[u]:= Preskumany;                                {vrchol u je preskúmaný}
        end;
    end;
Begin                                                            {hlavný program}
    citaj_graf(G, 'graf.txt', false);
    bfs(G, 1);
End.

```

Ak si v procedúre *vypis* necháme pre každý vrchol vypísať hodnoty *u*, *p*[*u*], *d*[*u*], tak pre Petersenov graf a volanie *bfs*(*G*,1) dostaneme výstup v tvare: 1 0 0, 2 1 1, 4 1 1, 6 1 1, 3 2 2, 9 2 2, 7 4 2, 8 4 2, 5 6 2, 10 6 2.

Algoritmus BFS „vybuduje“ z hrán, po ktorých do objavených vrcholov prichádzame tzv. *BFS strom* obsahujúci všetky vrcholy dosiahnuteľné z počiatočného vrcholu  $s$  (obr. 2.1). Všimnite si, že ak prečítame vrcholy v BFS strome po úrovniach, dostaneme poradie, v ktorom boli pri prehľadávaní navštívené. Z tohto stromu možno pre každý vrchol  $u$  priamo vyčítať aj spomínané hodnoty  $p[u]$  = rodič vrcholu v strome a  $d[u]$  = úroveň vrcholu v strome. Strom zároveň predstavuje jednu z kostier prehľadávaného grafu.



Obr. 2.1 BFS strom Petersenovho grafu

Z BFS stromu je tiež zrejmé, že hodnota  $d[u]$  predstavuje dĺžku *najkratšej cesty* z počiatočného vrcholu  $s$  do vrcholu  $u$  (cesta s najmenším počtom hrán je v neohodnotenom grafe najkratšou cestou). Vďaka poľu  $p$  môžeme túto najkratšiu cestu po skončení prehľadávania ľahko zrekonštruovať.

```

procedure najdi_cestu (s, c: integer);
var i: integer;
begin
    i := c;
    while i <> s do begin
        write(i, ' ');
        i := p[i];
    end;
    writeln(s);
end;

```

Pre volanie najdi\_cestu(1,10) dostaneme postupnosť: 10 6 1. Vrcholy sú ale usporiadané v opačnom poradí, od cieľa k štartu. Jednoduchou rekurzívnou procedúrou môžeme získať túto cestu v správnom poradí: 1 6 10.

```

procedure najdi_cestu2 (s, c: integer);
var i: integer;
begin
    if (s = c) then write(s, ' ')
    else begin
        najdi_cestu2(s, p[c]); {rekurzívne volanie}
        write(c, ' ');
    end;
end;

```

V našom príklade sme algoritmus BFS použili na *súvislý graf*. Čo sa stane, ak graf  $G$  nebude súvislý? Prehľadávanie skončí po preskúmaní všetkých vrcholov, ktoré sú dosiahnuteľné z počiatočného vrcholu  $s$ . Prehľadáme teda len ten komponent grafu, v ktorom sa nachádza vrchol  $s$ . Ak po skončení prehľadávania nájdeme v poli *stav* vrchol, ktorý zostal neobjavený, znamená to, že vstupný graf  $G$  nie je súvislý! *Jednou z dôležitých aplikácií algoritmov prehľadávania je práve testovanie grafu na súvislosť*. Mnohé iné grafové algoritmy totiž požadujú na vstupe súvislý graf. Napíšeme logickú funkciu na overenie súvislosti grafu:

```
function je_suvisly (var G: graf): boolean;  
var i: integer;  
begin  
  bfs(G, 1);  
  i := 1;  
  while (i <= G.n) and (stav[i] <> Neobjaveny) do inc(i);  
  je_suvisly := i > G.n;  
end;
```

## 2.3 Prehľadávanie grafu do hĺbky

Pri algoritme BFS sa skutočne postupuje najskôr „do šírky“ a až potom „do hĺbky“ grafu. Z práve navštíveného vrcholu sa totiž pokračuje najskôr do všetkých susedných vrcholov a až potom sa navštívia susedia susedov (pozri BFS strom na obr. 2.1).

Názov algoritmu napovedá, že v tomto prípade to bude naopak. Najskôr sa ide „do hĺbky“ a až potom „do šírky“ grafu. Algoritmus DFS je založený na rovnakom princípe ako *backtracking* (prehľadávanie s návratom). Kým sa dá, postupujeme v grafe od aktuálneho vrcholu k ďalšiemu zatiaľ nenavštívenému susednému vrcholu. Ak sa už takto pokračovať nedá t. j. dostaneme sa do takého vrcholu, v ktorého všetkých susedoch sme už boli (do „slepej uličky“), vrátime sa o krok späť a skúsime „inú cestu“. Nakoniec takto vyčerpáme všetky možnosti – navštívime všetky vrcholy grafu.

Z programátorského hľadiska stačí použiť na uchovávanie zatiaľ nepreskúmaných vrcholov namiesto radu *zásobník*. Keďže je však metóda DFS prirodzene rekurzívna, nemusíme žiadny vlastný zásobník vôbec programovať. O správne poradie spracovania vrcholov sa za nás „automaticky“ postará mechanizmus rekurzcie.



Prehľadávanie začneme vo vrchole 1. Má nenavštíveného suseda? Áno, dvoch: vrcholy 2 a 3. Vyberieme prvý z nich, vrchol 2 a pokračujeme rekurzívne ďalej: Má nenavštíveného suseda? Nie. Vrátime sa teda o krok späť do vrcholu 1 a prejdeme do ďalšieho zatiaľ nenavštíveného suseda – do vrcholu 3. Má vrchol 3 nejakých nenavštívených susedov? Má: vrcholy 4 a 5. Navštívime vrchol 4 a sme znovu v slepej uličke. Vrátime sa preto späť a pokračujeme vrcholom 5. Ten má tiež susedné vrcholy, v ktorých sme ešte neboli – vrcholy 6 a 7. Navštívime vrchol 6, vrátime sa späť a pokračujeme vrcholom číslo 7. Vrchol 7 má dvoch nenavštívených susedov – 8 a 9. Prejdeme do vrcholu 8. Tu máme tiež dve možnosti ako pokračovať – vrcholy 9 a 10. Navštívime vrchol 9. Vo vrchole 7 sme už boli, pokračujeme preto vrcholom 10. Vo vrchole 8 sme už boli, prejdeme teda ďalej do vrcholu 11. Tento vrchol má ešte dvoch nenavštívených susedov – vrcholy 12 a 13. Prejdeme do vrcholu 12, vrátime sa späť a ako posledný navštívime vrchol 13.

Algoritmus DFS môžeme rovnako ako algoritmus BFS použiť na testovanie súvislosti grafu. Ak by sme pri prehľadávaní súvislého grafu evidovali hrany, po ktorých prechádzame, našli by sme jednu z jeho kostier.

## 2.4 Komponenty súvislosti

Použitím algoritmov prehľadávania BFS alebo DFS ľahko zistíme, či je graf súvislý. Ak však súvislý nie je, potrebujeme často zistiť, z koľkých a akých komponentov sa skladá. Pri riešení väčšiny problémov totiž potrebujeme spracovať celý graf t.j. všetky komponenty súvislosti a nie len ten komponent, ktorý obsahuje vrcholy dosiahnuteľné z počiatočného vrcholu prehľadávania. Napíšeme procedúru, ktorá nájde všetky komponenty súvislosti zadaného grafu:

```
procedure komponenty_suvislosti (var G: graf);
var i, k: integer;
begin
  k:= 0;
  inicializuj_dfs(G);
  for i:= 1 to G.n do begin
    if stav[i] = Neobjaveny then begin
      inc(k);
      write('Komponent ', k, ': ');
      dfs(G, i);
      writeln;
    end;
  end;
end;
```

## 2.5 Cvičenia

1. Zrealizujte prehľadávanie do hĺbky na *Petersenovom grafe*. Aký výstup dostaneme pre volania:
  - a)  $\text{dfs}(G, 1)$
  - b)  $\text{dfs}(G, 4)$
  - c)  $\text{dfs}(G, 10)$ ?

2. Na grafe bludiska v *Hampton Courte* zrealizujte prehľadávanie do šírky. Aký bude výstup procedúry pre volania:
  - a)  $\text{bfs}(G, 1)$
  - b)  $\text{bfs}(G, 4)$
  - c)  $\text{bfs}(G, 8)$  ?

Nakreslite aj príslušné BFS stromy.

3. Je daný počet miest a zoznam priamych ciest spájajúcich niektoré dvojice miest. Žiadne dve cesty sa nekrižia mimo mesta. Zistite, či sa z každého mesta možno dostať do ktoréhokoľvek iného mesta.

Ak nie, rozdeľte všetky mestá do čo najväčších skupín tak, aby v danej skupine miest bolo možné cestovať z každého mesta do všetkých ostatných miest.

4. Najviac používanou službou Internetu je nepochybne služba WWW – informácie sú predsa v súčasnosti vo veľkej miere kľúčom k úspechu. Je jasné, že každý „surfujúci“ chce získať informáciu čo najrýchlejšie. Ak sa musí z úvodnej stránky nejakého rozsiahlejšieho webu dlho „preklikávať“ k tomu, čo ho skutočne zaujíma, stráca väčšinou záujem a „odchádza inam“.

Štruktúru každého webu (prepojenie stránok hypertextovými odkazmi) je možné reprezentovať grafom (bude orientovaný či neorientovaný?). Vo vstupnom súbore je najskôr uvedený celkový počet WWW stránok tvoriacich web a ďalej postupne pre jednotlivé stránky zoznam všetkých odkazov z príslušnej stránky na iné. V takto zadanom webe nájdite všetky také stránky, na ktoré sa z úvodnej stránky možno dostať najviac tromi „kliknutiami“ na hypertextový odkaz.

5. V tajnej organizácii pracuje  $N$  agentov. Každý z nich pozná len niekoľko ďalších spolupracovníkov, nie však všetkých. Medzi všetkými však existuje nejaké spojenie (ak nie priame, tak cez iného agenta). Agenti sa oslovujú zásadne svojimi číslami (sú očíslovaní číslami  $1, 2, \dots, N$ ). Kľúčovým agentom nazveme takého agenta, ktorého

úmrtie by spôsobilo, že sa organizácia rozpadne na aspoň dve časti, medzi ktorými neexistuje žiadne spojenie.

Napište program, ktorý načíta počet agentov  $N$  a pre každého agenta čísla spolupracovníkov, o ktorých vie. Potom zistí, či je v organizácii nejaký kľúčový agent.

*Návod:* V príslušnom grafe hľadáme artikuláciu t. j. vrchol, po odstránení ktorého sa súvislý graf rozpadne na aspoň dva komponenty.

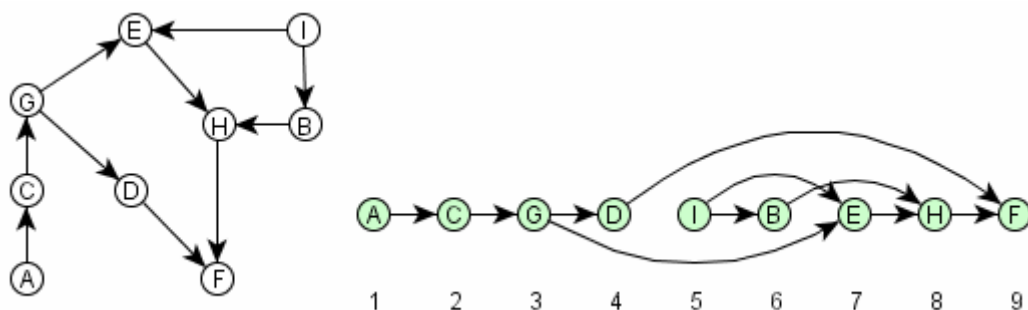
6. Napište nerekurzívnu procedúru na prehľadávanie grafu do hĺbky.

*Návod:* Použite vlastný zásobník. Pri vyberaní vrcholu zo zásobníka dajte pozor, aby ste znovu neskúmali vrchol, ktorý už preskúmaný je.

## 3 Topologické triedenie

### 3.1 Topologické usporiadanie vrcholov acyklického grafu

Orientovaný graf nazývame *acyklický*, ak neobsahuje cyklus (neexistuje v ňom dráha, ktorá by začínala aj končila v tom istom vrchole). Vrcholy takého grafu možno vždy očíslovať tak, aby každá hrana viedla z vrcholu s menším číslom do vrcholu s väčším číslom t. j. zľava doprava. Práve takého usporiadanie vrcholov acyklického orientovaného grafu získame *algoritmom topologického triedenia*. Výsledné usporiadanie vrcholov sa tiež zvykne nazývať *topologické*.



Obr. 3.1 Acyklický graf a jedno topologické usporiadanie jeho vrcholov

Algoritmus je založený na skutočnosti, že v každom acyklickom grafe musí existovať aspoň jeden vrchol, do ktorého nevstupuje žiadna hrana. Dôkaz tohto tvrdenia je veľmi jednoduchý. Ak by to tak nebolo, našli by sme v grafe taký vrchol  $v_1$ , do ktorého vstupuje hrana z nejakého vrcholu  $v_2$ , do ktorého tiež vstupuje hrana z nejakého vrcholu  $v_3$  atď. Keďže graf má konečný počet vrcholov, raz sa musí v tejto postupnosti niektorý vrchol zopakovať. To by znamenalo, že v grafe existuje cyklus a to je v acyklickom grafe nemožné.

Pri vytváraní topologického usporiadania vrcholov postupujeme takto: Nájdeme v grafe vrchol, do ktorého nevstupuje žiadna hrana (vezmeme ľubovoľný z nich). Ak taký vrchol nenájdeme, znamená to, že príslušný graf nie je acyklický a nemá zmysel vo výpočte ďalej pokračovať (algoritmus skončí chybovou správou). Nájdenný vrchol zaradíme na začiatok usporiadania a z grafu ho odstránime spolu so všetkými hranami, ktoré z tohto vrcholu vychádzajú. Ak bol pôvodný graf acyklický, tak aj graf, ktorý takto získame, je acyklický. Môžeme naň teda aplikovať rovnaký postup: nájdeme vrchol, do ktorého nevstupuje



žiadna hrana a zaradíme ho na druhé miesto vo výslednom usporiadaní. Potom ho z grafu odstránime spolu so všetkými z neho vychádzajúcimi hranami atď.

Takto postupne vyčerpáme všetky vrcholy grafu a nájdeme jedno topologické usporiadanie vrcholov. Nemusí to byť však jediné možné usporiadanie. Extrémom je prípad grafu s  $N$  vrcholmi, ktorý neobsahuje žiadne hrany – pre takýto graf existuje zrejme až  $N!$  rôznych topologických usporiadaní.

Skôr ako napíšeme program, zamyslime sa ešte nad tým, pri akých problémoch je tento algoritmus užitočný. Typickým príkladom aplikácie tohto algoritmu v bežnom živote je problém plánovania úloh pri riešení nejakého projektu. Jednotlivé úlohy reprezentujeme v grafe vrcholmi. Ak musí byť úloha  $u$  vyriešená pred úlohou  $v$ , z vrcholu  $u$  do vrcholu  $v$  povedie orientovaná hrana. Naším cieľom je usporiadať úlohy v projekte tak, aby pri začatí určitej úlohy boli všetky úlohy, ktoré sú nevyhnutné pre jej riešenie, už vyriešené.

Alebo iný príklad: Pri zostavovaní učebného plánu, je dôležité zohľadniť nadväznosť jednotlivých predmetov (*predmety = vrcholy*, *povinné nadväznosti = orientované hrany*). Topologické usporiadanie predmetov znamená zostavenie takého učebného plánu, v ktorom sú všetky predmety nevyhnutné na zvládnutie istého predmetu zaradené pred týmto predmetom.

A ešte jeden príklad: V programoch sa často stretávame s procedúrami, ktoré obsahujú volania iných procedúr (*procedúry = vrcholy*, *volania procedúr = orientované hrany*). Mnohé programovacie jazyky požadujú také usporiadanie deklarácií procedúr, aby sa v zápise programu nevyskytovali žiadne dopredné referencie.

Zhrnutie slovami matematika: Vstupom algoritmu je ľubovoľná konečná množina, na ktorej je definované čiastočné usporiadanie. Problém topologického triedenia spočíva v rozšírení tohto čiastočného usporiadania na lineárne.

V programe opäť použijeme pripravené unity *GrafUnit* a *FifoUnit*. Graf je popísaný vo vstupnom textovom súbore počtom vrcholov a zoznamom orientovaných hrán. Opäť budeme predpokladať, že vstup je korektný vzhľadom na dohodnutý formát a hodnoty konštánt definovaných v unite *GrafUnit*. Výsledné usporiadanie budeme vytvárať v poli *usp*.

Topologické triedenie zrealizuje na vstupnom grafe  $G$  rovnomenná funkcia. Pre každý vrchol je užitočné vedieť, koľko hrán do príslušného vrcholu vstupuje t. j. počet jeho predchodcov (v poli *indeg*). Umožní nám to rýchlo nájsť vrchol, do ktorého nevstupuje

žiadna hrana. Do pomocného zoznamu  $q_0$  (radu) budeme priebežne ukladať všetky takéto vrcholy.

Kým je rad neprázdny, opakujeme v cykle *while* tieto činnosti:

1. Vyberieme zo zoznamu  $q_0$  vrchol  $x$  a uložíme ho do poľa  $usp$ .
2. V poli *indeg* znížime všetkým vrcholom, do ktorých vedie hrana z vrcholu  $x$ , počet predchodcov. Ak pri tom „vyrobíme“ ďalší vrchol s nulovým počtom predchodcov, vložíme ho do zoznamu  $q_0$ .

Ak po skončení cyklu *while* zistíme, že sme nezaradili do poľa  $usp$  všetky vrcholy, znamená to, že graf na vstupe nebol acyklický! Túto informáciu vynesieme do hlavného programu pomocou návratovej hodnoty funkcie *topologicke\_triedenie*.

Všimnite si, že v programe žiadne vrcholy ani hrany z pôvodnej grafovej štruktúry „fyzicky“ neodstraňujeme. Túto operáciu sme nahradili evidovaním momentálneho počtu predchodcov jednotlivých vrcholov v poli *indeg*.

```
program TopSort;
uses GrafUnit, FifoUnit;

var G: graf;
    usp: array [1..MAXV] of integer;
    i: integer;

function topologicke_triedenie (var G: graf): boolean;
var indeg: array [1..MAXV] of integer;
    q0: fifo;
    x, y, i, j: integer;
begin
    for i:= 1 to G.n do indeg[i]:= 0;
    for i:= 1 to G.n do
        for j:= 1 to G.deg[i] do inc( indeg [G.adj[i,j]] );
    nastav(q0);
    for i:= 1 to G.n do
        if indeg[i] = 0 then vloz(q0, i);
    j:= 0;
    while not prazdny(q0) do begin
        j:= j+1;                                     {1}
        x:= vyber(q0);
        usp[j]:= x;
        for i:= 1 to G.deg[x] do begin               {2}
            y:= G.adj[x,i];
            dec( indeg[y] );
            if indeg[y] = 0 then vloz(q0, y);
        end;
    end;
    topologicke_triedenie:= j = G.n;
end;
```

```

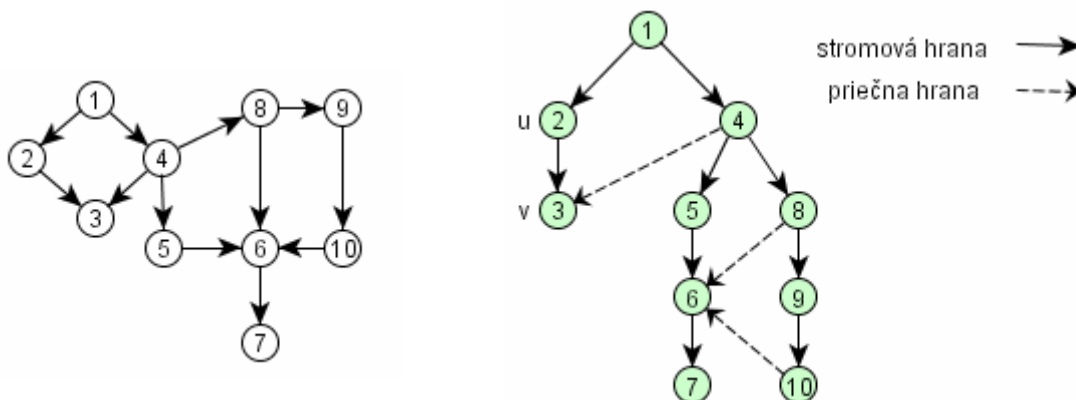
Begin
  citaj_graf(G, 'graf.txt', true);
  if topologicke_triedenie(G) then
    for i:= 1 to G.n do write(usp[i], ' ')
  else
    writeln('Chyba! Graf nie je acyklicky!');
End.

```

Odhadneme ešte zložitosť tejto implementácie algoritmu topologického triedenia. V každom z  $N$  krokov zaradíme do usporiadania práve jeden vrchol. Vrchol s nulovým počtom predchodcov nájdeme v konštantnom čase – máme ho totiž pripravený v rade  $q_0$ . Pri aktualizácii poľa *indeg* sa každá hrana použije práve raz. Výsledná časová výpočtová zložitosť je teda  $O(N+M)$ , kde  $M$  je počet hrán grafu.

### Topologické triedenie pomocou prehľadávania do hĺbky

Vrcholy acyklického orientovaného grafu možno topologicky usporiadať aj jednoduchou modifikáciou algoritmu prehľadávania do hĺbky. Pomocou neho ľahko nájdeme vrchol, z ktorého *nevychádzajú žiadne hrany*. Topologické usporiadanie budeme vytvárať „od konca“. Na správne miesto vrchol zaradíme už počas prehľadávania – v okamihu preskúmania príslušného vrcholu.



Obr. 3.2 Acyklický graf a jeho DFS strom

Pri prehľadávaní grafu do hĺbky vytvoria hrany, ktorými sme prichádzali do dovtedy nenavštívených vrcholov strom – tzv. *DFS strom* (podobne ako pri algoritme BFS). Tieto hrany nazývame *stromové*.

Ideme prehľadávať orientovaný graf (obr. 3.2). Ak pri skúmaní vrcholu  $u$  „narazíme“ na orientovanú hranu  $(u, v)$ , máme len tri možnosti:

1. vo vrchole  $v$  sme ešte neboli (je neobjavený) – označíme ho teda ako objavený a rekurzívne spustíme DFS z tohto vrcholu. Vrchol  $v$  bude zrejme preskúmaný skôr ako vrchol  $u$ . Po skončení DFS na vrchole  $v$  je predsa potrebné ešte dokončiť DFS na vrchole  $u$  t. j. prehľadať ďalší podstrom (ak existuje).
2. vo vrchole  $v$  sme už predtým boli (je objavený) – hrana  $(u, v)$  sa v tomto prípade nazýva *spätná* (je to hrana, ktorá vedie späť do nejakého predka daného vrcholu v DFS strome). V acyklickom grafe taká ale byť nemôže. Spätná hrana by znamenala, že graf obsahuje cyklus. Tento prípad môžeme teda z našich úvah vylúčiť.
3. vrchol  $v$  je už preskúmaný, čo znamená, že bol preskúmaný skôr ako vrchol  $u$ . Takúto hranu nazývame *priečna* (táto hrana nesmeruje do žiadneho predka ani potomka vrcholu  $v$  v DFS strome). V orientovanom grafe môže priečna hrana smerovať len sprava doľava (prečo?).

Ak bol vrchol  $v$  pri prehľadávaní preskúmaný ako prvý, vzhľadom na *bod 2* to znamená, že z neho určite nevychádzajú žiadne hrany. V topologickom usporiadaní ho preto môžeme dať na posledné miesto. Z grafu ho môžeme odstrániť. Toto „odstránenie“ zabezpečíme tak, že označíme vrchol  $v$  ako preskúmaný. Vrchol, ktorý bol preskúmaný ako druhý v poradí, je buď

- a) priamo vrchol  $u$ , z ktorého sme do vrcholu  $v$  prišli, alebo
- b) vrchol  $w$  z ďalšieho podstromu s koreňom vo vrchole  $u$ . Ak taký podstrom existuje, všetky vrcholy v tomto podstrome budú ale preskúmané skôr ako vrchol  $u$ !

V oboch prípadoch je teda jasné, že v momente preskúmania vrcholu  $u$ , všetky hrany z  $u$  smerujú práve do tých vrcholov, ktoré už boli do výsledného usporiadania zaradené. Vrchol  $u$  teda v prípade a) môžeme umiestniť na predposledné miesto, v prípade b) na prvé voľné miesto pred vrcholy z príslušného podstromu. Vrchol, ktorý bol preskúmaný ako posledný, sa takto dostane vo výslednom usporiadaní na prvé miesto.

```

program TopSort2;
uses GrafUnit;
type stav = (Neobjaveny, Objaveny, Preskumany);

var G: graf;
    usp: array [1..MAXV] of integer;
    stav: array [1..MAXV] of stav;
    i, t: integer;

procedure inicializuj_dfs (var G: graf);
var i: integer;
begin
    for i:= 1 to G.n do stav[i]:= Neobjaveny;
end;

procedure dfs (var G: graf; u: integer);
var v: integer;
begin
    stav[u]:= Objaveny;
    for v:= 1 to G.deg[u] do begin
        if stav[G.adj[u,v]] = Neobjaveny then begin
            dfs(G, G.adj[u,v])
        end
        else if stav[G.adj[u,v]] = Objaveny then begin
            writeln('Chyba! Graf nie je acyklicky!');
            halt;
        end;
    end;
    stav[u]:= Preskumany; usp[t]:= u; dec(t);
end;

Begin
    citaj_graf(G, 'graf.txt', true);
    t:= G.n;
    inicializuj_dfs(G);

    for i:= 1 to G.n do begin
        if stav[i] = Neobjaveny then dfs(G, i);
    end;

    for i:= 1 to G.n do write(usp[i], ' ');    {výsledné usporiadanie}
End.

```

V príklade na obr. 3.2 sme začali prehľadávať graf z vrcholu, z ktorého existuje dráha do všetkých ostatných vrcholov. Jediným volaním procedúry `dfs` navštívime všetky vrcholy grafu. Ak by sme ale začali graf prehľadávať napr. z vrcholu 4, do vrcholov 1 a 2 by sme sa nedostali. Tento graf totiž *nie je silne súvislý* (existujú v ňom vrcholy, medzi ktorými neexistuje spojenie). Preto postupujeme pri prehľadávaní podobne, ako by sme hľadali komponenty súvislosti neorientovaného grafu.

Na záver: V [17] nájdete programovú implementáciu prvého algoritmu topologického triedenia, v ktorej je graf reprezentovaný dynamickou údajovou štruktúrou jednosmerný zoznam jednosmerných zoznamov hrán.

### 3.2 Cvičenia

1. V záverečný deň letnej olympiády sa beží maratón. Bežci sú na dresoch označení číslami  $1, 2, \dots, P$ . Hugo má  $N$  kamarátov a všetci sú veľkými fanúšikmi atletiky. Každý z nich sa pred štartom pokúša predpovedať výsledok. Presné poradie pretekárov v cieľi sa ale neodváža vopred odhadovať žiaden z nich. Všetky ich vyhlásenia sú formulované v tvare: bežec  $i$  určite dobehne pred bežcom  $j$  ( $1 \leq i, j \leq P$ ). Spolu padlo  $V$  rôznych vyhlásení. Je teoreticky možné, aby mali po skončení maratónu všetci Hugovi kamaráti pravdu? Ak áno, v akom poradí by mohli maratónci do cieľa dobehnúť?
2. Študenti a pedagógovia nitrianskych gymnázií plánujú vytvoriť svetový rekord v dĺžke pochodovania v jednom zástupe. Podľa nariadenia ministerstva školstva musí byť každý učiteľ pri akejkoľvek školskej akcii schopný jediným pohľadom skontrolovať všetkých svojich študentov. Pri pochode musia teda všetci zúčastnení zachovať v zástupe také poradie, aby mal každý učiteľ všetkých svojich študentov pred sebou. Vstupný súbor obsahuje počet pochodujúcich  $N$ , za ktorými je vymenovaných  $K$  dvojíc čísel určujúcich kto má koho na starosti (prvé je číslo učiteľa, druhé je číslo študenta). Nie každý študent má v zástupe svojho učiteľa a niektorí učitelia zase pochodujú dobrovoľne bez svojich študentov. Napíšte program, ktorý zistí, v akom poradí sa majú zúčastnení na štarte do zástupu postaviť. Ak také poradie nemožno dosiahnuť, program to musí oznámiť.

## 4 Najkratšia cesta

Problém najkratšej cesty patrí ku klasickým problémom teórie grafov. V praxi majú algoritmy, ktoré ho riešia, široké využitie. Typický príkladom sú optimalizačné problémy v rôznych dopravných a komunikačných sieťach (hľadanie najkratšej príp. najlacnejšej cesty z mesta  $x$  do mesta  $y$ , smerovanie paketov medzi uzlami siete Internet a pod.).

V neohodnotenom grafe najkratšiu cestu medzi dvoma vrcholmi predstavuje cesta s minimálnym počtom hrán. Vieme ju nájsť v lineárnom čase pomocou algoritmu BFS. V ohodnotených grafoch ale rozumieme najkratšou cestou takú cestu, na ktorej je minimálny súčet ohodnotení hrán. Tu nám už algoritmus BFS nepomôže (v ohodnotenom grafe najkratšia cesta medzi dvoma vrcholmi nemusí byť zároveň cestou s najmenším počtom hrán!).

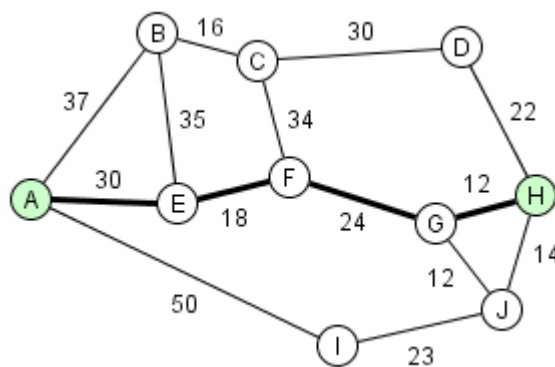
Ako ohodnotenia hrán sa najčastejšie používajú kladné príp. nezáporné čísla. Ak pripustíme aj záporné ohodnotenia, nesmie v grafe existovať *záporný cyklus* (súčet ohodnotení hrán takého cyklu je záporný). V grafe so záporným cyklom nemá zmysel najkratšiu cestu vôbec hľadať. Ak je totiž možné dostať sa z počiatočného vrcholu do niektorého vrcholu záporného cyklu, potom ku ktorejkoľvek „najkratšej“ ceste vždy nájdeme cestu ešte kratšiu. Stačí prejsť po hranách záporného cyklu dookola toľkokrát, aby súčet ohodnotení dostatočne klesol.

Na hľadanie najkratšej cesty medzi dvoma vrcholmi príp. všetkými dvojicami vrcholov v ohodnotenom grafe existujú rôzne špeciálne algoritmy. Zväčša „fungujú“ rovnako pre orientované i neorientované grafy. Ktorý z algoritmov sa pri riešení konkrétneho problému použije závisí aj od toho, čo o danom grafe a ohodnoteniach jeho hrán vieme.

### 4.1 Dijkstrov algoritmus

Najkratšiu cestu medzi dvoma vrcholmi grafu s hranami ohodnotenými nezápornými číslami, nájdeme pomocou algoritmu, ktorého autorom je *E. W. Dijkstra*.

Graf na obr. 4.1 reprezentuje mapu s niekoľkými mestami a cestnou sieťou, ktorá je medzi nimi vybudovaná. Ohodnotenia hrán predstavujú dĺžku príslušných cestných úsekov v kilometroch. Z mesta  $A$  chceme cestovať autom do mesta  $H$ . Benzín je drahý a času tiež nie je nazvyš, zaujíma nás teda najkratšia možná cesta. Ako na to?



Obr. 4.1 Ohodnotený neorientovaný graf a najkratšia cesta z vrcholu A do vrcholu H

Uvažujme všeobecne o najkratšej ceste z počiatočného vrcholu  $s$  do cieľového vrcholu  $c$ . Dĺžka najkratšej cesty z vrcholu  $s$  do toho istého vrcholu  $s$  sa zrejme rovná  $0$ . Ak sú vrcholy  $s$  a  $x$  spojené hranou a táto hrana má spomedzi všetkých hrán vychádzajúcich z vrcholu  $s$  najmenšie ohodnotenie, tak práve táto hrana nutne predstavuje najkratšiu cestu z vrcholu  $s$  do vrcholu  $x$ . Ohodnotenia hrán sú totiž nezáporné čísla. Každá iná cesta z  $s$  do  $x$  vedúca cez nejaké ďalšie vrcholy by mohla byť len dlhšia (nanajvýš rovná, v prípade nulového ohodnotenia hrany).

Ak  $(s, \dots, x, \dots, c)$  je najkratšia cesta z vrcholu  $s$  do vrcholu  $c$ , tak zrejme aj  $(s, \dots, x)$  musí byť najkratšia cesta z vrcholu  $s$  do vrcholu  $x$ . Inak by bolo možné cestu  $(s, \dots, x, \dots, c)$  ešte skrátiť.

Práve na týchto jednoduchých myšlienkach je založený Dijkstrov algoritmus, ktorý postupne nájde najkratšie cesty z počiatočného vrcholu  $s$  do všetkých ostatných vrcholov, teda aj cieľového vrcholu  $c$ . Prejdeme teraz k jeho programovej implementácii.

V poli  $d$  si budeme pre každý vrchol priebežne pamätať dĺžku najkratšej cesty z vrcholu  $s$ , ktorú sme doposiaľ pre daný vrchol našli. Budeme rozlišovať, či je pre príslušný vrchol táto hodnota už *trvalá* (určuje výslednú dĺžku najkratšej cesty do tohto vrcholu) alebo je len *dočasná* (je možné, že sa ešte zmenší).

Na začiatku poznáme len najkratšiu cestu do počiatočného vrcholu  $s$ , nastavíme teda hodnotu  $d[s]$  na  $0$ . Všetkým ostatným vrcholom grafu priradíme dočasne hodnotu „nekonečno“ (zatiaľ nepoznáme do nich žiadne najkratšie cesty). Celý výpočet bude prebiehať dovtedy, kým nebudú mať všetky dostupné vrcholy priradenú trvalú hodnotu. Na evidenciu vrcholov, ktoré už majú trvalú hodnotu, použijeme množinu  $T$  (mohli by sme



použiť aj pole s prvkami typu *boolean*). V poli *p* si pre každý vrchol pamätáme predchodcu vrcholu na najkratšej ceste (podobne ako pri algoritme BFS).

V cykle *while* sa opakujú tri činnosti:

1. Do množiny *T* pribudne nový vrchol *u*, pre ktorý sme práve našli najkratšiu cestu.
2. Prejdeme teda všetky vrcholy, do ktorých vedie z vrcholu *u* hrana a overíme, či nemožno cestu do nich skrátiť cez vrchol *u*. Ak áno, „vylepšíme“ príslušnú hodnotu v poli *d*. Aktualizujeme aj predchodcu v poli *p*.
3. Nájdeme vrchol, ktorý má najmenšiu dočasnú hodnotu t. j. ďalší vrchol *u* pre bod 1.

V programe pracujeme s ohodnoteným grafom – použijeme pripravený unit *WGrafUnit* (z angl. *weighted* = ohodnotený). Definovali sme nový typ *wgraf* s hranami ohodnotenými vo všeobecnosti celými číslami typu *integer*. Uvedieme úsek definícií potrebných konštánt a typov v unite *WGrafUnit*:

```
const NEKONECNO = MAXINT;
      MAXV      = 100;
      MAXDEG    = 50;

type  hrana = record
      v: integer;      {koncový vrchol hrany}
      h: integer;      {ohodnotenie hrany}
    end;

      wgraf = record
      adj: array [1..MAXV, 1..MAXDEG] of hrana;
      deg: array [1..MAXV] of integer;
      n: integer;
      m: integer;
    end;
```

Procedúry a funkcie z pôvodného unitu *GrafUnit* si čitateľ ľahko upraví do podoby vhodnej pre ohodnotený graf sám. Vstupný súbor popisujúci graf bude mať rovnaký formát, pre každú hranu však okrem koncových vrcholov pribudne aj tretí údaj – ohodnotenie príslušnej hrany.

Výstupom programu bude tabuľka najkratších ciest z vrcholu *s* do všetkých ostatných vrcholov. Zaujímala nás ale hlavne cesta medzi vrcholmi *s* a *c* – vypíšeme ju preto celú. Využijeme už známu procedúru *najdi\_cestu2* (z kapitoly 2).

V každom kroku pribudne do množiny *T* práve jeden vrchol, cyklus *while* sa zopakuje najviac *N* krát. Body 2 a 3 sa realizujú v čase  $O(N)$ . Dijkstrov algoritmus teda pracuje s celkovou časovou výpočtovou zložitou  $O(N^2)$ .

```

program Najkratsia_cesta_Dijkstra;

uses WGrafUnit;

var G: wgraf;
    i,
    s, c: integer;           {počiatočný a koncový vrchol hľadanej cesty}

    p: array [1..MAXV] of integer; {predchodca na najkratšej ceste}
    d: array [1..MAXV] of integer; {momentálne ohodnotenie vrcholu}

procedure dijkstra (var G: wgraf; s: integer);

var T: set of 1..MAXV;      {množina vrcholov s už trvalou hodnotou d}
    u: integer;             {aktuálny vrchol}
    v, h: integer;          {vrchol a ohodnotenie hrany doň vedúcej}
    i, j, mind: integer;    {pomocné premenné}

begin
    for i:=1 to G.n do begin {inicializácia}
        d[i]:= NEKONECNO;
        p[i]:= -1;
    end;

    T:= [];
    d[s]:= 0;
    u:= s;

    while not (u in T) do begin
        T:= T + [u];           {1}
        for i:= 1 to G.deg[u] do begin {2}
            v:= G.adj[u,i].v;
            h:= G.adj[u,i].h;
            if (d[v] > d[u]+h) then begin
                d[v]:= d[u]+ h;
                p[v]:= u;
            end;
        end;

        u := s; mind:= NEKONECNO; {3}
        for i:= 1 to G.n do
            if not (i in T) and (d[i] < mind ) then begin
                mind:= d[i];
                u:= i;
            end;
        end;
    end;

    ...

Begin
    citaj_graf (G, 'graf.txt', false);
    readln(s, c)
    dijkstra(G, s);
    for i:= 1 to G.n do writeln(i:2, d[i]:10);
    najdi_cestu2(s, c);
End.

```

Dijkstrov algoritmus možno použiť aj pri hľadaní najkratšej cesty v neohodnotenom grafe namiesto spomínaného prehľadávania grafu do šírky. Aj takýto graf totiž môžeme považovať za ohodnotený, pričom všetky jeho hrany majú rovnaké ohodnotenie 1.

### Najkratšia cesta v acyklickom grafe

Jednoduchou modifikáciou Dijkstrovho algoritmu získame iný postup na hľadanie najkratšej cesty, ktorý je možný použiť aj v prípade grafov so záporne ohodnotenými hranami. Príslušný graf musí byť ale *acyklický*.

Ako už vieme, vrcholy acyklického grafu možno vždy topologicky usporiadať. Predpokladáme teda, že vrcholy vstupného grafu už máme takto usporiadané. Algoritmus pre acyklické grafy sa od klasického Dijkstrovho algoritmu líši iba tým, akým spôsobom sa vyberá vrchol, ktorého hodnota sa prehlási za trvalú. V tomto prípade nepôjde o vrchol s najmenšou dočasnou hodnotou. Vrcholy sa vyberajú jednoducho postupne jeden za druhým, v poradí podľa topologického usporiadania.

Správnosť tohto postupu plynie z toho, že dočasná hodnota takto zvoleného vrcholu už nemôže byť v budúcnosti cez žiadny iný vrchol znížená. Žiadna ďalšia hrana už vzhľadom na topologické usporiadanie vrcholov doň už vstupovať nemôže.

## 4.2 Algoritmus Bellman-Ford

Stručne naznačíme ešte iný známy algoritmus na hľadanie najkratšej cesty medzi dvoma vrcholmi ohodnoteného grafu s  $N$  vrcholmi a  $M$  hranami. Ohodnotenia hrán môžu byť na rozdiel od Dijkstrovho algoritmu aj záporné čísla. Aj keď to vyzerá zvlášť, aj s takými grafmi sa môžeme v praxi skutočne stretnúť. Záporné ohodnotená hrana môže predstavovať napr. stratu pri neúspešnej obchodnej transakcii na burze.

V inicializačnej fáze algoritmus nastaví dĺžku cesty do počiatočného vrcholu  $s$  na 0 a pre všetky ostatné vrcholy na „nekonečno“. Potom  $N-1$  krát prejde všetky hrany grafu a pre každú hranu  $(u, v)$  s ohodnotením  $h$  v prípade potreby „vylepší“ dĺžku zatiaľ najkratšej cesty do jej koncového vrcholu t.j. zrealizuje priradenie  $d[v] := \min(d[v], d[u] + h)$ .

Nakoniec ešte skontroluje, či v grafe existuje záporný cyklus t. j. overí, či je pre nejakú hranu  $(u, v)$  platí, že  $d[v] - d[u] > h$ . Ak áno, vráti hodnotu *false*, inak vráti hodnotu *true*.

Algoritmus nájde najkratšie cesty v čase  $O(N \cdot M)$ .

### 4.3 Algoritmus Floyd-Warshall

Vráťme sa k mape na obr. 4.1. Predstavme si teraz, že chceme otvoriť nový hypermarket a máme sa rozhodnúť, v ktorom meste ho postavíme. Najlepšie by bolo vybrať také mesto, aby nikto z obyvateľov ostatných okolitých miest nemal do nového obchodu príliš ďaleko.

Sformulujeme teraz tento problém v jazyku teórie grafov: V grafe hľadáme *centrálny vrchol* t.j. taký vrchol, ktorý má spomedzi všetkých vrcholov najmenšiu *excentricitu* (excentricitou vrcholu  $x$  rozumieme najdlhšiu spomedzi najkratších ciest z vrcholu  $x$  do všetkých ostatných vrcholov grafu). Aby sme vrchol s touto vlastnosťou našli, potrebujeme zrejme poznať *najkratšie cesty medzi všetkými dvojicami vrcholov daného grafu*. Tie by sme mohli v čase  $O(N^3)$  zistiť tak, že  $N$  krát použijeme Dijkstrov algoritmus (pre každý vrchol grafu raz).

Ukážeme si ale iný, veľmi elegantný algoritmus na nájdenie najkratších ciest z každého vrcholu do každého, tzv. *Floyd-Warshallov algoritmus*. Ohodnotenia hrán môžu byť v tomto prípade aj záporné čísla, v grafe však nesmie byť záporný cyklus.

Ohodnotený graf bude v tomto prípade vhodnejšie reprezentovať maticou  $D$  typu  $N \times N$ , pre ktorej prvky platí:

$$\begin{aligned}d_{ij} &= 0 && \text{ak } i = j \\d_{ij} &= h_{ij} && \text{ak } i \neq j \text{ a } (i, j) \text{ je hrana s ohodnotením } h_{ij} \\d_{ij} &= \infty, && \text{ak } i \neq j \text{ a hrana } (i, j) \text{ v grafe neexistuje}\end{aligned}$$

Túto maticu použijeme pri výpočte aj na uchovávanie medzivýsledkov. Po skončení algoritmu bude každý jej prvok  $d_{ij}$  obsahovať dĺžku najkratšej cesty z vrcholu  $i$  do vrcholu  $j$ .

Floyd-Warshallov algoritmus je typickým príkladom efektívneho algoritmu využívajúceho metodológiu *dynamického programovania*. Pri takomto prístupe sa najskôr snažíme zmenšiť rozmer pôvodnej (vyriešiť jednoduchší podproblém, triviálny prípad). Postupným zväčšovaním rozmeru úlohy potom nájdeme riešenie pôvodného problému využívajúc pri výpočte výsledky získané vyriešením úloh menších rozmerov.

Označme  $d_{ij}^{(k)}$  dĺžku najkratšej cesty z vrcholu  $i$  do vrcholu  $j$ , ktorá prechádza len cez vrcholy z množiny  $\{1, 2, \dots, k\}$ . Presnejšie, ak naozaj prechádza aj cez nejaké iné vrcholy, tak to môžu byť len vrcholy z tejto množiny. Ak existuje hrana  $(i, j)$ , tak zrejme  $d_{ij}^{(0)} = h_{ij}$ .

Naším cieľom je pre každú dvojicu vrcholov  $i$  a  $j$  určiť hodnotu  $d_{ij}^{(n)}$ . Uvažujme najkratšiu cestu z  $i$  do  $j$ , ktorá môže prechádzať len cez vrcholy z množiny  $\{1, 2, \dots, k\}$ . Máme len dve možnosti:

1) táto cesta *neprechádza* cez vrchol  $k$

2) táto cesta *prechádza* cez vrchol  $k$

1) Potom  $d_{ij}^{(k)} = d_{ij}^{(k-1)}$ .

2) To ale znamená, že najkratšiu cestu z  $i$  do  $j$  možno rozdeliť na dva úseky – ide o najkratšiu cestu z  $i$  do  $k$  a najkratšiu cestu z  $k$  do  $j$ . Celková dĺžka najkratšej cesty z  $i$  do  $j$  je potom  $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ .

Obe možnosti spojíme do jednej:  $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ .

Výpočet uskutočníme „zdola nahor“. Začneme s maticou  $D^{(0)}$  a postupne pre  $k = 1, \dots, n$  vypočítame maticu  $D^{(k)}$ . Pri výpočte novej matice  $D^{(k)}$  využijeme predchádzajúci výsledok t.j. maticu  $D^{(k-1)}$ .

```

program Najkratsia_cesta_Floyd_Warshall;
const MAXV = 100;
      NEKONECNO = MAXLONGINT div 2;

type   Matica = array [1..MAXV, 1..MAXV] of longint;

var    D: Matica;
      n: 1..MAXV; {počet vrcholov grafu}
      ...

procedure floyd (var D: Matica);
var i, j, k: integer;
      cez_k: longint;
begin
  for k:= 1 to n do
    for i:= 1 to n do
      for j:= 1 to n do begin
        cez_k := d[i,k] + d[k,j];
        if (cez_k < d[i, j]) then d[i,j] := cez_k;
      end;
    end;
  end;

Begin
  inicializuj_maticu(D);
  floyd(D);
  vypis_maticu(D);
End.

```

Algoritmus zjavne pracuje s časovou výpočtovou zložitou  $O(N^3)$ . Zdá sa, že sme si oproti možnosti  $N$  krát použiť Dijkstrov algoritmus veľmi nepolepšili. Zápis algoritmu je však na prvý pohľad krátky a natoľko jednoduchý, že v skutočnosti bude v praxi predsa len o niečo rýchlejší.

Výstupom uvedeného programu je hľadaná tabuľka najkratších vzdialeností. Uviedli sme len procedúru, ktorá zrealizuje samotný výpočet. Ostatné procedúry si čitateľ v rámci cvičenia ľahko doplní sám.

Na základe údajov získaných Floyd-Warshallovým algoritmom nájdite vhodné umiestnenie pre hypermarket z úvodu tejto podkapitoly.

#### 4.4 Cvičenia

1. Napíšte program, ktorý v zadanom ohodnotenom grafe nájde najkratšie cesty z počiatočného vrcholu do všetkých ostatných vrcholov použitím algoritmu Bellman-Ford.
2. Napíšte program, ktorý v zadanom ohodnotenom grafe nájde najkratšie cesty medzi všetkými dvojicami vrcholov. Nesmiete ale použiť algoritmus Floyd-Warshall.
3. Medzi  $N$  mestami očíslovanými číslami  $1, 2, \dots, N$  je vybudovaná sieť autobusových liniek. Každá linka spája niektorú dvojicu miest, autobusy stoja len na konečných staniciach. Medzi niektorými dvojicami miest nemusí priama linka existovať.

Zo vstupného súboru načítajte počet miest a zoznam existujúcich autobusových liniek. Pre každú linku sú v ňom okrem čísel koncových staníc uvedené ešte dve kladné čísla: *doba jazdy v minútach a cena cestovného lístka v korunách.*

Pre zvolenú dvojicu miest nájdite najlacnejšie autobusové spojenie z východzieho mesta do cieľového. Ak existuje viac navzájom rôznych najlacnejších spojení, vyberte z nich to, ktoré je najrýchlejšie.

*Návod:* V príslušnom grafe je každá hrana ohodnotená až dvoma rôznymi hodnotami. Tie sa pri hľadaní najkratšej cesty uplatňujú v pevne danom poradí: najprv cena lístka a až v prípade zhody rozhoduje doba jazdy. Základnú verziu Dijkstrovho algoritmu modifikujte tak, aby pri označovaní vrcholov zohľadňoval obe kritériá. Pre každý vrchol je teraz potrebné uchovávať dve hodnoty. Prvá hodnota predstavuje najnižšiu cenu lístka, za ktorú sa do tohto vrcholu dokážeme dostať a bude riadiť priebeh výpočtu. Druhá hodnota udáva najkratšiu dobu jazdy, za ktorú sa do tohto vrcholu

vieme dostať pri použití trasy s momentálne známou minimálnou cenou – tú udáva prvá hodnota.

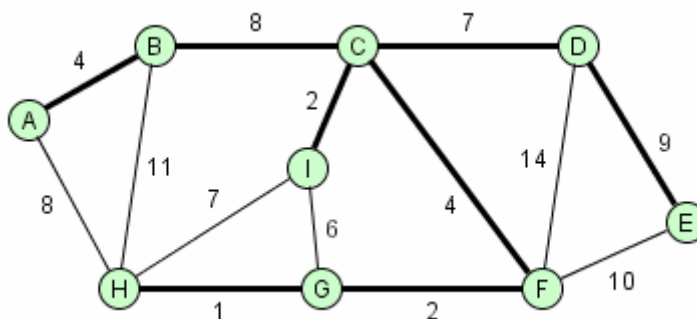
4. Na mape je vyznačených  $N$  letísk očíslovaných číslami  $1, 2, \dots, N$ . Vstupný súbor obsahuje pre každé letisko dvojicu čísel  $[x_i, y_i]$ , ktoré predstavujú súradnice  $i$ -teho letiska v rovine. Poznáme maximálny dolet lietadla (t. j. vzdialenosť, akú lietadlo preletí bez medzipristátia a doplnenia pohonných hmôt). Tiež je dané letisko, z ktorého sa štartuje a cieľové letisko, do ktorého sa letí. Vyriešte nasledujúce úlohy:

- Medzi dvoma zvolenými letiskami nájdite takú cestu, ktorá bude najkratšia vzhľadom na počet medzipristátí, bez ohľadu na jej celkovú dĺžku.
- Medzi dvoma zvolenými letiskami nájdite cestu s najkratšou celkovou dĺžkou letu bez ohľadu na počet medzipristátí.
- Medzi dvoma zvolenými letiskami nájdite takú cestu, ktorá bude najkratšia vzhľadom na celkovú dĺžku. Ak existuje viac rôznych ciest zistenej minimálnej dĺžky, vyberte z nich cestu s najmenším počtom medzipristátí.

*Návod:* Najskôr spočítajte vzdialenosti medzi všetkými dvojicami letísk a vylúčte také dvojice, ktorých vzdialenosť prevyšuje dolet lietadla. Ostatné vzdialenosti uložte. Tieto predstavujú dĺžky hrán v grafe, ktorého vrcholy reprezentujú dané letiská.

## 5 Minimálna kostra

Plánujeme výstavbu energetickej siete (elektrina, plyn a pod.), ktorou chceme prepojiť skupinu miest. Rozvody je potrebné položiť tak, aby viedli do každého mesta a aby boli náklady na vybudovanie celej siete čo najmenšie. Situáciu znázorníme súvislým neorientovaným ohodnoteným grafom, v ktorom vrcholy zodpovedajú jednotlivým uzlom v sieti t. j. mestám a hrany rozvodom medzi nimi. Ohodnotenia hrán predstavujú náklady na vybudovanie príslušného úseku vedení. Vyriešiť práve sformulovaný problém znamená nájsť minimálnu kostru tohto grafu<sup>5</sup>.



Obr. 5.1 Súvislý neorientovaný ohodnotený graf a jeho minimálna kostra

Problémom minimálnej kostry sa zaoberal už v roku 1926 český matematik *O. Borůvka* inšpirovaný skutočným projektom elektrovodnej siete, ktorá sa mala vybudovať na severe Moravy. Sieť, ktorú Borůvka navrhol, bola aj naozaj postavená.

Algoritmy na hľadanie minimálnej kostry nachádzajú v praxi uplatnenie nielen pri budovaní „najlacnejších“ energetických a telekomunikačných sietí. Môžu byť dokonca užitočné aj pri približnom riešení niektorých „ťažkých“ problémov ako je napr. známy *problém obchodného cestujúceho*. Navyše sú dôležitým príkladom takých „hladných“ algoritmov, ktoré vždy nájdu optimálne riešenie<sup>6</sup>.

V tejto kapitole si predstavíme dva algoritmy na hľadanie minimálnej kostry – *Kruskalov algoritmus (1956)* a *Primov algoritmus (1957)*. Budeme pracovať so *súvislými neorientovanými grafmi ohodnotenými kladnými číslami*. Ešte predtým si ale pripomenieme niektoré poznatky z teórie grafov.

<sup>5</sup> Namiesto *ohodnotenie* hrany môžeme hovoriť aj *cena hrany* a hľadať *najlacnejšiu* kostru.

<sup>6</sup> Pozri poznámku na konci kapitoly.



Súvislý graf, ktorý neobsahuje kružnice, nazývame strom. Ľubovoľné dva vrcholy stromu sú spojené jedinou cestou. Strom s  $N$  vrcholmi má  $N - 1$  hrán. Ak spojíme ľubovoľné dva nesusedné vrcholy stromu hranou, dostaneme graf, ktorý obsahuje práve jednu kružnicu.

Faktor grafu  $G$  je taký podgraf, ktorý obsahuje všetky vrcholy grafu  $G$ . *Kostrou súvislého grafu  $G$  rozumieme taký faktor grafu  $G$ , ktorý je strom.*

V každom súvislom grafe existuje aspoň jedna kostra. Ak chceme nájsť nejakú kostru súvislého grafu, stačí ak vynecháme z grafu čo najväčší možný počet hrán tak, aby graf zostal súvislý (odstraňované hrany budú zrejme ležať na nejakej kružnici).

*Minimálnou kustrou* ohodnoteného grafu nazývame kostru s najmenším súčtom ohodnotení všetkých jej hrán. Graf môže mať aj viac navzájom rôznych minimálnych kostier. Zväčša nás ale zaujíma len jedna z nich.

## 5.1 Kruskalov algoritmus

Aj keď algoritmus na hľadanie minimálnej kostry navrhol už v roku 1926 *O. Borůvka*, všeobecne známym sa stal problém minimálnej kostry až po publikovaní práce *J. B. Kruskala* v roku 1956.

Kruskalov algoritmus je veľmi jednoduchý. Hrany grafu usporiadame vzostupne podľa ich ohodnotení (od „najlacnejšej“ po „najdrahšiu“). Hrany potom preberáme v tomto poradí postupne jednu po druhej. Pre každú z nich skúmame, či ju môžeme do vytvárajúcej kostry zaradiť, teda či by jej pridaním do vytvárajúcej kostry nevznikla kružnica. Do kostry zaradíme len tie hrany, pridaním ktorých kružnica nevznikne. Výpočet skončí, keď sme do kostry zaradili potrebných  $N - 1$  hrán a prepojili tak všetkých  $N$  vrcholov grafu.

Graf bude v tomto prípade vhodné reprezentovať *zoznamom hrán*. V programe na to použijeme jednorozmerné pole záznamov, v ktorom si pre každú hranu zapamätáme jej koncové vrcholy a ohodnotenie. Toto pole usporiadame niektorým zo známych triediacich algoritmov. Ako však overíme, či by zaradením skúmanej hrany do kostry vznikla kružnica? Stačí si uvedomiť, že kostru grafu vytvárame postupne.

Na začiatku výpočtu v kostre nie je žiadna hrana, každý vrchol je izolovaný a tvorí samostatný komponent súvislosti. Pridaním každej ďalšej hrany do kostry sa počet komponentov zníži o 1. Ak by to tak nebolo, pridaná hrana by musela spájať takú dvojicu vrcholov, ktoré už sú z jedného komponentu súvislosti, čím by vytvorila kružnicu. Hrana,

ktorá môže byť pridaná do kostry teda vždy spája takú dvojicu vrcholov, ktoré sú z dvoch rôznych komponentov súvislosti a spojí tak tieto dva komponenty do jedného.

Jednoduchým spôsobom ako priebežne evidovať vytvárané komponenty súvislosti je použitie pomocného jednorozmerného poľa *komp*, v ktorom je pre každý vrchol uložené číslo komponentu, do ktorého momentálne vrchol patrí. Pred zaradením ďalšej hrany do kostry tak ľahko overíme, či jej koncové hrany nie sú z toho istého komponentu (v konštantnom čase). Ak nie sú, zaradíme hranu do kostry a informáciu o komponentoch v pomocnom poli aktualizujeme. Všetkým vrcholom nového („väčšieho“) komponentu, ktorý vznikol zaradením novej hrany do kostry, nastavíme v poli *komp* rovnaké číslo.

```

program Minimalna_kostra_Kruskal;

const MaxPV = 100;      {maximálny počet vrcholov grafu}
      MaxPH = 1000;     {maximálny počet hrán grafu}

type hrana = record
      u, v: integer;      {spojené vrcholy}
      h: integer;         {ohodnotenie hrany}
    end;

      wgraf = array [1..MaxPH] of hrana;

var G: wgraf;
      N: 1..MaxPV;          {počet vrcholov grafu}

procedure kruskal (var G: wgraf);
var komp: array [1..MaxPV] of integer;  {komponenty}
      i, j, p, k1, k2: integer;
begin
  for i:= 1 to N do komp[i]:= i;  {inicializácia - vrcholy sú izolované}
  i:= 0;                          {skúmaná hrana}
  p:= 0;                          {počet hrán zaradených do kostry}

  while p < N-1 do begin
    inc (i);
    k1:= komp[ G[i].u ];
    k2:= komp[ G[i].v ];
    if k1 <> k2 then begin          {hranu i zaradíme do kostry}
      inc(p);
      writeln(G[i].u, '-', G[i].v:5);
      for j:= 1 to N do
        if komp[j] = k2 then komp[j]:= k1;
      end;
    end;

end;

Begin
  vstup(G);
  tried(G);
  kruskal(G);
End.

```

Procedúry zabezpečujúce načítanie grafu zo vstupu a usporiadanie hrán vo vzorovom programe zámerne neuvádzame, čitateľ si v rámci cvičenia môže program dokončiť sám. V procedúre `kruskal` hrany tvoriace minimálnu kostru priamo vypisujeme na obrazovku.

Triedenie poľa, v ktorom sú uložené hrany grafu, môžeme zrealizovať napr. algoritmom *Quicksort* v čase  $O(M \cdot \log M)$ . Graf má najviac  $M$  hrán, výpočet bude mať teda najviac  $M$  krokov, z ktorých každý má zložitosť  $O(N)$ . Celková zložitosť uvedenej implementácie Kruskalovho algoritmu je tak  $O(M \cdot N)$ .

Pomocné pole *komp* zodpovedá údajovej štruktúre, ktorú nazývame *faktorová množina*. Ide o  $N$ -prvkovú množinu, ktorá je rozdelená do niekoľkých navzájom disjunktných tried, pričom každý prvok patrí do práve jednej triedy. Realizujú sa na nej dve základné operácie: zistenie, do ktorej triedy prvok patrí a zjednotenie dvoch tried. V našom programe sme ju implementovali veľmi jednoducho – pomocou jednorozmerného poľa. Dá sa ukázať, že pri efektívnejšej implementácii faktorovej množiny pomocou stromovej reprezentácie jednotlivých tried Kruskalov algoritmus spracuje graf s  $N$  vrcholmi a  $M$  hranami v čase  $O(M \cdot \log N)$ .

Z popísaného postupu je zrejmé, že Kruskalov algoritmus po konečnom počte krokov nájde nejakú kostru grafu. Bude to však vždy minimálna kostra?

Možno dokázať nasledujúce tvrdenie (tzv. *podmienku minimálnej kostry*), na ktorom je Kruskalov algoritmus založený: Kostra  $T$  je minimálna práve vtedy, ak pre ľubovoľnú hranu  $h$ , ktorá nepatrí kostre  $T$ , platí, že má väčšie alebo rovnaké ohodnotenie ako hrany kružnice, ktorá vznikne, keď hranu  $h$  pridáme ku kostre  $T$ . Dôkaz nájde čitateľ napr. v [11].

Kruskalov algoritmus do kostry v každom kroku vyberá najlacnejšiu hranu, ktorá nevytvorí kružnicu. Taká hrana, ktorá by vytvorila kružnicu je zbytočná a môžeme ju vynechať. Má totiž určite väčšiu nanajvýš rovnakú hodnotu ako všetky predchádzajúce do kostry vybrané hrany a vrcholy, ktoré by táto hrana spojila, sú už predsa nejakou cestou spojené vďaka „lacnejšej“ hrane vybranej do kostry v niektorom z predchádzajúcich krokov algoritmu. Kruskalov algoritmus je teda korektný.

## 5.2 Primov algoritmus

Kruskalov i Primov algoritmus konštruujú minimálnu kostru v princípe úplne rovnako. V každom kroku vyberajú do kostry najlacnejšiu hranu, ktorá nevytvorí kružnicu.

Kruskalov algoritmus zaradením každej ďalšej hrany do kostry vždy spojí dva rôzne komponenty súvislosti do jedného. Kostra sa teda vybuduje z viacerých podstromov, ktorých je na začiatku  $N$ . Primov algoritmus vybuduje kostru z jediného podstromu, ktorý sa postupne rozširuje pridávaním ďalších hrán. Na začiatku obsahuje tento podstrom iba počiatočný vrchol. Každou hranou pridanou do kostry pripojíme k tomuto podstromu nový vrchol grafu. Primov algoritmus je veľmi podobný Dijkstrovmu algoritmu na hľadanie najkratšej cesty.

V uvedenej programovej implementácii Primovho algoritmu budeme graf reprezentovať opäť pomocou zoznamov susedov, použijeme pripravený unit *WGrafUnit*.

Množina  $T$  bude v programe predstavovať podstrom, ktorého rozširovaním budujeme hľadanú minimálnu kostru a bude obsahovať všetky doposiaľ pripojené vrcholy. Kým budú v grafe nejaké nepripojené vrcholy, opakujeme dve činnosti: Nájdeme v grafe „najlacnejšiu“ hranu spomedzi všetkých hrán, ktoré spájajú nejaký už pripojený vrchol (t. j. vrchol z množiny  $T$ ) s nejakým zatiaľ ešte nepripojeným vrcholom. Potom túto hranu zaradíme do kostry a práve pripojený vrchol pridáme do množiny  $T$ .

V programe používame dve pomocné polia. V poli  $d$  si pre jednotlivé nepripojené vrcholy pamätáme hodnotu momentálne „najlacnejšej“ hrany spomedzi všetkých hrán, ktoré spájajú tento nepripojený vrchol s nejakým už pripojeným vrcholom z množiny  $T$ . Pole  $p$  obsahuje pre každý vrchol  $v$  číslo toho vrcholu z množiny  $T$ , ku ktorému môžeme vrchol  $v$  pripojiť pridaním hrany s hodnotou  $d[v]$ .

Celý výpočet zrealizujeme podobne ako pri Dijkstrovom algoritme v cykle *while*:

1. Najskôr do množiny  $T$  pribudne nový – práve pripojený vrchol  $u$ .
2. Potom preskúmame všetky jeho zatiaľ ešte nepripojené susedné vrcholy. Pre každého takého suseda  $v$ , ktorého možno pripojiť k podstromu  $T$  „lacnejšou“ hranou ako tou, o ktorej sme dovtedy vedeli, aktualizujeme hodnoty  $d[v]$  a  $p[v]$ .
3. Napokon vyhladáme v grafe ďalší vrchol, ktorý má byť pripojený, teda ďalšieho kandidáta pre bod 1.

```

program Minimalna_kostra_Prim;
uses WGrafUnit;

var G: wgraf;
    i: integer;
    p: array [1..MAXV] of integer;
    d: array [1..MAXV] of integer;

procedure prim (var G: wgraf; s: integer);
var T: set of 1..MAXV;
    u: integer;
    v, h: integer;           {vrchol a ohodnotenie hrany doň vedúcej}
    i, j, mind: integer;
begin
    for i:=1 to G.n do begin      {inicializácia}
        d[i]:= NEKONECNO;
        p[i]:= -1;
    end;

    T:= [];
    d[s]:= 0;
    u:= s;

    while not (u in T) do begin
        T:= T + [u];                {1}
        for i:= 1 to G.deg[u] do begin    {2}
            v:= G.adj[u,i].v;
            h:= G.adj[u,i].h;
            if not (v in T) and (d[v] > h) then begin
                d[v]:= h;
                p[v]:= u;
            end;
        end;

        u:= s; mind:= NEKONECNO;        {3}
        for i:= 1 to G.n do
            if not (i in T) and (d[i] < mind) then begin
                mind:= d[i];
                u:= i;
            end;
        end;
    end;

Begin
    citaj_graf(G, 'graf.txt', false);
    prim(G, 1);
    for i:= 2 to G.n do writeln(p[i], '-', i); {hrany minimálnej kostry}
End.

```

Primov algoritmus spracuje graf s  $N$  vrcholmi v čase  $O(N^2)$ . Pre grafy s veľkým počtom hrán, rádovo  $N^2$ , je vhodnejší Primov algoritmus. Pre riedke grafy je naopak efektívnejší Kruskalov algoritmus.

## Poznámka

Kruskalov i Primov algoritmus patria do triedy tzv. *hladných algoritmov* (z angl. *greedy algorithms*). Hladný algoritmus pri hľadaní riešenia „neuvažuje dopredu“, ale v každom kroku jednoducho zvolí najlepšiu z momentálne známych možností. V našom prípade v každom kroku vyberieme do kostry hranu s najmenšou hodnotou. Pri uplatňovaní „hladnej stratégie“ však musíme byť veľmi pozorný! Vždy totiž k optimálnemu riešeniu viesť nemusí.

Ak používame hladný algoritmus, musíme najskôr dokázať, že riešenie, ktoré nájde, je vždy optimálne. Zväčša stačí ukázať, že ak by sme v niektorom kroku výpočtu postupovali inak (t. j. v rozpore s „hladnou stratégiou“), optimálne riešenie by sme nedostali.

Ako sme už spomenuli, algoritmy riešiace problém minimálnej kostry sú príkladom takých hladných algoritmov, ktoré vždy nájdu optimálne riešenie. To isté platí aj pre Dijkstrov algoritmus na riešenie „problému najkratšej cesty“.

## 5.3 Cvičenia

1. Na mape zaostalej krajiny je vyznačených  $N$  miest. Vláda sa rozhodla krajinu elektrifikovať. V jednom z miest preto postavila elektrárňu. Zostáva dobudovať rozvodnú sieť. Od inžinierov vláda samozrejme požaduje najlacnejšie riešenie.

Na vstupe je  $N$  dvojíc čísel, ktoré predstavujú súradnice  $[x_i, y_i]$  mesta  $i$  na mape. Napíšte program, ktorý vypíše všetky také dvojice miest, medzi ktorými je potrebné položiť elektrické vedenie tak, aby boli všetky mestá elektrifikované a dĺžka vedenia bola čo najkratšia.

2. Nemenovaná štátna inštitúcia má v meste  $N$  budov a chce ich prepojiť počítačovou sieťou. Vedenie rozhodlo, že v  $K$  budovách zabezpečí pripojenie na Internet ( $1 \leq K \leq N$ ) a niektoré dvojice budov prepoja optickým káblom. Ak sa dá medzi dvoma budovami komunikovať cez niekoľko optických káblov (priamo alebo cez niektoré iné budovy), hovoríme, že sú v jednom komponente siete. Aby spolu mohli komunikovať budovy z rôznych komponentov siete, musí každý z týchto komponentov obsahovať aspoň jeden počítač pripojený na Internet.

Na vstupe sú dané čísla  $N$  a  $K$  a pre každú dvojicu budov kladné celé číslo predstavujúce cenu optického kábla medzi nimi. Napíšte program, ktorý vyberie, ktorých  $K$  budov je potrebné pripojiť na Internet a ktoré dvojice budov postačí spojiť

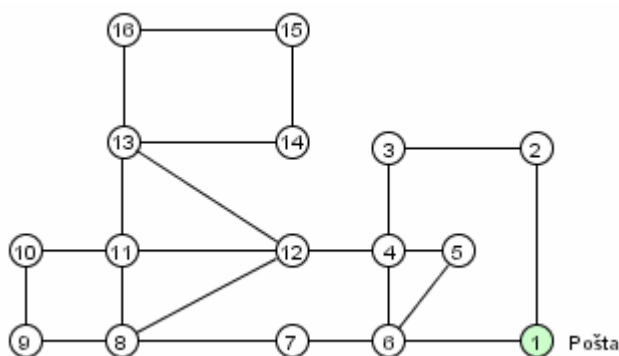
optickým káblom tak, aby každé dve budovy mohli spolu komunikovať a aby celkové náklady na položenie optických káblov boli čo najnižšie.

*Návod:* Pre  $K = 1$  nájdeme minimálnu kostru príslušného grafu. Pre  $K > 1$  už ale v grafe nie je potrebné pospájať všetky vrcholy, lebo na komunikáciu medzi budovami môžeme využiť aj Internet. Postačí teda, aby počítačová sieť pozostávala z  $K$  súvislých komponentov. V každom z nich vyberieme jeden vrchol, ktorý bude pripojený na Internet.

## 6 Eulerovský ťah

### 6.1 Hľadáme v grafe uzavretý eulerovský ťah

Poštárovi prideliť rajón, v ktorom má každý deň roznášať noviny. Graf reprezentujúci mapu prideleného rajónu je na obr. 6.1. Poštár musí pri roznáške prejsť každou z ulíc v rajóne, jeho obchôdzka začína i končí na pošte. Najlepšie by bolo, keby pri roznáške prešiel každou z ulíc práve raz.



Obr. 6.1 Neorientovaný graf reprezentujúci mapu poštárovho rajónu

„Poštárov problém“ môžeme preformulovať na problém, s ktorým sa už čitateľ určite stretol: *Nakreslite čiarový obrázok (a tým je aj graf na obr. 6.1) jedným ťahom*. V našom prípade navyše musíme začať a skončiť v tom istom vrchole. Je to vôbec možné? Ak áno, za akých podmienok?

Odpovede na tieto otázky poznáme vďaka švajčiarskemu matematikovi *Leonhardovi Eulerovi*, ktorý je považovaný za otca teórie grafov, už od 18. storočia. Euler inšpirovaný známym problémom *siedmich mostov mesta Kráľovec*<sup>7</sup>, prišiel na to, aký graf sa dá nakresliť jedným ťahom a aký nie. Uvedieme vetu, ktorú *Euler* dokázal, a na základe ktorej budeme pri hľadaní riešenia „poštárovho problému“ postupovať. Najskôr si však pripomeňme niekoľko pojmov:

<sup>7</sup>V roku 1763 *Euler* žil v meste *Kráľovec* (*Kaliningrad*, *Königsberg*), ktorým pretekala rieka *Pregel*. Rieka obmývala dva ostrovy, označme ich C a D. Medzi brehmi rieky, ktoré označíme A a B, a ostrovmi bolo spolu sedem mostov (dva mosty spájali A a C, jeden A a D, jeden C a D, dva B a C, jeden B a D). Obyvatelia pri nedeľných prechádzkach riešili tento problém: je možné prechádzku začať na pevnine, prejsť každým mostom práve raz a skončiť na tom istom mieste, kde začali? *Euler* dokázal, že to možné nie je. Graf, ktorým možno situáciu znázorniť, totiž nemá všetky vrcholy párneho stupňa (všetky štyri vrcholy sú dokonca nepárneho stupňa).



Ťahom nazývame taký sled, v ktorom sa žiadna hrana neopakuje. *Eulerovský ťah* je taký ťah, ktorý obsahuje každú hranu grafu práve raz. Ak začína a končí v tom istom vrchole, nazývame ho *uzavretý eulerovský ťah*. Ak je počiatočný vrchol rôzny od konečného, nazývame ho *otvorený eulerovský ťah*. Graf, ktorý obsahuje uzavretý eulerovský ťah, nazývame *eulerovský*.

### Veta (Euler, 1736)

V grafe existuje uzavretý eulerovský ťah práve vtedy, keď je súvislý a každý jeho vrchol má párny stupeň.

Uvedenú vetu teraz aj dokážeme. Dôkaz vykonáme v dvoch krokoch:

1.  $\Rightarrow$  Nech graf  $G$  má uzavretý eulerovský ťah (UEŤ). Je zrejmé, že graf  $G$  musí byť súvislý. Ľubovoľný vrchol rôzny od počiatočného vrcholu ťahu označme  $v$ . Ak pri „kreslení“ ťahu prideme do vrcholu  $v$  po nejakej hrane, po ďalšej hrane musíme z neho odísť. Pri každom prechode vrcholom  $v$  spotrebujeme práve dve hrany, koľkokrát doň prideme, toľkokrát z neho odídeme. To znamená, že vrchol  $v$  je incidentný s párnym počtom hrán a teda má párny stupeň. Z počiatočného vrcholu vychádza prvá hrana bez toho, aby sme doň najprv vošli. Ak sa do tohto vrcholu dostaneme počas ťahu, tak z neho aj vyjdeme. Posledná hrana ťahu končí v počiatočnom vrchole a „spári“ sa s prvou hranou ťahu. Aj počiatočný vrchol má teda párny stupeň.

2.  $\Leftarrow$  Predpokladajme, že graf  $G$  je súvislý a každý jeho vrchol má párny stupeň. Máme dokázať, že v takom grafe existuje UEŤ. Zvolíme ľubovoľný vrchol  $v$  a začneme „kresliť“ ťah. Postupne prechádzame po hranách grafu. Keď prideme k ľubovoľnému vrchole, musí tam byť hrana, ktorou môžeme odísť. Zastavíme sa až v počiatočnom vrchole  $v$  (ak už z neho nemôžeme pokračovať). Tento uzavretý ťah označme  $T_1$ . Ak sme v ňom použili všetky hrany, našli sme hľadaný UEŤ. V opačnom prípade nájdeme v grafe nejaký vrchol  $u$ , z ktorého ešte vychádzajú nepoužívané hrany (vzhľadom na párny stupeň každého vrcholu sú minimálne dve). Vo vrchole  $u$  začneme ťah  $T_2$  a pokračujeme, kým sa znova nevrátíme do vrcholu  $u$ . Ťahy  $T_1$  a  $T_2$  zjednotíme do jedného ťahu  $T$  nasledovným spôsobom: vyjdeme z počiatočného vrcholu  $v$  ťahu  $T_1$  a postupne do ťahu  $T$  pridávame vrcholy a hrany ťahu  $T_1$ , až kým sa nedostaneme do vrcholu  $u$ . Tento je počiatočným vrcholom ťahu  $T_2$ . Ťah  $T_2$  prejdeme celý, čím sa vrátíme späť do vrcholu  $u$ . Z vrcholu  $u$  potom pokračujeme po hranách a vrcholoch ťahu  $T_1$ . Dostaneme sa späť do

počiatočného vrcholu v ťahu  $T_1$ . Takto vytvoríme „väčší“ ťah  $T$ . Ak sa v grafe ešte stále nachádzajú vrcholy incidentné s nepoužitými hranami, pokračujeme rovnakým spôsobom dovtedy, kým nespotrebujeme všetky hrany grafu. Keďže graf  $G$  je konečný, po konečnom počte krokov UET určite nájdeme.

Z dokázanej vety priamo vyplýva, že v grafe existuje *otvorený eulerovský ťah* práve vtedy, ak je súvislý a práve dva vrcholy grafu majú nepárny stupeň (prečo?).

Eulerovský ťah možno v grafe nájsť viacerými spôsobmi. My využijeme postup, ktorý vychádza z algoritmu sformulovaného v dôkaze uvedenej Eulerovej vety. Napíšeme program, ktorým nájdeme jednotlivé „malé“ uzavreté ťahy  $T_i$ , z ktorých je už potom ľahké skonštruovať celý uzavretý eulerovský ťah.

V programe budeme pracovať so neorientovaným grafom, môžeme teda opäť zužitkovať pripravený unit *GrafUnit*. Najprv samozrejme musíme overiť, či je daný graf eulerovský. Logickou funkciou `je_eulerovsky` preto skontrolujeme stupne všetkých vrcholov (predpokladáme, že graf na vstupe je súvislý!).

Kľúčovou bude rekurzívna procedúra `prejdi`. Pomocou nej v programe postupne prechádzame po hranách a navštevujeme jednotlivé vrcholy grafu. Vrchol, ktorý navštívime, označíme v poli *navst* hodnotou *true*. Hrany, po ktorých prechádzame, z grafu odstraňujeme. V momente keď objavíme v grafe *kružnicu* t. j. keď prideme do vrcholu *od*, v ktorom sme už boli, „cúvame“ späť po tejto kružnici a pritom ju vypíšeme vrchol po vrchole na obrazovku.

Ak stupeň vrcholu *od* ešte neklesol na nulu t. j. sú s ním incidentné nejaké nespotrebované hrany, pokračujeme z neho ďalej. Až keď bude stupeň vrcholu *od* rovný nule, našli sme jeden čiastkový ťah  $T_i$ . V cykle *for* v hlavnom programe sa potom presunieme na ďalší vrchol, ktorý má nenulový stupeň. Ak taký ešte existuje, zavoláme preň procedúru `prejdi` a nájdeme príslušný uzavretý ťah  $T_{i+1}$  atď.

```

program Uzavrety_eulerovsky_tah;
uses GrafUnit;

var G: graf;
    navst: array [1..MAXV] of boolean;
    i, j: integer;

function je_eulerovsky (var G: graf): boolean;
var u: integer;
begin
    je_eulerovsky:= true;
    for u:= 1 to G.n do
        if G.deg[u] mod 2 = 1 then begin
            je_eulerovsky:= false;
            break;
        end;
    end;
end;

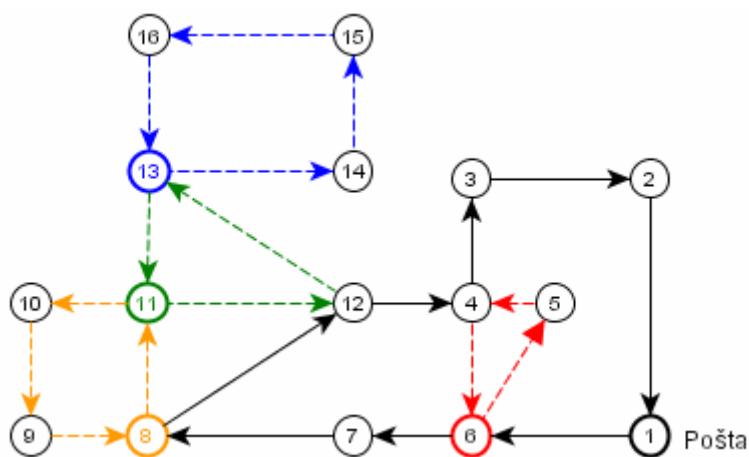
procedure prejdi (od: integer; var posl: integer);
var kam: integer;
begin
    if navst[od] then begin
        posl:= od;
        write(od, ' ');
    end
    else
        repeat
            navst[od]:= true;
            kam:= G.adj[od, G.deg[od]];
            zrus_hranu(G, od, kam, false);
            prejdi (kam, posl);           {rekurzívne volanie}
            navst[od]:= false;
            if posl<>od then begin
                write(od, ' ');
                exit;
            end;
            writeln(od);
        until G.deg[od]=0;
end;

Begin
    citaj_graf(G, 'graf.txt', false);
    vypis_graf(G);

    if je_eulerovsky(G) then begin
        for i:= 1 to G.n do navst[i]:= false;
        for i:= 1 to G.n do begin
            if G.deg[i]>0 then prejdi(i, j);
        end;
    end
    else writeln('Graf nie je eulerovsky!');

End.

```



Obr. 6.2 Grafická interpretácia postupu realizovaného procedúrou `prejdi`

Procedúra `prejdi` je z programátorského hľadiska veľmi zaujímavá, ale i náročná. Má dva parametre (jeden je volaný hodnotou a jeden odkazom), obsahuje cyklus a v ňom rekurzívne volanie. Na prvý pohľad tak nemusí byť každému celkom jasné, ako vlastne „funguje“.

Odporúčame čitateľovi, aby využil možnosti, ktoré mu poskytuje vývojové prostredie, v ktorom programuje, a pre viac rôznych grafov krokoval celý výpočet sledujúc pri tom zmeny hodnôt jednotlivých premenných a nahradzovanie parametrov pri rekurzívnych volaniach. Dôležité je tiež uvedomiť si, akým spôsobom pracuje procedúra `zrus_hranu` z unitu *GrafUnit*.

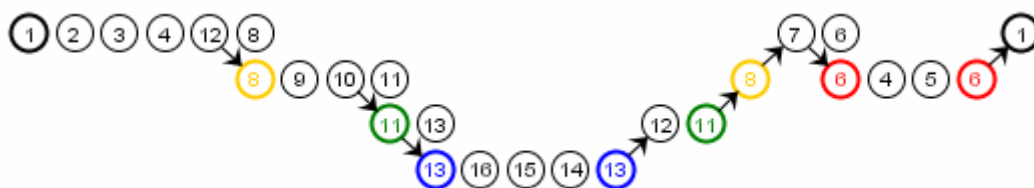
Na grafe poštárovho rajónu naznačíme ako v programe postupujeme a ako chápeme získaný výstup (obr. 6.2):

Z vrcholu 1 prideme cez vrcholy 6, 5, 4 opäť do vrcholu 6, kde objavíme červenú kružnicu. Vrátime sa teda po nej naspäť do vrcholu 6 a pokračujeme ďalej cez vrcholy 7, 8, 11, 12, 13, 14, 15, 16 a vo vrchole 13 objavíme modrú kružnicu. Vrátime sa po nej späť do vrcholu 13, prejdeme do vrcholu 11 a tu objavíme zelenú kružnicu. Vrátime sa po nej cez 13, 12 späť do vrcholu 11 a pokračujeme vrcholom 10, 9 a 8. Vo vrchole 8 objavíme žltú kružnicu. Vrátime sa po nej naspäť a potom pokračujeme z vrcholu 8 do vrcholu 12, 4, 3, 2 a 1, kde nakoniec objavíme čiernu kružnicu. Vrátime sa po nej naspäť do vrcholu 1.

Na obr. 6.2 je pre každú z objavených kružníc príslušnou farbou zvýraznený aj vrchol, príchodom do ktorého sme túto kružnicu v grafe objavili. Keď prideme do takéhoto

vrcholu druhýkrát, obrátime sa a v opačnom smere vypíšeme vrcholy, ktoré túto kružnicu tvoria, na výstup.

Z nájdených kružníc v grafe vytvoríme hľadaný uzavretý eulerovský ťah spôsobom ako na obr. 6.3. Zľava doprava prečítame vrcholy, ktorými pri kreslení hľadaného ťahu prechádzame. Začneme vo vrchole 1. Každý vrchol v ktorom začína kružnica nahradíme touto kružnicou – prejdeme po nej a pokračujeme ďalej doprava v „smere šípky“.



Obr. 6.3 Grafická interpretácia výsledku získaného pomocou procedúry prejsť

## 6.2 Cvičenia

1. Odhadnite časovú výpočtovú zložitosť programu. Bude uvedený program pracovať aj pre neorientované grafy s násobnými hranami a slučkami?
2. Program, ktorý sme uviedli v tejto kapitole, vypisuje na výstup všetky kružnice, ktoré pri prechádzaní po grafe objavuje. Samotné zjednotenie do hľadaného uzavretého eulerovského ťahu ale nezrealizuje. Modifikujte uvedený program (alebo napíšte iný) tak, aby na výstupe oznámil celkový výsledok.
3. K dispozícii máme  $N$  kameňov domina. Je možné uložiť ich tak, aby vytvorili uzavretý „kruh“? Vytvorte program, ktorý na vstupe prečíta počet kameňov  $N$  a pre každý kameň dvojicu navzájom rôznych čísel  $\{i, j\}$ , ktorými je tento kameň označený. Potom overí, či je možné kamene vedľa seba uložiť podľa podmienok úlohy. Ak nie, tak program navrhne, ktoré ďalšie kamene by sa mali k pôvodným kameňom pridať, aby to možné bolo.

Pripomenieme pravidlá domina: Každý hrací kameň je rozdelený na dve polovice a každá z nich je označená číslom. Dva kamene môžu byť položené vedľa seba len ak sa dotýkajú polovicami s rovnakým číslom.

4. Iný algoritmus – tzv. *Fleuryho algoritmus* – na nájdenie uzavretého eulerovského ťahu (UEŤ) v danom eulerovskom grafe, čitateľ nájde napr. v [3]. Sformulujeme ho len slovne:

- I. Konštrukciu UEŤ začneme v ľubovoľnom vrchole grafu, prejdenu hranu označíme.
- II. Pri konštrukcii UEŤ nikdy nevyberáme ako nasledujúcu tú hranu, ktorá je takým mostom, že po jej odstránení by sa graf pozostávajúci z neoznačených hrán a príslušných vrcholov rozpadol na netriviálne komponenty alebo netriviálny komponent a vrchol, v ktorom sme konštrukciu UEŤ začali.

Zamyslite sa nad tým, ako by ste tento algoritmus naprogramovali.

KONIEC ☺

## Literatúra

- [1] BENTLEY, J.: Perly programovania. Bratislava : Alfa, 1992. 216 s. ISBN 80-05-01056-7
- [2] BOSÁK, J.: Grafy a ich aplikácie. Bratislava: Alfa, 1980. 176 s.
- [3] FRONC, M.: Teória grafov. Žilina : VŠDS, 1993.
- [4] HVORECKÝ, J. – FRANEK, M.: Algoritmy pre 4. ročník gymnázií. Bratislava : SPN, 1987. 104 s.
- [5] KUČERA, L.: Kombinatorické algoritmy. Praha: SNTL, 1983. 288 s.
- [6] MORRIS, J: Data Structures and Algorithms (University of Western Australia). <http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/index.html> (16. 8. 2004)
- [7] NEČAS, J. Grafy a jejich použití. Praha : SNTL, 1978. 192 s.
- [8] NIEPEL, L.: Kombinatorika a teória grafov II. Bratislava : Matematicko-fyzikálna fakulta UK, 1991. 116 s. ISBN 80-223-0400-X
- [9] PLESNÍK, J.: Grafové algoritmy. Bratislava : VEDA, 1983. 344 s.
- [10] SEDGEWICK, R.: Algoritmy v C. Praha : SoftPress, 2003. ISBN 80-86497-56-9
- [11] SEDLÁČEK, J.: Úvod do teorie grafů. 3. vyd. Praha: Academia, 1981. 272 s.
- [12] TÖPFLER, P.: Algoritmy a programovací techniky. Praha : Prometheus, 1995. 299 s. ISBN 80-85849-83-6
- [13] TÖPFLEROVÁ, D. – TÖPFLER, P.: Sbíрка úloh z programování. Praha : Grada, 1992. 104 s. ISBN 80-85424-99-1
- [14] VOLKER, C. – SCHWILL, A.: Lexikón informatiky. Bratislava : SPN, 1991. 544 s. ISBN 80-08-00755-9
- [15] VRBA, A.: Grafy. Bratislava : SPN, 1989. 56 s.
- [16] WINCZER, M.: Zbierka úloh KSP 1983 – 2000. Bratislava : Katedra vyučovania informatiky FMFI UK, 2001. <http://www.ksp.sk> (16. 8. 2004)
- [17] WIRTH, N.: Algoritmy a štruktúry údajov. 2.vyd. Bratislava : Alfa, 1989. 488 s. ISBN 80-05-00153-3
- [18] National Institute of Standards and Technology: Dictionary of Algorithms and Data Structures. <http://www.nist.gov/dads/> (16. 8. 2004)
- [19] Archív Matematickej olympiády – kategória P (zadania a vzorové riešenia predchádzajúcich ročníkov). <http://www.ksp.sk/mop/archiv/index.php3> (16.8. 2004)

## Prílohy

### Unit pre prácu s neohodnoteným grafom

```
unit GrafUnit;

interface

const MAXV    = 100;
      MAXDEG  = 50;

type graf = record
  adj: array [1..MAXV, 1..MAXDEG] of integer;
  deg: array [1..MAXV] of integer;
  n: integer;
  m: integer;
end;

procedure inicializuj_graf (var G: graf);
procedure citaj_graf      (var G: graf; subor: string; orient: boolean);
procedure pridaj_hranu    (var G: graf; x, y: integer; orient: boolean);
procedure zrus_hranu      (var G: graf; x, y: integer; orient: boolean);
procedure vypis_graf      (var G: graf);
function  existuje_hrana  (var G: graf; x, y: integer): boolean;

implementation

procedure inicializuj_graf;
var i: integer;
begin
  G.n := 0;
  G.m := 0;
  for i:=1 to MAXV do G.deg[i]:= 0;
end;

procedure citaj_graf;
var i, x, y: integer;
    t: text;
begin
  inicializuj_graf(G);
  assign(t, subor);
  reset(t);
  readln(t, G.n);
  while not eof(t) do begin
    readln(t, x, y);
    pridaj_hranu(G, x, y, orient);
  end;
  close(t);
end;
```



```

function existuje_hrana;
var i: integer;
begin
    existuje_hrana:= false;
    for i:= 1 to G.deg[x] do
        if G.adj[x, i] = y then existuje_hrana:= true;
    end;

procedure pridaj_hranu;
var i: integer;
begin
    inc(G.deg[x]);
    G.adj[x, G.deg[x]]:= y;
    if (orient = false) then pridaj_hranu(G, y, x, true)
        else inc(G.m);
    end;

procedure zrus_hranu;
var i: integer;
begin
    for i:= 1 to G.deg[x] do
        if (G.adj[x, i] = y) then begin
            G.adj[x, i]:= G.adj[x, G.deg[x]];
            dec(G.deg[x]);
            if (orient = false) then zrus_hranu(G, y, x, true)
                else dec(G.m);
        end;
    exit;
    end;

procedure vypis_graf;
var i, j: integer;
begin
    for i:= 1 to G.n do begin
        write(i:2, ': ');
        for j:= 1 to G.deg[i] do write(' ', G.adj[i, j]);
        writeln;
    end;
end;

Begin
End.

```

## Unit pre prácu s ohodnoteným grafom

```

unit WGrafUnit;

interface

const NEKONECNO = MAXINT;
        MAXV      = 100;
        MAXDEG    = 50;

```

```

type hrana = record
    v: integer;
    h: integer;
end;

    wgraf = record
    adj: array [1..MAXV, 1..MAXDEG] of hrana;
    deg: array [1..MAXV] of integer;
    n: integer;
    m: integer;
end;

procedure inicializuj_graf (var G: wgraf);
procedure citaj_graf    (var G: wgraf; subor: string; orient: boolean);
procedure pridaj_hranu (var G: wgraf; x,y: integer; orient: boolean; h: integer);
procedure zrus_hranu   (var G: wgraf; x,y: integer; orient: boolean);
procedure vypis_graf   (var G: wgraf);
function existuje_hrana (var G: wgraf; x, y: integer): boolean;

implementation

procedure inicializuj_graf;
var i: integer;
begin
    G.n := 0;
    G.m := 0;
    for i:=1 to MAXV do G.deg[i] := 0;
end;

procedure citaj_graf;
var i, x, y, h : integer;
    t: text;
begin
    inicializuj_graf(G);
    assign(t, subor);
    reset(t);
    readln(t, G.n);
    while not eof(t) do begin
        readln(t, x, y, h);
        pridaj_hranu(G, x, y, orient, h);
    end;
    close(t);
end;

procedure pridaj_hranu;
var i: integer;
begin
    inc(G.deg[x]);
    G.adj[x, G.deg[x]].v := y;
    G.adj[x, G.deg[x]].h := h;
    if (orient = false) then pridaj_hranu(G, y, x, true, h)
    else inc(G.m);
end;

```

```

function existuje_hrana;
var i: integer;
begin
    existuje_hrana:= false;
    for i:= 1 to G.deg[x] do
        if G.adj[x, i].v = y then existuje_hrana:= true;
    end;

procedure zrus_hranu;
var i: integer;
begin
    for i:= 1 to G.deg[x] do
        if (G.adj[x, i].v = y) then begin
            G.adj[x, i]:= G.adj[x, G.deg[x]];
            dec(G.deg[x]);
            if (orient = false) then zrus_hranu(G, y, x, true)
                else dec(G.m);
        end;
        exit;
    end;

end;

procedure vypis_graf;
var i, j: integer;
begin
    for i:= 1 to G.n do begin
        write(i:2, ': ');
        for j:= 1 to G.deg[i] do write(' ', G.adj[i, j].v);
        writeln;
    end;

end;

Begin
End.

```

## Unit pre prácu s údajovou štruktúrou rad

```

unit FifoUnit;

interface

const  MAXPP = 1000;

type  fifo = record
    rad: array [1..MAXPP] of integer;
    zac: integer;
    kon: integer;
    poc: integer;
end;

procedure nastav    (var q: fifo);
procedure vloz      (var q: fifo; x: integer);
function  vyber     (var q: fifo): integer;
function  prazdny   (var q: fifo): boolean;
function  plny      (var q: fifo): boolean;

```

```

implementation

procedure nastav;
begin
    q.poc:= 0;
    q.zac:= 1;
    q.kon:= MAXPP;
end;

procedure vloz;
begin
    if plny(q) then begin
        writeln('Chyba: rad je plny !! ');
        halt;
    end
    else begin
        if q.kon = MAXPP then q.kon:= 1
        else inc(q.kon);
        q.rad [q.kon]:= x;
        inc(q.poc);
    end;
end;

function vyber;
begin
    if prazdny(q) then begin
        writeln('Chyba: rad je prazdny !!');
        halt;
    end
    else begin
        vyber:= q.rad [ q.zac ];
        if q.zac = MAXPP then q.zac:= 1
        else inc(q.zac);
        dec(q.poc);
    end;
end;

function prazdny;
begin
    prazdny:= q.poc = 0;
end;

function plny;
begin
    plny:= q.poc = MAXPP;
end;

Begin
End.

```