# 2.1. Úvod do teórie grafov (Graph theory)

Pokým matematika má dlhú a slávnu históriu, tak toto nie je prípad matematického oboru teórie grafov. Za prelomový rok sa tu považuje až rok 1736, keď L. Euler publikoval riešenie tzv. Problému Kalingradských mostov (Königsberg Bridge Problem). Dvesto rokov neskôr, roku 1936, napísal Dénes König prvú knihu o teórií grafov a odvtedy sa začala táto disciplína prudko rozvíjať.

### 2.1.1. Čo je to graf?

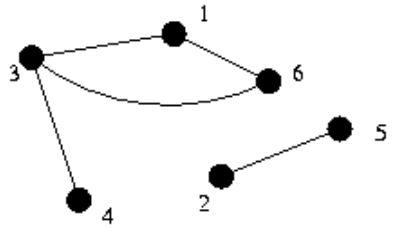
Formálne povedané, graf je:

- konečná neprázdna množina vrcholov(vertices) ***V*** a
- (aj prázdna) množina dvojíc vrcholov (hrán(edges)) ***E***

Vrcholy si môžeme predstaviť ako „stanovištia“. Množina všetkých vrcholov ***V*** je potom množina všetkých možných stanovíšť. Analogicky, hrany reprezentujú cesty medzi dvojicami týchto stanovíšť; množina ***E*** obsahuje všetky cesty medzi stanovišťami.

### 2.1.2. Reprezentácia

Grafy si obyčajne predstavujeme v tejto analógií. Vrcholy sú body alebo krúžky a hrany sú čiary medzi nimi.




V grafe na obrázku, ***V*** = {1,2,3,4,5,6} a ***E*** = {(1,3),(1,6),(2,5),(3,4),(3,6)}. Každý vrchol patrí do množiny ***V*** a každá hrana patrí do množiny ***E***. Niektoré vrcholy nemusia byť spojené žiadnou hranou, takéto vrcholy nazývame izolované.

Číselné hodnoty, špecifikujúce ich dĺžku alebo cenu, sú niekedy asociované s hranami, také grafy sa potom nazývajú hranovo ohodnotené(edge-weighted) (ohodnotené grafy(weighted graphs)). Číslo späté s hranou sa potom nazýva hodnota, alebo váha(weight). Rovnako to platí pre vrcholovo ohodnotené grafy.

### 2.1.3. Základné pojmy

Hrana sa nazýva slučka(self-loop), keď je tvaru (u,u). Graf na obrázku neobsahuje slučky. Graf sa nazýva jednoduchý(simple), keď neobsahuje slučky a násobné hrany (viac hrán medzi rovnakými vrcholmi). Graf, ktorý obsahuje slučky alebo násobné hrany sa nazýva multigraf(multigraph). My sa budeme zaoberať prevažne jednoduchými grafmi a teda, keď hovoríme graf, predpokladáme, že je jednoduchý.

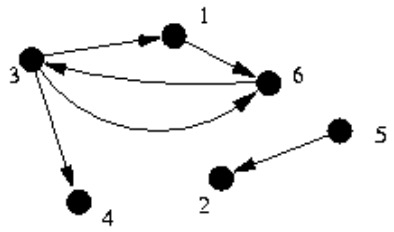
Hovoríme, že hrana (*u*,*v*) susedí (inciduje) s vrcholom *u* a vrcholom *v*. Napríklad, hrana (1,3), susedí s vrcholom 3. Stupeň(degree) vrcholu je počet hrán, ktoré s ním incidujú. Napríklad, vrchol 3 má stupeň 3, zatiaľ čo vrchol 4 má stupeň 1. Hovoríme, že vrchol *u* susedí s vrcholom *v*, ak existuje hrana incidentná k obom vrcholom (hrana medzi nimi). Napríklad, vrchol 2 susedí s vrcholom 5.

Počet vrcholov grafu značíme *N*. Hovoríme, že graf je riedky(sparse), keď počet jeho hrán *M* je malý v porovnaní s počtom všetkých možných hrán $\frac{N(N-1)}{2}$, inak je hustý(dense).

### 2.1.4. Orientované grafy

Grafy, ktoré sme zatiaľ popisovali sa nazývajú neorientované(undirected), pretože hrany idú „oboma smermi". Ak sa dalo prechádzať, napríklad, z vrcholu 1 do vrcholu 3, dalo sa aj naopak, z vrcholu 3 do vrcholu 1. Inými slovami, keď (1,3) patrilo do množiny hrán ***E***, tak aj (3,1) tam patrilo.

Niekedy, je k hranám priradený aj ich smer, potom hovoríme o orientovaných(directed) grafoch; hranám potom hovoríme orientované hrany, alebo šípy(arcs). Hrany v orientovaných grafoch sa znázorňujú šípkami, ktoré určujú ich smer.

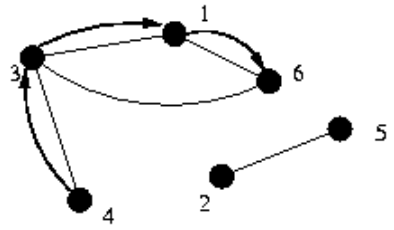



Odchádzajúci stupeň(out-degree) vrcholu je počet šípov, ktorý začínajú v danom vrchole. Prichádzajúci stupeň(in-degree) vrcholu je, naopak, počet šípov, ktoré končia v danom vrchole. Napríklad, vrchol 6 má prichádzajúci (vstupný) stupeň 2 a odchádzajúci (výstupný) 1.

### 2.1.5. Cesty

Cesta(path) z vrcholu *x* do vrcholu *y* je postupnosť vrcholov $(v_0,v_1,...,v_k)$ taká, že $v_0=x$ a $v_k=y$ a hrany $(v_0,v_1),(v_2,v_3),...$ patria do množiny hrán ***E***. Dĺžka tejto cesty je *k*.

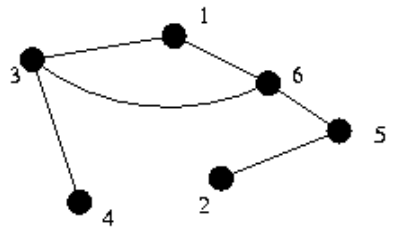
Napríklad v neorientovanom grafe v príklade (4,3,1,6) cesta:




Hovoríme, že táto cesta obsahuje vrcholy $v_0,v_1,...,v_k$ a hrany $(v_0,v_1),(v_2,v_3),...$ Vrchol *v* nazývame dosiahnuteľným z vrcholu *u*, ak existuje cesta z *u* do *v*. Cestu je jednoduchá, keď neobsahuje žiaden vrchol dva a viac krát. Cestu nazývame cyklom, ak je to cesta z nejakého vrcholu do toho istého vrcholu. Cyklus je jednoduchý, ak neobsahuje žiaden vrchol (okrem začiatku a konca cesty, ktorý je práve raz na začiatku cesty a práve raz na konci) dva a viac krát. Tieto definície platia aj pre orientované grafy (t.j. šípy $(v_0,v_1),(v_2,v_3),...$ patria do ***E***).

### 2.1.6. Dosiahnuteľnosť

Neorientovaný graf nazývame spojitým(alebo súvislým)(connected), keď existuje cesta z každého vrcholu do každého iného. Graf v príklade 2.1.1. nie je spojitý, lebo neexistuje, napríklad, cesta z vrcholu 2 do vrcholu 4. Avšak, ak by sme pridali hranu medzi vrcholy 5 a 6, vznikol by spojitý (súvislý) graf.

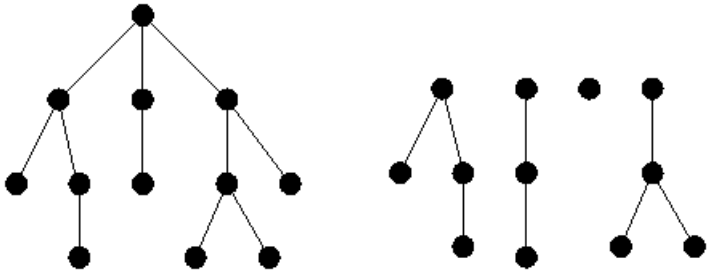



Komponent grafu je maximálna množina vrcholov s vlastnosťou, že každý jej vrchol je dosiahnuteľný z každého iného vrcholu v komponente. Pôvodný graf obsahuje práve dva komponenty: {1,3,4,6} a {2,5}. Všimnime si, že množina {1,3,4} nie je komponent, pretože nie je maximálna s danou vlastnosťou.

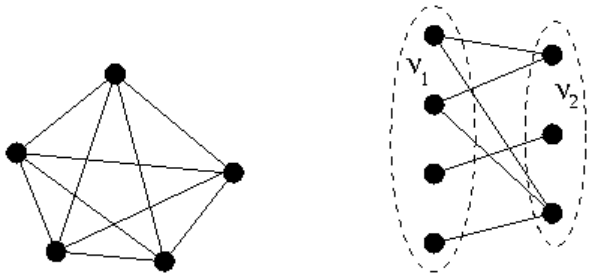
Orientovaný graf nazývame silne súvislým(strongly connected), ak existuje cesta medzi každými dvoma vrcholmi grafu. Silne súvislý komponent orientovaného grafu je vrchol *u* a množina všetkých vrcholov *v* takých, že existuje cesta z u do v aj z v do u.

### 2.1.7. Špeciálne typy grafov

Neorientovaný graf nazývame stromom(tree), keď je acyklický (neobsahuje cykly) a je súvislý.



Hovoríme, že strom je zakorenený(rooted), keď sme jasne vyznačili jeho najvyšší vrchol - koreň(root). Teda, každý vrchol *v* má práve jedného rodiča(parent) – vrchol *u* susediaci s *v*, ktorý je bližšie ku koreňu, a môže mať ľubovoľný počet detí(children), čo sú zvyšné vrcholy, ktoré s ním susedia. Strom na obrázku vľavo je zakorenený. Neorientovaný graf (na obrázku vpravo), ktorý neobsahuje cykly je les(forest). Orientovaný acyklický graf sa niekedy nazýva dag(directed-acyclic graph).




Hovoríme, že graf je úplný (kompletný), keď obsahuje hranu medzi každou dvojicou vrcholov. Úplný graf o *N* vrcholoch sa označuje $K_N$. Hovoríme, že graf je bipartitný, keď množina vrcholov môže byť rozdelená na dve množiny $V_1$, $V_2$ tak, že hrany vedú len z $V_1$ do $V_2$ alebo z $V_2$ do $V_1$.

### 2.1.8. Reprezentácia grafov v počítači

Spôsob uchovávania grafov v počítači závisí od konkrétneho problému. Existujú však aj, štandardné metódy ako k tomu pristupovať, medzi hlavné kritéria ich hodnotenia patrí pamäťová a časová efektívnosť operácií, ktoré od nich vyžadujeme[1]:

- ***zoznam hrán*** – hrany si pamätáme ako zoznam dvojíc vrcholov. Tento spôsob je jednoduchý na naprogramovanie a odladenie, pamäťovo efektívny, avšak väčšinou nepostačuje, pretože vyhľadanie hrán, ktoré začínajú v danom vrchole si vyžaduje prejsť vždy celý zoznam.
- ***matica susednosti*** – graf si pamätáme v matici *NxN*, pričom prvok *i,j* obsahuje 1, ak hrana $(i, j) \in E$, inak prvok *i,j* obsahuje 0. Pre neorientované grafy, je táto matica podľa hlavnej diagonály symetrická. Jednoducho sa to naprogramuje, ale pre riedke grafy to je pamäťovo neefektívne. Vyhľadávanie hrán, ktoré začínajú v danom vrchole je už rýchlejšie – vyžaduje si to prejsť len jeden riadok matice - ale v mnohých prípadoch to stále nepostačuje.
- ***zoznamy susednosti*** – ku každému vrcholu si pamätáme zoznam z neho odchádzajúcich hrán (pre neorientované grafy si teda pamätáme každú hranu dva razy). Je to pamäťovo aj časovo efektívne, ale vyžaduje si to precíznu implementáciu, ktorá nieje vždy možná.
- ***implicitne zadané grafy*** – niekedy si situácia nevyžaduje, aby sme si graf explicitne pamätali ako množinu vrcholov a hrán. Existujú situácie, keď si vieme danú množinu hrán generovať hneď na mieste[2].

Keď *N* je počet vrcholov grafu, *M* je počet hrán grafu a $d_{max}$ je najväčší zo stupňov vrcholov, tak nasledujúca tabuľka detailne sumarizuje efektivitu (počty potrebných operácií) jednotlivých prístupov.

| Efektívnosť | zoznamy hrán | matica susednosti | zoznamy susednosti |
|---|---|---|---|
| Pamäťové nároky | $2 \cdot M$ | $N^2$ | $2 \cdot M$ |
| Sú dva vrcholy susedné? | $M$ | 1 | $d_{max}$ |
| Susedné vrcholy daného vrcholu | $M$ | $N$ | $d_{max}$ |
| Pridaj hranu do grafu | 1 | 1 | 1 |
| Zmaž hranu z grafu | $M$ | 2 | $2 \cdot d_{max}$ |

[1] Uvedomme si, že každá situácia nie vždy vyžaduje optimálne riešenie. Menej optimálnejšia riešenie sú dobré v tom, že sú voči tým lepším síce pomalšie, ale prehľadnejšie, robustnejšie (ťažšie sa pri implementácií v nich spraví chyba) a stále dostatočne rýchle pre naše potreby.

[2] Predstavme si šachového koňa, ktorý skáče po šachovnici. Vrcholy grafu budú políčka šachovnice, hrana medzi dvoma políčkami bude vtedy, keď kôň môže skočiť z jedného políčka na druhé. Vždy, keď je kôň na nejakom políčku, vieme (bez zložitého počítania) presne určiť, kam môže skočiť v ďalšom kroku (vieme určiť hrany) a teda si hrany grafu nemusíme vopred vypočítavať a niekde pamätať.

## 2.3. Prehľadávanie do hĺbky (Depth-First Search)

S pojmom graf sa spája množstvo vlastností, ktoré sa tešia nášmu záujmu. Je daný graf súvislý? Ak nie, aké sú jeho súvislé komponenty? Obsahuje daný graf nejaký cyklus? Tieto a množstvo iných problémov môžeme jednoducho riešiť technikou nazývanou prehľadávanie do hĺbky(depth-first search), čo je prirodzený spôsob, ako „navštíviť“ každý vrchol a prejsť každú hranu grafu systematickým spôsobom. Variácie tohto prehľadávania, riešia množstvo ďalších úloh spojených s grafovými vlastnosťami a súvislosťou.

### 2.3.1. Spôsob prehľadávania

Pri prehľadávaní do hĺbky, ako to naznačuje už názov, prehľadávame vrcholy čo možno „hlbšie“ v grafe vždy, keď je to možné. Teda, najskôr prehľadávame hrany vrcholu *v*, ktorý bol objavený ako posledný, pokým vrchol *v* má neobjavené odchádzajúce hrany. Keď sme už všetky hrany vrcholu *v* objavili, v prehľadávaní sa vrátime do vrcholu, z ktorého sme objavili vrchol *v*. Tento proces pokračuje pokým neobjavíme všetky vrcholy, ktoré sú dosiahnuteľné z počiatočného vrcholu(zdroja). Ak v grafe zostanú nejaké neobjavené vrcholy, jeden z nich vyberieme ako nový zdroj a prehľadávanie zopakujeme s počiatkom vo vybranom vrchole. Celý tento postup opakujeme pokým nevyčerpáme všetky vrcholy grafu.

Uvádzame rekurzívnu implementáciu v jazyku Pascal a C:

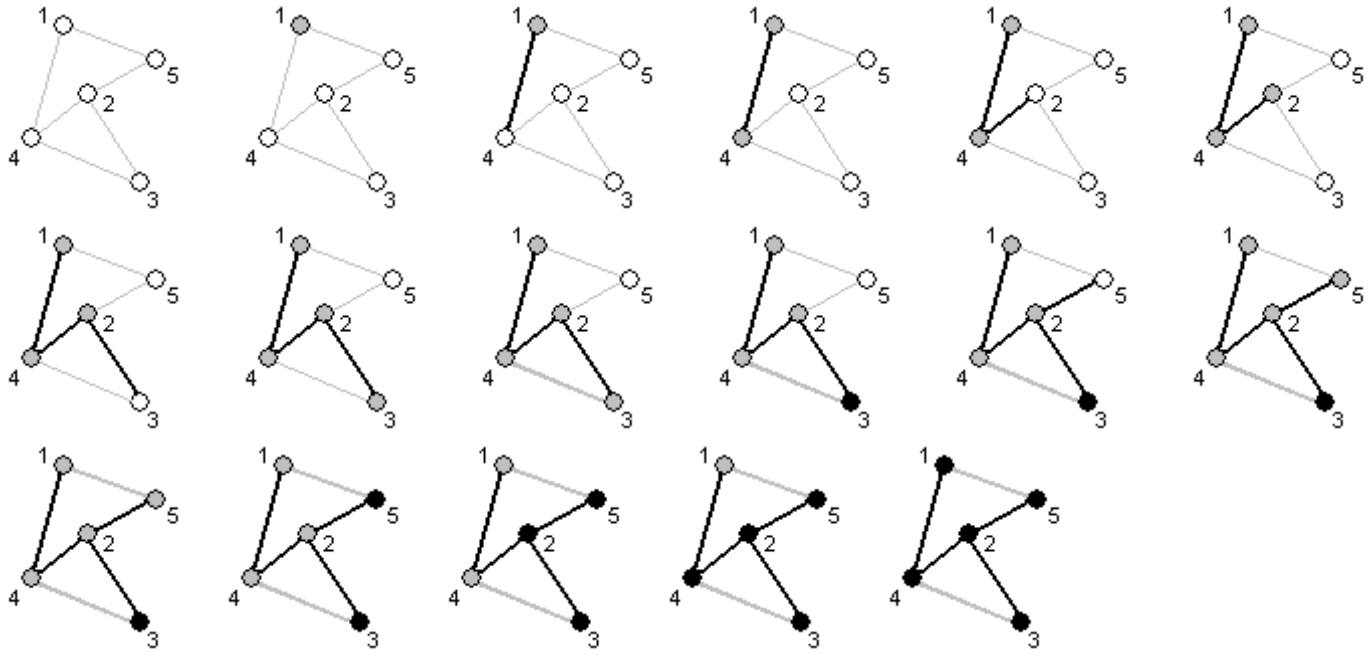
```
var graph:array[1..N,1..N] of integer;
    color[1..N] of integer;

{rekurzívne prehľadaj vrchol v}
procedure visit(v:integer);
var i:integer;
begin
  color[v]:=GRAY;
  {začíname prehľadávať}
  for i:=1 to n do
    if graph[v][i]<>0 then
      if color[i]=WHITE then
        visit(i);
  {koniec prehľadávania}
  color[v]:=BLACK;
end;

{prehľadávanie do hĺbky}
procedure dfs;
var i:integer;
begin
  {každý vrchol dostane bielu farbu}
  for i:=1 to n do
    color[i]:=WHITE;
  {začni z neofarbeného vrcholu}
  for i:=1 to n do
    if color[i]=WHITE then
      visit(i);
end;
```

```
int graph[N][N];
int color[N];

/* prehľadaj z vrcholu v */
void visit(int v)
{
  int i;
  /* začiatok */
  color[v]=GRAY;
  for(i=0;i<n;i++)
    if(graph[v][i]!=0)
      if(color[i]==WHITE)
        visit(i);
  /* koniec */
  color[v]=BLACK;
}

/* prehľadávanie do hĺbky */
void dfs(void)
{
  int i;
  /* každý vrchol zafarbíme na bielo */
  for(i=0;i<n;i++)
    color[i]=WHITE;
  /* začni z neofarbeného vrcholu */
  for(i=0;i<n;i++)
    if(color[i]==WHITE)
      visit(i);
}
```

Prehľadávanie nám, na začiatok, pridelí každému vrcholu bielu farbu, keď sa v priebehu prehľadávania „dostaneme“ do vrcholu zafarbíme ho na šedo a nakoniec, keď farbenie vo vrchole ukončíme, vrchol začiernime.

### 2.3.2. Detaily celého procesu

Proces prehľadávania, na príklade grafu, je zobrazený prehľadne nižšie (začíname vo vrchole 1):




Rekurzívne prehľadávanie začneme navštívením vrcholu 1. Objavením hrany (1,4) zistíme, že vrchol 4 ešte nie je objavený (má bielu farbu), v prehľadávaní teda pokračujeme navštívením vrcholu 4. Tam, objavením hrany (4,2), navštívime ďalší ešte neobjavený (biely) vrchol 2. Odtiaľ sa presunieme hranou (2,3) do vrcholu 3. Vo vrchole 3 prehľadávame jeho hrany, narazíme na hranu (3,4), avšak táto hrana vedie do vrcholu 4, ktorý je už objavený (nemá bielu farbu) a teda vrchol 4 už z vrcholu 3 nenavštevujeme. Prešli sme všetky hrany vrcholu 3. Vrchol 3 ofarbíme načierno a prehľadávanie v ňom končí (prehľadávanie pokračuje vo vrchole v ktorom sme vrchol 3 prvý raz objavili), vraciame sa do vrcholu 2. Objavením hrany (2,5) pokračujeme do vrcholu 5. Z vrcholu 5, objavením hrany (5,1), nepokračujeme ďalej do vrcholu 1, pretože už bol v minulosti navštívený. Skončili sme prehľadávať hrany vrcholu 5, ofarbíme ho načierno, vraciame sa do vrcholu 2. Vrchol 2 už nemá hrany, ktoré by sme ešte nenavštívili a teda ho ofarbíme načierno, vraciame sa do vrcholu 4. V vrchole 4 objavíme ďalšiu hranu (4,3) – túto hranu sme už objavili z vrcholu 3, ale ešte nie z vrcholu 4. Prehľadávanie ale vo vrchole 3 nepokračuje, pretože vrchol 3 je už zafarbený. Vo vrchole 4 teda už nie je čo prehľadávať, ofarbíme ho načierno a vraciame sa späť do vrcholu 1. Tu objavíme hranu (1,5), ktorá však nevedie do ešte neobjaveného vrcholu, vrchol 1 teda ofarbíme načierno a prehľadávanie končí.

### 2.3.3. Efektívnosť

Časová efektívnosť samotného prehľadávania prudko závisí na voľbe dátovej štruktúry, v ktorej si graf pamätáme. Pamäťová zložitosť je vždy lineárna O(N) od počtu vrcholov grafu $N$, aj keď v rekurzívnej verzií sa pri veľkých grafoch dosť „namáha“ stack volaní funkcií, existuje ale aj priamočiari prepis do nerekurzívnej formy. Časovú zložitosť odvodíme ako $NxD$, kde $D$ je čas, za aký sme schopný zistiť všetkých susedov pre daný vrchol. Pre graf v matici $NxN$: $D=N$ (celé prehľadávanie $O(N^2)$) pre graf zadaný ako zoznamy susedov $D=deg_i$, (celé prehľadávanie $O(N+M)$) pre graf zadaný ako zoznam $M$ hrán je $D=M$ (celé prehľadávanie $O(N.M)$).

## 2.4. Dátová štruktúra pre disjunktné množiny (Disjoint-set data structure)

Niekedy zoskupujeme abstraktné objekty do množiny disjunktných množín. Od takejto dátovej štruktúry vyžadujeme dve základné operácie:

- do ktorej množiny patrí vybraný prvok
- zjednotiť dve rôzne množiny

### 2.4.1. Grafová analógia

Dátová štruktúra pre disjunktné množiny si udržiava množinu $S = \{S_1,S_2,...,S_k\}$ disjunktných dynamických množín. Každá množina $S_i$ je jednoznačne identifikovateľná reprezentantom, čo je prvok $r_i \in S_i$. Zväčša nezáleží, ktorý prvok množiny vyberieme, dôležité je, keď sa spýtame na reprezentanta skupiny dva razy bez toho, aby sme štruktúru nejako pozmenili, aby bol rovnaký.

Grafy sa celkom prirodzene podobajú na množiny objektov. Vrcholy korešpondujú s objektmi a hrany znamenajú „byť v rovnakej množine“. Každý súvislý komponent je potom charakterizovaný inou disjunktnou množinou. Pre množiny nás zaujíma základná otázka „je x v rovnakej množine ako y?“, pre grafy sa potom prirodzene pýtame „je vrchol x spojený s vrcholom y?“.

### 2.4.2. Dynamická štruktúra

Daná množina hrán, jednoducho môžeme graf ofarbiť farbami tak, že vrcholy v rovnakom komponente budú mať rovnakú farbu. Otázky, či sú dva vrcholy v jednom a tom istom komponente, zodpovieme v konštantnom čase. My sa tu ale budeme zaoberať tzv. dynamickými štruktúrami, ktoré nám umožňujú pridávania hrán mať ľubovoľne prepletené s otázkami.

Našou úlohou teda je teda navrhnúť nasledujúce operácie:

- MAKE-SET(x) – vytvorí novú množinu $S_x$ s jediným prvkom $(S_x = \{x\})$
- UNION(x,y) – zjednotí množinu $S_x$ $(x \in S_x)$ s množinou $S_y$ $(y \in S_y)$
- FIND-SET(x) – vráti jednoznačného reprezentanta množiny $S_x$ $(x \in S_x)$

Neuberiem sa cestou zoznamov hrán k vrcholom, ale viac zameriame sa na vnútornú štruktúru pozitívne orientovanú k operáciám UNION a FIND-SET. Vnútorne si množinu $S$ budeme reprezentovať *lesom zakorenených stromov*. Stromy budú množiny a ich korene budú ich reprezentantmi. Potrebujeme byť schopný zistiť, či dva prvky patria do toho istého stromu, a kombinovať dva stromy do jedného, zistíme, že tieto operácie sa dajú implementovať veľmi efektívne.

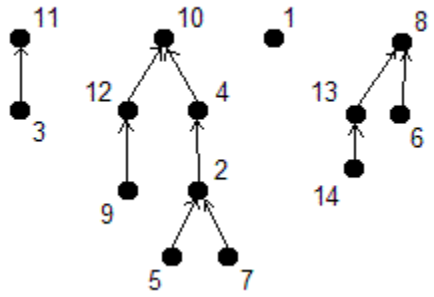
### 2.4.3. Ako to funguje?

Každý prvok bude ukazovať na svojho rodiča v strome, pričom rodič nebude ukazovať nikam, namiesto toho si bude pamätať počet jeho detí (počet prvkov daného stromu). Na takúto štruktúru nám v počítači bude postačovať jedno čísla $V_i$ na vrchol, pričom čísla vrcholov sú nezáporné čísla, takže, keď $V_i$ bude záporné, značí opačné číslo k počtu vrcholov stromu, inak to značí poradové číslo rodiča. Nasledujú implementácie požadovaných funkcií v jazyku C a v jazyku Pascal:

```
dad:array[1..N] of integer;

procedure make_set(x:integer);
begin
      dad[x]:=-1;
end;

function find_set(x:integer):integer;
begin
      while dad[x]>0 do
            x:=dad[x];
      find_set:=x;
end;

procedure union(x,y:integer);
var sx,sy:integer;
begin
      sx:=find_set(x);sy:=find_set(y);
      if sx<>sy then
      begin
            dad[sx]:=dad[sx]+dad[sy];
            dad[sy]:=sx;
      end;
end;
```

```
int dad[N];

void make_set(int x)
{
      dad[x]=-1;  /* len jeden člen */
}

int find_set(int x)
{
      /* záporné číslo -> koreň */
      while(dad[x]>=0)
            x=dad[x];
      return x;
}
void union(int x,int y)
{
      int sx=find_set(x),sy=find_set(y);
      if(sx!=sy)
      {
            dad[sx]+=dad[sy];
            dad[sy]=sx;
      }
}
```

Pre názorné pochopenie uvádzame príklad takéhoto stromu,:




Pole `dad[]` je nasledujúce:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| dad[i] | ***-1*** | ***4*** | ***11*** | ***10*** | ***2*** | ***8*** | ***2*** | ***-4*** | ***12*** | ***-7*** | ***-2*** | ***10*** | ***8*** | ***13*** |

### 2.4.4. Degenerované prípady

V praktickom svete, nám môžu nastať určité nepredvídané problémy. Uvažujme napríklad les na obrázku vyššie. Prvky 5 a 7 sú celkom hlboko v strome, pričom pri väčších stromoch by sa niektoré prvky mohli dostať poriadne hlboko a počet operácií, ktoré by sme museli vykonať pri operácií FIND-SET by úmerne rástol s ich hĺbkou. Toto by našu štruktúru spravilo ťažkopádnou a nevhodnou na rýchle otázky.

Existujú viaceré postupy na zrýchlenie, ktoré vo vzájomnej kombinácií fungujú ešte rýchlejšie. Jeden prirodzený spôsob, je spraviť to „správne“ pri operácií UNION, lepšie ako jednoducho zavesiť druhý strom pod prvý – vrcholom druhého stromu sa zvýši dĺžka cesty ku koreňu o jedna. Na minimalizovanie

vzdialenosť ku koreňu pre väčšiu časť vrcholov, je rozumné vybrať za koreň nového stromu koreň toho stromu, ktorý má viac potomkov. Táto myšlienka sa volá vyvažovanie váhou(weight balancing). Jednoducho sa implementuje tak, že, pamätajúc počet prvkov v koreni stromu, sa vždy pri operácií UNION rozhodneme spraviť koreňom vrchol s väčším počtom potomkov.

V ideálnom prípade by sme chceli, aby každý člen množiny ukazoval na koreň (aby operácia FIND-SET trvala čo možno najmenej). Na dosiahnutie tohto stavu by vyžadovalo prejsť aspoň všetky vrcholy jedného zo stromov, ktorý zjednocujeme. Toto môže byť ale neporovnateľne viac, ako počet vrcholov, ktoré pri operácií FIND-SET normálne prejdeme cestou ku koreňu. K tomuto „ideálu“ sa však môžeme priblížiť, tak že nasmerujeme všetky vrcholy, ktoré prejdeme, na koreň! Toto, spočiatku, vyzerá ako drastický krok, ale nie je to nič zložité a táto metóda kompresie cesty(path compression) nám razantne vylepší priemernú časovú zložitosť. Implementujeme ju jednoducho: keď už v operácií nájdeme koreň, urobíme znovu rovnaký prechod, pričom, vrcholy, ktoré prejdeme nasmerujeme do koreňa.

Uvázdame implementácie popísaných postupov v jazyku Pascal a C:

```
dad:array[1..N] of integer;

procedure make_set(x:integer);
begin dad[x]:=-1; end;

function find_set(x:integer):integer;
var i,k:integer;
begin
  i:=x;
  while dad[i]>0 do
    i:=dad[i];
  {kompresia cesty}
  while dad[x]>0 do
  begin k:=dad[x];dad[x]:=i;x:=k; end;
  find_set:=i;
end;

procedure union(x,y:integer);
var sx,sy:integer;
begin
  sx:=find_set(x);sy:=find_set(y);
  if sx<>sy then
  begin
    {ak strom sx ma viac prvkov ako sy}
    if dad[sx]<dad[sy] then
    begin
      inc(dad[sx],dad[sy]); dad[sy]:=sx;
    end else
    begin
      inc(dad[sy],dad[sx]); dad[sx]:=sy;
    end;
  end;
end;
```

```
int dad[N];

void make_set(int x)
{ dad[x]=-1; }

int find_set(int x)
{
  int k,i=x;
  while(dad[i]>=0)
    i=dad[i];
  /* kompresia cesty */
  while(dad[x]>=0)
  { k=dad[x]; dad[x]=i; x=k; }
  return x;
}

void union(int x,int y)
{
  int sx=find_set(x),sy=find_set(y);
  if(sx!=sy)
  {
    /* vahové vyvažovanie */
    if(dad[sx]<dad[sy])
    {
      dad[sx]+=dad[sy]; dad[sy]=sx;
    }
    else
    {
      dad[sy]+=dad[sx];
      dad[sx]=sy; }
    }
  }
}
```

Na záver poznamenajme, že tieto vylepšenia nám v praktických podmienkach zaručujú, že sa na akúkoľvek z týchto operácií vždy použije veľmi malý čas ohraničený malou konštantou $\alpha$.

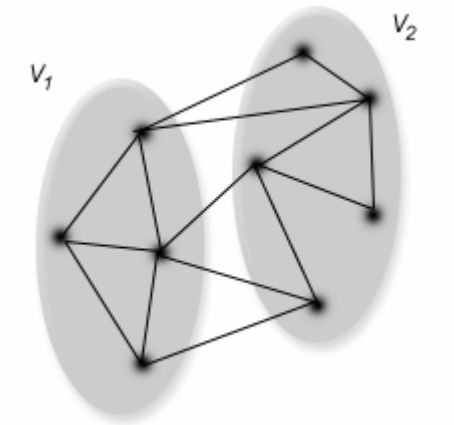
## 2.5. Najlacnejšia kostra (Minimum-weight spanning tree)

Kostra, v grafovej analógií, je podmnožina hrán pôvodného grafu taká, že zabezpečuje cesty medzi dvojicami vrcholov. V neorientovanom grafe G to je podgraf G - strom T taký, že existuje cesta medzi ľubovoľnými dvoma vrcholmi grafu G len po hranách stromu T. Najlacnejšia kostra je taká, spomedzi všetkých kostier G, ktorá má minimálny súčet ohodnotení hrán.

Problém nájdenia najlacnejšej kostry v danom grafe patrí medzi staršie úlohy na grafoch. Ako prvý sa ním zaoberal Borůvka (1926) inšpirovaný úlohou navrhovania optimálnej elektrovodnej siete. Borůvka dáva aj polynomiálny, ale dosť komplikovane formulovaný algoritmus. Jeho postup zlepšil Jarník (1930), avšak až po Kruskalovi (1956) sa problém stal dostatočne známy.

### 2.5.1. Základná vlastnosť

Kostry v neorientovaných grafoch majú nasledujúcu dôležitú vlastnosť:




Pre ľubovoľné rozdelenie vrcholov grafu G do dvoch disjunktných množín $V_1$ a $V_2$, minimálna kostra grafu obsahuje najkratšiu z hrán, ktoré vedú medzi $V_1$ a $V_2$.

Prečo to platí? Nech *e* je najkratšia hrana vedúca medzi $V_1$ a $V_2$, naviac predpokladajme, že *e* nie je v najlacnejšej kostre. Predstavme si graf, ktorý vznikne pridaním hrany *e* do našej najlacnejšej kostry. Tento graf obsahuje cyklus; v tomto cykle niektorá iná hrana spája množiny $V_1$ a $V_2$. Zmazaním tej hrany a pridaním hrany *e* do najlacnejšej kostry dostávame lacnejšiu kostru, čo je v spore s tým, že *e* nie je v najlacnejšej kostre.

Táto vlastnosť nám už umožňuje vytvoriť algoritmus konštruujúci pre daný graf G najlacnejšiu kostru G.

### 2.5.2. Kruskalov algoritmus

Algoritmus je priamo založený na uvedenej vlastnosti kostier. Kruskal navrhuje pred samotným hľadaním kostry usporiadať všetky hrany podľa ceny do rastúcej postupnosti $w(e_1) \le w(e_2) \le \ldots \le w(e_M)$. Potom postupne berieme hrany $e_1, e_2, \ldots, e_M$ a skúšame ich pridať (začíname s prázdnou kostrou) ku kostre T tak, aby v nej nevznikol cyklus. Ak po pridaní hrany $e_i$ vznikne v kostre cyklus, vynecháme ju a pokračujeme ďalšou hranou.

Všimnime si, že jediná prekážke je zisťovanie, či sa v kostre T po pridaní hrany $e_i = (u, v)$ nachádza cyklus, to je ale práve vtedy, keď už existuje cesta medzi vrcholmi *u* a *v*. Na rýchlu realizáciu otázok tohto typu použijeme štruktúru pre disjunktné množiny s kompresiou cesty.

Utriedenie si vyžaduje $O(M \log M)$ operácií, druhá časť, kde hrany pridávame do kostry, si vyžaduje v čase $O(M \log^* N)$, preto celý algoritmus pracuje v čase $O(M \log M)$. Vidíme, že najväčšie problémy nám spôsobuje triedenie hrán. Pokiaľ je hrán ale málo, Kruskalov algoritmus nám poskytuje vcelku dobré riešenie.

### 2.5.3. Primov algoritmus

Nasledujúci algoritmus sa prisudzuje Primovi(1957), aj keď jeho ideu už využíva Jarník(1930) a nezávisle ho znovu objavuje aj Dijkstra(1959) – algoritmus totiž funguje podobne ako Dijkstrov algoritmus pre hľadanie najkratšej cesty.

Primov algoritmus (opačne ako Kruskalov) konštruuje kostru postupným priberaním nových vrcholov do kostry. V každom kroku algoritmu, teda hrany vybrané do kostry T, tvoria jeden strom, pričom zvyšné vrcholy sú vzájomne izolované. Začíname (s prázdnou kostrou) z ľubovoľného vrcholu v grafe. V jednom kroku algoritmu, priberieme vrchol $v$ taký, že hrana $e = (u, v)$ je najlacnejšia zo všetkých hrán takých, že $u \in T$ a $v \in G - T$. Toto nám v jednom kroku rozšíri kostru po najlacnejšej hrane, ktorá susedí s nejakým vrcholom kostry T.

Nasleduje implementácia Primovho algoritmu:

```
var e:array[1..100,1..100]of longint;
    { t[i] - je vrchol i v kostre? }
    { dst[i] – vzdialenosť od kostry }
    t,dst:array[0..100]of longint;
    n:integer;

function prim:integer;
var i,j,k,val:longint;
begin
  for i:=0 to n do
  begin
    t[i]:=0; dst[i]:=9999999;
  end;
  { začíname z vrcholu 1 }
  dst[1]:=0;
  for k:=1 to n do
  begin
    i:=0; { nájdi najlacnejšiu hranu }
    for j:=1 to n do
      if (t[j]=0)and(dst[i]>dst[j]) then
        i:=j;
    { vrchol i pridaj do kostry }
    t[i]:=1;
    for j:=1 to n do
      if t[j]=0 then { susedia i }
        if dst[j]>e[i][j] then
          dst[j]:=e[i][j];
  end;
  { spočítaj dĺžku kostry }
  j:=0;
  for i:=1 to n do
    inc(j,dst[i]);
  prim:=j;
end;
```

```
int e[100][100]; /* hrany grafu */
/* t[i] - je vrchol i v kostre? */
/* dst[i] – vzdialenosť od kostry */
int t[100],dst[100];
int n;

int prim(void)
{
  int i,j,k,val;
  for(i=0;i<n;i++)
  {
    t[i]=0; dst[i]=9999999;
  }
  /* začíname z vrcholu 1 */
  dst[0]=0;
  for(k=0;k<n;k++)
  { val=9999999;
    /* nájdi najlacnejšiu hranu */
    for(j=0;j<n;j++)
      if(t[j]==0&&val>dst[j])
      { i=j;  val=dst[j]; }
    /* vrchol i pridaj do kostry */
    t[i]=1;
    for(j=0;j<n;j++)
      if(t[j]==0) /* susedia i */
        if(dst[j]>e[i][j])
          dst[j]=e[i][j];
  }
  /* spočítaj dĺžku kostry */
  j=0;
  for(i=0;i<n;i++)
    j+=dst[i];
  return j;
}
```

Takáto realizácia Primovho algoritmu si vyžaduje v každom kroku preskúmanie N vrcholov, takže vyžaduje $O(N^2)$ operácií. Existujú aj rýchlejšie implementácie, ktoré ale kvôli svojej zložitosť často strácajú praktický úžitok.

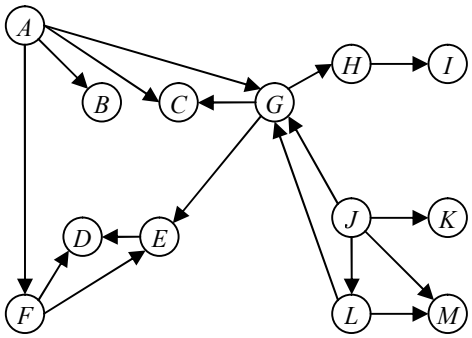
## 2.6. Topologické triedenie (Topological sorting)

Cyklické grafy sa objavujú v mnohých aplikáciách vyžadujúcich orientované grafy. V mnohých situáciách sa orientované grafy používajú na vyjadrenie závislostí činností ktoré chceme vykonať, v týchto prípadoch je existencia cyklus považuje za fatálny prípad, teda, vyžadujú sa orientované grafy bez orientovaných cyklov. Tieto grafy voláme orientované acyklické grafy(directed acyclic graphs) v skratke dagy.

Dag môže mať veľa cyklov, keď neberieme orientácie hrán v úvahu, avšak ich definujúca vlastnosť je jednoducho, nikdy by sme sa nemali dostať do cyklu prechádzaním hrán v smere ich orientácie. Dagy sú odlišné od všeobecných grafov; s časti sú stromom, s časti grafom. Celkom určite vieme využiť ich štruktúru pri ich spracovaní.

### 2.6.1. Topologické usporiadanie(topological ordering)

Základnom operáciou na dagoch je spracovanie vrcholov grafu v takom poradí, že žiaden vrchol nie je spracovaní skôr ako vrchol, ktorý na neho ukazuje.




Topologické usporiadanie vrcholov grafu *G* je lineárne usporiadanie všetkých jeho vrcholov také, že ak *G* obsahuje hranu (*u*,*v*), potom *u* za nachádza v tomto usporiadaní pred vrcholom *v*. Ak daný graf G nie je acyklický (dag), potom zjavne takéto usporiadanie neexistuje.

Pre graf na obrázku napríklad nasledovné usporiadanie spĺňa podmienky a teda je topologické pre vrcholy daného grafu:

J K L M A G H I F E D B C

Ak by sme nakreslili hrany medzi takto znázornenými vrcholmi, všetky by išli zľava doprava. Vo všeobecnosti usporiadanie vrcholov generované topologickým triedením nie je jediné možné. V našom príklade, napríklad usporiadanie A J G F K L E M B H C I D je jedno z ďalších možných.

### 2.6.2. Reverzné topologické usporiadanie(reverse topological ordering)

Niekedy je výhodné interpretovať hrany v grafe naopak: hrana (*x*,*y*) znamená, že vrchol *x* závisí na vrchole *y*. Napríklad, vrcholy grafu môžu reprezentovať výrazy v matematickej knižke, pričom hrana (*x*,*y*) znamená, že definícia výrazu *x* využíva definíciu výrazu *y*. V tomto prípade je dôležité nájsť usporiadanie definícií tak, aby bol každý výraz definovaný predtým ako sa niekde použije. Toto korešponduje s umiestnením vrcholov do radu tak, aby hrany medzi nimi šli len sprava doľava. Takéto reverzné topologické usporiadanie pre náš príklad je napríklad nasledovné:

D E F C B I H G A K M L J

Rozdiel tu nie je až taký kritický: reverzné topologické triedenie grafu je ekvivalentné topologickému triedeniu grafu s otočenými orientáciami hrán. Reverzné topologické triedenie grafu sa dá ale jednoducho získať prehľadávaním do hĺbky, tak, že vždy keď prehľadávanie v rekurzií „odchádza z vrcholu“, vypíšeme číslo vrcholu; nakoniec máme vypísané čísla vrcholov v reverznom (opačnom) topologickom triedení.

### 2.6.3. Realizácia v programovacom jazyku

Modifikované prehľadávanie do hĺbky, ktoré generuje reverzné topologické usporiadanie grafu uvádzame implementované v programovacích jazykoch Pascal a C.

```
var n,nto:integer;
    saw:array[1..100]of integer;
    torder:array[1..100]of integer;
    g:array[1..100,1..100]of integer;

procedure dfs_rts(v:integer);
var i:integer;
begin
  saw[v]:=1;
  { prehľadajme ďalšie vrcholy }
  for i:=1 to n do
    if g[v,i]=1 then
      if saw[i]=0 then
        dfs_rts(i);
  torder[nto]:=v;
  inc(nto);
end;

procedure tsort;
var i:integer;
begin
  nto:=1;
  for i:=1 to n do
    if saw[i]=0 then
      dfs_rts(i);
  { hľadáme topologické usp. }
  for i:=n downto 1 do
    writeln(torder[i]);
end;
```

```
int n,g[100][100]; /* graf */
int saw[100]; /* navštívili sme */
/* reverzné topologické usp. */
int nto,torder[100];

void dfs_rts(int v)
{
  int i;
  saw[v]=1;
  /* prehľadajme ďalšie vrcholy */
  for(i=0;i<n;i++)
    if(g[v][i])
      if(saw[i]==0)
        dfs_rts(i);
  /* pridaj do reverzného usp. */
  torder[nto++]=v;
}

void tsort(void)
{
  int i;
  nto=0;
  for(i=0;i<n;i++)
    if(saw[i]==0)
      dfs_rts(i);
  /* hľadáme topologické usp. */
  for(i=n-1;i>=0;i--)
    printf("%d\n",torder[i]);
}
```

Správnosť tohto postupu je zrejmá, pretože vrchol pridáme do reverzného topologického usporiadania vždy až vtedy, keď sme tam už pridali všetkých jeho priamych a nepriamych nasledovníkov.

Rovnako ako pri prehľadávaní do hĺbky, časová zložitosť nájdenia topologického usporiadanie vrcholov závisí od spôsobu zapamätania grafu. Topologické triedenie sme implementovali pre maticu *NxN* a preto je časová zložitosť rovnako $O(N^2)$.

### 2.6.4. Podmienka existenice

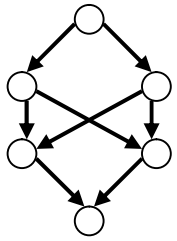
Topologické triedenie sa využíva hlavne na spracovanie vrcholov v poradí neporušujúcom závislosti vyjadrené hranami grafu *G*. Vieme teda, čo získame, ak vieme vrcholy takto utriediť. Čo však, ak sa takéto usporiadanie vrcholov nedá v grafe nájsť? Medzi dôležité vlastnosti grafu patrí teda aj neexistencia topologického utriedenia. V orientovanom grafe je neexistencia topologického triedenia ekvivalentná existencií cyklu. Plynie, **ak v grafe neexistuje cyklus, určite existuje topologické utriedenie jeho vrcholov**, rovnako aj opačne. Správnosť ekvivalencie vyplýva so správnosti prehľadávania do hĺbky.

## 2.7. Toky v sieťach (Network flow)

Ohodnotené orientované grafy(weighted directed graphs) sú vhodným modelom pre viaceré aplikácie zahrňujúce prevoz materiálov cez prepojené siete. Tok(flow) v sieti budeme chápať ako abstrakciu od nám známych fyzikálnych tokov (ropy, vody, elektriny, materiálu všeobecne) modelovanú v grafoch.

### 2.7.1. Úloha o maximálnom toku

Uvažujme sústavu rúr rôznych veľkostí, zložito spojených, s regulátormi množstva a smeru prietoku na križovatkách. Nech má táto sústava jediný zdroj – ropné pole – a jediný spotrebič – ropnú rafinériu – na ktorý sú všetky rúry koniec koncov napojené. Akým nastavením regulátorov na križovatkách docielime najvyšší prietok celej sústavy? Zložité vzťahy (platné pre prietoky na križovatkách) robia tento problém nie veľmi jednoduchým.



Vyššie popísanú situáciu si znázornime náčrtkom siete rúr. Rúry majú všetky rovnakú kapacitu a ropa môže tiecť len smerom nadol. Regulátory na križovatkách rozdeľujú prichádzajúci tok do odchádzajúcich rúr. Nezávisle na nastaveniach regulátorov, sústava je vyvážená vtedy, keď množstvo ktoré priteká hore je rovné množstvu, ktoré odteká dole, a keď množstvo pritekajúce do každej križovatky sa rovná množstvu, ktoré z nej odteká.

Daný je teda orientovaný graf $G$, v ktorom sú vyznačené dva vrcholy: vstup $s$ a výstup $t$: ostatné vrcholy sú tzv. *prechodové*. Ďalej, každá hrana má priradené nezáporné číslo $c_{ij}$ – kapacitu. Každej hrane naviac priradíme číslo $f_{ij}$. Ak priradené čísla spĺňajú nasledovné podmienky

- Kapacitné ohraničenia: $0 \le f_{ij} \le c_{ij}, \qquad ij \in E(G)$
- Podmienka kontinuity: $\left(\sum_{j \in V^-(i)} f_{ji}\right) - \left(\sum_{j \in V^+(i)} f_{ij}\right) = 0,\ i \in V(G) - \{s,t\}$

Potom sa priradenie $f$ nazýva $s$–$t$ tok v $G$, pričom čísla $f_{ij}$ je množstvo hmoty, ktoré tečie hranou $ij$. Podmienky kontinuity vyjadrujú jednoduchý fakt, že to, čo do vrcholu priteká, z neho aj odteká. Všimnime si že uvedený grafový model plne zodpovedá našej pôvodnej predstave o sústave rúr. Číslo

$$F(f) = \left(\sum_{j \in V^-(t)} f_{jt}\right) - \left(\sum_{j \in V^+(t)} f_{tj}\right) = -\left[\left(\sum_{j \in V^-(s)} f_{js}\right) - \left(\sum_{j \in V^+(s)} f_{sj}\right)\right]$$

nazývame veľkosť (alebo hodnota) $s$–$t$ toku. $F(f)$ je vlastne „čisté“ množstvo, ktoré pretečie do $t$. Vzhľadom na podmienky kontinuity, je $F(f)$ aj „čistý“ odtok z $s$.

Úloha o maximálnom toku, pre daný orientovaný graf $G$ s danými kapacitami hrán a dvoma význačnými vrcholmi $s$ a $t$, spočíva v nájdení tokov $f$ na jednotlivých hranách tak, aby $f$ bol $s$–$t$ tok s maximálnou veľkosťou. Taký tok sa nazýva *maximálny*(*maxflow*).

### 2.7.2. Ford-Fulkersonov algoritmus

Základný prístup k tomuto problému bol navrhnutý v roku 1956. Páni L.R.Ford a D.R. Fulkerson prišli s postupom, ktorým sa dá zvýšiť ľubovoľný tok okrem maximálneho. Začínajúc s nulovým tokom ($f_{ij} = 0$), aplikujeme Ford-Fulkersonov postup na zvýšenie toku, pokým sa už nedá použiť – našli sme maximálny tok.

Zrejme môžeme každý tok zvýšiť aspoň o najmenšie množstvo nepoužitej kapacity na hranách na nejakej ceste z ***s*** do ***t*** tým, že zvýšime tok na hranách tej cesty o toto množstvo. Toto môžeme aplikovať, pokým sa na každej ***s–t*** ceste nenachádza aspoň jedna *nasýtená* hrana ($f_{ij} = c_{ij}$). Existuje ale ďalší spôsob, ako zvýšiť tok! Uvažujme, ignorujúc orientácie hrán, ľubovoľné ***s–t*** cesty – môžu obsahovať aj hrany opačné k daným. Zvýšenie toku na takejto ceste potom urobíme zvýšením toku na hranách so správnou orientáciou (forward edges) a znížením toku na spätných hranách(backward edges).

Uvedené pozorovanie teraz zovšeobecníme. Nech ***P*** je nejaká ***s–t*** polocesta (cesta, v ktorej ignorujeme orientácie hrán), nech $P^+$ je množina hrán rovnakej orientácie a $P^-$ množina hrán nesúhlasnej orientácie polocesty ***P***. Čísla

$$r_{ij} = \begin{cases} c_{ij} - f_{ij}, & ij \in P^+ \\ f_{ij}, & ij \in P^- \end{cases}$$

nazveme rezervy. Minimálna z rezerv na ***P*** sa nazýva rezerva polocesty ***P***. Ak je rezerva nejakej ***s–t*** polocesty kladná, nazývame ju aj *zväčšujúca* polocesta a potom nové priradenie $f'$

$$f'_{ij} = \begin{cases} f_{ij} + \delta, & ij \in P^+ \\ f_{ij} - \delta, & ij \in P^- \\ f_{ij}, & ij \notin P \end{cases}$$

je opäť tokom v ***G*** a $F(f') = F(f) + \delta$, kde $\delta$ je rezerva polocesty ***P***.

Naviac, keď neexistuje zväčšujúca polocesta – každá ***s–t*** polocesta obsahuje nasýtenú hranu súhlasnej orientácie alebo prázdnu hranu nesúhlasnej orientácie – tak nájdení tok je maximálny. Uvažujme všetky ***s–t*** polocesty a všimnime si na každej prvú nasýtenú súhlasnú hranu alebo prvú prázdnu nesúhlasnú hranu. Množina týchto hrán rozdeľuje daný graf na dve časti. Hrany ktoré vedú naprieč týmito množinami nazývame hrany rezu. Prietok na takomto reze je potom súčet prietokov na súhlasných hranách rezu mínus súčet prietokov na nesúhlasných hranách rezu. Ak sa *tok na nejakom reze rovná aktuálnemu toku* vieme, že daný tok je maximálny a že daný rez je minimálny (každý iný rez má aspoň taký prietok). Toto sa nazýva *veta o maximálnom toku a minimovom reze* (*maxflow-mincut theorem*) – tok nemôže byť väčší (pretože rez by musel byť tiež väčší) a žiaden menší rez neexistuje (pretože tok by musel byť menší tiež).

### 2.7.3. Realizácie Ford-Fulkersonovho algoritmu

Ford-Fulkersonov algoritmus popísaný vyššie nie je algoritmom v pravom slova zmysle. Nedáva nám totiž návod, ako konkrétne zväčšujúce polocesty hľadať. Ak zvolíme nejakú konkrétnu stratégiu ich hľadania dosiahneme rozdielne algoritmy na nájdenie maximálneho toku.

- *Algoritmus najkratších polociest*: Ak budeme zväčšovať tok vždy po najkratšej zväčšujúcej poloceste, vystačíme s $O(MN)$ zväčšeniami toku.
- *Algoritmus najširších polociest*: Ak sa každé zväčšenie uskutočňuje po najširšej zväčšujúcej ***s–t*** poloceste, vystačíme pri celočíselných kapacitách s $O(M \cdot \ln F(f))$ zväčšeniami toku.

### 2.7.4. Implementácia Ford-Fulkersonovho algoritmu

Uvedená implementácia nájde zlepšujúcu polocestu prehľadávaním do hĺbky a zvýši doteraz nájdený tok pozdĺž nej. Tok zvyšuje len po 1 jednotke, takže je nevhodná pre veľké toky, dá sa však jednoducho modifikovať tak, aby zvýšila tok o celú rezervu polocesty nájdenej.

```
var c,f:array[1..max,1..max]of integer;
    saw,dead:array[1..max]of integer;
    n,s,t:integer;

function push(v:integer):integer;
var i:integer;
begin
  if v=t then
   begin push:=1; exit; end;
  if (dead[v]=1) or (saw[v]=1) then
   begin push:=0; exit; end;
  saw[v]:=1;
  {predné hrany}
  for i:=1 to N do
    if c[v,i]>f[v,i] then
      begin
        if push(i)=1 then
         begin
          inc(f[v,i]);
          saw[v]:=0;
          push:=1;
          exit;
         end;
      end;
  {spätné hrany}
  for i:=1 to N do
    if f[i,v]>0 then
      begin
        if push(i)=1 then
         begin
          dec(f[i,v]);
          saw[v]:=0;
          push:=1;
          exit;
         end;
      end;
  {neúspešné zvýšenie cez vrchol v}
  saw[v]:=0;
  dead[v]:=1;
  push:=0;
end;
```

```
int c[N][N],f[N][N];
int saw[N],dead[N];
int n,s,t;

int push(int v)
{
  int i;
  {sme vo vrchole t}
  if(v==t)
    return 1;
  if(dead[v]||saw[v])
    return 0;
  {ešte sme neboli vo v}
  saw[v]=1;
  {predné hrany}
  for(i=0;i<n;i++)
    if(c[v][i]>f[v][i])
    {
      if(push(i))
      {
         f[v][i]++;
        saw[v]=0;
        return 1;
      }
    }
  {spätné hrany}
  for(i=0;i<n;i++)
    if(f[i][v])
    {
      if(push(i))
      {
        f[i][v]--;
        saw[v]=0;
        return 1;
      }
    }
  {neúspešné zvýšenie cez v}
  saw[v]=0;
  dead[v]=1;
  return 0;
}
```

Funkcia `push(v)` vráti, či sa dá cez vrchol ***v*** pretlačiť jednotka toku, teda, či existuje zväčšujúca ***v–t*** polocesta. Jednoduchým spôsobom sa dá funkcia pozmeniť tak, aby vracala množstvo toku o ktorý sa dá zvýšiť ***v–t*** tok a teda by sme výsledný maximálny tok našli rýchlejšie (nezväčšovali by sme ho pojednom).