Pokročilé použitia prehľadávania do hĺbky

Sylabus, resp. čo treba vedieť na skúške

- *algoritmus prehľadávania do hĺbky*
- *algoritmus na hľadanie artikulácii a mostov*
- *algoritmus na hľadanie silno súvislých komponentov*
- *algoritmus riešiaci problém 2-SAT*

Na skúške by ste mali poznať spomenuté algoritmy, vedieť ako fungujú, akú majú časovú zložitosť a vedieť dokázať ich správnosť.

Algoritmus prehľadávania do hĺbky patrí k jedným zo základných grafových algoritmov. Jeho užitočnosť však často zatieňuje prehľadávanie do šírky, ktoré vie na rozdiel od prehľadávania do hĺbky hľadať najkratšie cesty v neohodnotených grafoch. Prehľadávanie do hĺbky má však okrem jednoduchej rekurzívnej implementácie aj mnoho ďalších užitočných vlastností, ktoré sa prejavia pri riešení zložitejších problémov. V tejto prednáške si preto niekoľko takýchto problémov predstavíme a ukážeme si ako využiť prehľadávanie do hĺbky na ich riešenie.

Prehľadávanie do hĺbky

Skôr ako sa pustíme do riešenia zložitejších úloh, zopakujme si ako funguje algoritmus prehľadávania do hĺbky, tiež známy pod skratkou DFS (z anglického `Depth First Search`).

Ako naznačuje algoritmus, budeme sa snažiť postupne prehľadávať graf, začínajúc nejakým konkrétnym vrcholom. Na toto prehľadávanie použijeme nasledujúcu rekurzívnu myšlienku: Ak som vo vrchole v, spustím rekurízvne prehľadávanie zo všetkých susedov vrchola v, v ktorých som predtým nebol.

Je jasné, že takéto prehľadávanie skutočne navštívi každý vrchol zadaného súvislého grafu. Cesty, ktoré nájde však nemusia byť najkratšie možné, keďže postupuje rekurzívne. Predstavme si trojuholník tvorený troma vrcholmi 1, 2 a 3. Nech prehľadávanie začne vo vrchole 1. Toto prehľadávanie nájde ešte nenavštíveného suseda vrchola 1, napríklad vrchol 3. Prehľadávanie preto pokračuje z tohto vrcholu. A keďže vrchol 1 sme už navštívili, jediný nenavštívený sused vrchola 3 je vrchol 2. Z neho preto spustíme prehľadávanie, to však hneď skončí, lebo vrchol 2 už nemá nanavštíveného suseda. Algoritmus sa vynorí z rekurzie a pokračuje so spracovaním vrchola 3, tam však už tiež nič nezostalo, preto sa vynorí do rekurzie pre vrchol 1. Z neho však tiež nevedie žiadna hrana do nenavštíveného vrchola a algoritmus skončí. Napriek tomu, že sme začínali vo vrchole 1, ktorý je s vrcholom 2 susedný, dostali sme sa do neho cestou dĺžky 2.

Nasleduje pomerne jednoduchá implementácia algoritmus DFS.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
using namespace std;

vector<vector<int> > G; //zoznam susedov reprezentujici graf
vector<bool> T;         //oznacenie navstivenia
int n,m;                //pocet vrcholov a hran

void dfs(int v) {
    T[v]=true;          //oznacim navstivenie
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];
        if(T[w]) continue;
        dfs(w);         //rekurzive sa zavolam na nenavstiveny vrchol
    }
}

int main() {
    T.resize(n,false);
    //nacitaj graf G
    dfs(0);             //pusti prehladavanie z vrcholu 0
}
```

Oproti prehľadávaniu do šírky (BFS z anglického `Breadth First Search`) je takáto implementácia trošku jednoduchšia, keďže nevyžaduje žiadnu pomocnú dátovú štruktúru a takisto v podstate nezávisí na tom, v ktorom momente označíme daný vrchol za navštívený. Ak ste zžitý s rekurziou, takéto riešenie by pre vás malo byť intuitívne. Samozrejme, problém môže nastať ak máme napríklad obmedzenú veľkosť stacku. Časová zložitosť takéhoto algoritmu je $O(n+m)$ (n označuje počet vrcholov a m počet hrán).

Algoritmus DFS sa často používa hlavne pri stromových grafoch. Pokúsme sa preto pre zakorenený strom nájsť jeho hĺbku. No a ako vieme vypočítať hĺbku stromu s koreňom v? Predsa ako maximálna hĺbka zo všetkých podstromov vrchola v plus 1 (špeciálne ošetrujúc prípad, keď v nemá žiaden podstrom). No ale niečo takéto vieme vyjadriť spraviť iba s malou obmenou DFS algoritmu.

Ak predtým `dfs(v)` funkcia prehľadávala graf z vrcholu v, teraz bude chcieť zistiť hĺbku podstromu zakoreneného za vrchol v. A opäť bude musieť prejsť všetkými jeho susedmi. Do algoritmu sa preto pridá návratová hodnota funkcie `dfs(v)`, jedna premenná a jedno maximum naviac. S minimálnou zmenou algoritmus DFS dostávame riešenie nášho problému.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
using namespace std;

vector<vector<int> > G; //zoznam susedov reprezentujici graf
vector<bool> T;         //oznacenie navstivenia
int n,m;                //pocet vrcholov a hran

int dfs(int v) {
    T[v]=true;          //oznacim navstivenie
    int hlbka=-1;
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];
        if(T[w]) continue;
        hlbka=max(hlbka,dfs(w));         //zisti hlbku najhlbsieho podstromu
    }
    return hlbka+1;     //vrati hlbku podstromu pod vrcholom v
}

int main() {
    T.resize(n,false);
    //nacitaj graf G
    dfs(0);             //zisti hlbku celeho stromu s korenom vo vrchole 0
}
```

Pri zakorenených stromoch ešte zostaneme. Ďalšou úlohou bude pre dvojicu vrcholov v a w zistiť, či je vrchol v predkom vrchola w. Vrchol v je pritom predkom vrchola w ak sa nachádza na ceste z vrchola w do koreňa grafu, ktorý si označíme k.

Keďže cesta z vrchola w do koreňa k je jednoznačne určená, jedna možnosť je postupne prejsť túto cestu a skontrolovať, či sa na nej nachádzal vrchol v. Najjednoduchší spôsob ako prejsť túto cestu je začať vo vrchole w a posúvať sa vždy do otca aktuálneho vrchola. Jediný problém je, ako nájsť otca vrcholu w.

Na to nám však pomôže jedno DFS z koreňa k. Keď totiž vo funkcii `dfs(v')` voláme funkciu `dfs(w')`, tak vieme, že vrchol v' je otcom vrchola w'. Na začiatku preto spustíme takéto prehľadávanie a pre každý vrchol si zapamätáme, kto bol jeho otcom. S touto informáciou už vieme prejsť cestu z vrchola w do koreňa v čase závislom od hĺbky stromu. To je však v najhoršom prípade $O(n)$. A ak nedostaneme len jednu dvojicu (v,w), ale takýchto dvojíc bude pre ten istý strom viac, spomenutý algoritmus bude príliš pomalý.

Potrebujeme preto nájsť nejaké lepšie riešenie. Pozrime sa na to, čo nám prezradí DFS spustené z koreňa stromu k, keď budeme predpokladať, že vrchol v je naozaj predkom vrchola w. Uvedomme si, že DFS navštívi vrchol v **skôr** ako vrchol w. w sa totiž nachádza v podstrome s koreňom v a tento podstrom začíname prehľadávať od jeho koreňa. Druhé dôležité pozorovanie je, že keď sa vynárame z rekurzie, tak najskôr sa vynoríme z rekurzie pre vrchol w a až potom z rekurzie pre vrchol v. Keď sa vynárame z rekurzie, museli sme spracovať všetky vrcholy, ktoré sa nachádzali nižšie. Ak sa teda vynárame z v, museli sme sa už vynoriť z w, ktoré leží pod v.

Na overenie, či je vrchol v predkom vrchola w preto potrebujeme zistiť, či DFS spustené z vrchola k:

- prvýkrát navštívi vrchol v skôr ako vrchol w
- vynorí sa z rekurzie pre vrchol w skôr ako pre vrchol v

Pozorný čitateľ ľahko nahliadne, že ak je porušená aspoň jedna z týchto dvoch podmienok, vrchol v nebude predkom w (leží pod ním, alebo v úplne inom podstrome).

Hodnoty (časy), ktoré hľadáme sú však určené **jedným** DFS prehľadávaním z koreňa k, ktorý sa nemení a ak sme tieto hodnoty schopný vypočítať, obe podmienky vieme ošetriť v konštantnom čase. Takýto algoritmus by mal preto zložitosť $O(n)$ na predpočítanie, ale iba $O(1)$ na každú opýtanú dvojicu.

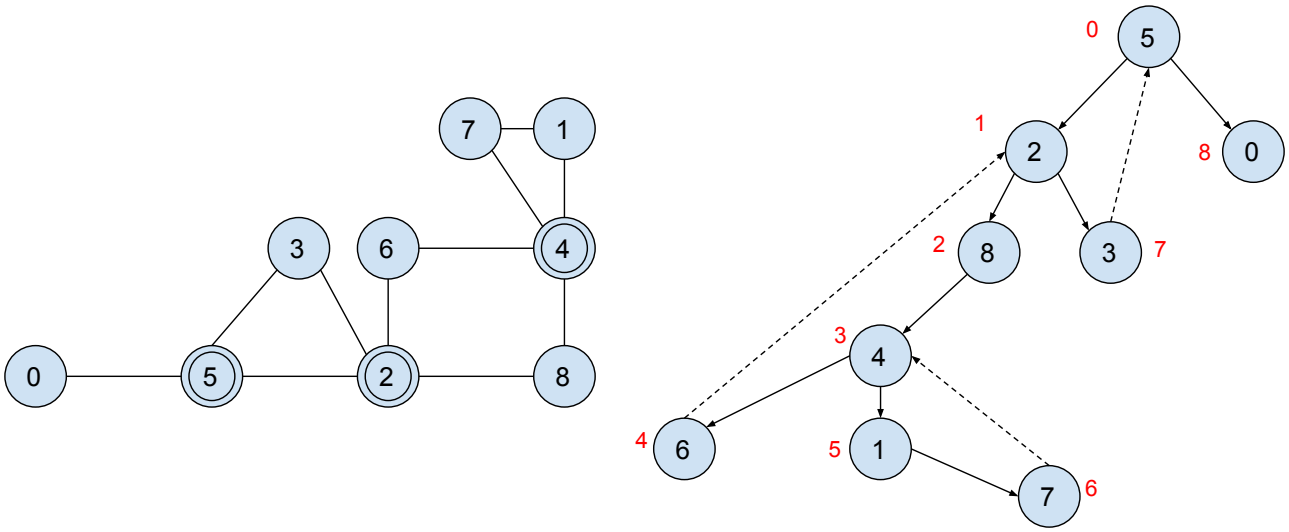
No a vypočítať tieto hodnoty nie je ťažké. Vieme predsa kedy prvýkrát objavíme vrchol x (keď tento vrchol označíme za navštívený) a aj kedy ho naposledy opúšťame (keď spracujeme všetkých jeho susedov). Stačí nám preto vytvoriť si jednu globálnu premennú `cas`, ktorá sa bude zvyšovať počas behu DFS algoritmu a do vhodných polí si zaznačíme čas objavenia (pole `Z[]`) a čas vynorenia (pole `K[]`).

Listing programu (C++)

```
#include <cstdio>
#include <vector>
using namespace std;

vector<vector<int> > G; //zoznam susedov reprezentujici graf
vector<bool> T;         //oznacenie navstivenia
vector<int> Z,K;        //pole objavenia a vynorenia
int n,m;                //pocet vrcholov a hran
int cas=0;              //globalna premenna casu

void dfs(int v) {
    T[v]=true;
    Z[v]=cas++;         //prve navstivenie vrchola v
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];
        if(T[w]) continue;
        dfs(w);         //rekurzive sa zavolam na nenavstiveny vrchol
    }
    K[v]=cas++;         //rekurzivne vynorenie z vrchola v
}

bool predok(int v, int w) { //je vrchol v predok vrchola w?
    if(Z[v]<=Z[w] && K[v]>=K[w]) return true; //overime obe podmienky
    return false;
}

int main() {
    T.resize(n,false);
    Z.resize(n); K.resize(n);
    //nacitaj graf G
    dfs(0);             //pusti prehladavanie z vrcholu 0
}
```

Artikulácie a mosty

Nech G je neorientovaný súvislý graf. Vrchol v nazveme **artikulácia**, ak sa graf $G - \{v\}$ skladá z viac ako jedného komponentu.



Obr. 1: Obrázok grafu a DFS stromu, ktorý vznikne ak na ňom spustíme prehľadávanie z vrcholu 5.

Artikulácie sú teda vrcholy, ktorých odstránenie z grafu spôsobí rozpadnutie tohto grafu na viacero komponentov. Na obrázku 1 sú to vrcholy 2, 4 a 5. Otázkou je, ako takého vrcholy nájsť. Samozrejme, jedna možnosť je vyskúšať odstrániť každý vrchol a potom niektorým z prehľadávaní zistiť, či vytvorený graf obsahuje viacero komponentov. Takéto riešenie má však časovú zložitosť $O(n(n+m))$, čo je príliš pomalé.

Opäť sa teda pozrieme, ako bude vyzerať prehľadávanie DFS algoritmom. Uvedomme si, že počas prehľadávania narazíme iba na dva typy hrán:

- **stromová** – hrana patriaca do DFS stromu, ktorou objavujeme nový vrchol
- **spätná** – hrana vedúca do už objaveného vrchola vyššie v strome

Na obrázku 1 si môžete prezrieť zakreslenie DFS stromu, ktorý vytvorí algoritmus DFS začínajúci vo vrchole 5. Spätné hrany sú označené prerušovanou šipkou. Začať prehľadávanie sme mohli z ľubovoľného vrcholu. Naviac, vrcholom priraďujeme nové čísla (označené červenou), ktoré voláme **index**. Index vrcholom priraďujeme podľa poradia v akom sme ich našli. To znamená, že na ceste z koreňa do ľubovoľného vrchola indexy vrcholov rastú a spätná hrana vedie vždy do vrcholu s menším indexom (vedie do už skôr objaveného vrcholu).

Ďalej si uvedomme, že ak by sa v našom grafe nenachádzala spätná hrana, zadaný graf by bol strom a v strome je každý nelistový vrchol artikulácia. To čo zabraňuje vrcholu aby bol artikuláciou sú práve spätné hrany. To si môžeme všimnúť aj na obrázku 1. Vrchol 8 nie je artikuláciou, lebo ak ho odstránime, medzi vrcholmi 2 a 4 bude stále existovať cesta cez spätnú hranu vedúcu z vrchola 6 do vrchola 2.

A práve to je pointa riešenia. Ak po odstránení nejakého vrchola x zostane celý graf súvislý, tak komponenty ležiace pod x obsahujú spätnú hranu, ktorá ich spája s vrcholom s menším indexom ako má vrchol x. Na zisťovanie toho, či takáto hrana existuje si zavedieme nasledovnú hodnotu.

Low link vrchola v ($ll(v)$) je najmenšie číslo a, že existuje cesta z vrchola v skladajúca sa z niekoľkých (aj 0) stromových hrán (vedúcich dodola) a **práve jednej** spätnej hrany, a táto cesta vedie do vrcholu s indexom a.

Pomocou hodnoty $ll(v)$ teraz vieme vysloviť dôležitú vetu:

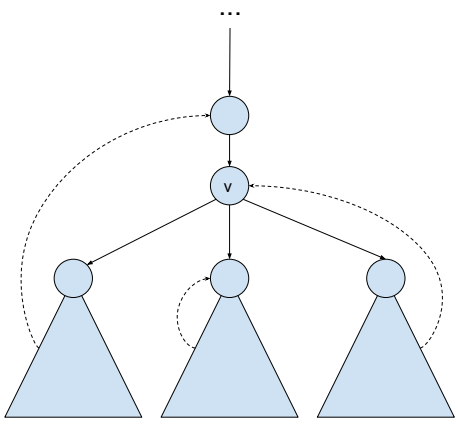
Veta: Nech v je vrchol grafu G, ktorý nie je koreň. Potom v je artikulácia práve vtedy ak aspoň pre jedného jeho syna w platí $Index[v] \leq ll(w)$.

Dôkaz: Nech w je syn vrchola v, pre ktorý platí spomenutá podmienka. To znamená, že žiadna spätná hrana z podstromu pod vrcholom w nevedie do vrcholu s menším indexom ako v. Ak preto odstránime tento vrchol, nebude existovať žiadne spojenie medzi týmto komponentom a zvyškom grafu.

Na dokázanie druhej implikácie použijeme nepriamy dôkaz. Nech pre všetkých synov vrchola v platí $ll(w) < Index[v]$. Potom z každého podstromu pod vrcholom v vedia hrana do komponentu nad vrcholom v. Ak preto odstránime vrchol v, graf zostane súvislý, preto v nebol artikulácia.

Dôvod, prečo sme toto nemohli tvrdiť o koreni DFS stromu je ten, že nikdy nevieme dosiahnuť menší index ako má koreň tohto stromu – 0. Uvedomme si však, že koreň je artikulácia práve vtedy, ak má aspoň dva podstromy. Medzi týmito podstromami totiž nevedia žiadna hrana (ak by viedla, tak pri prehľadaní jedného prehľadám aj druhý, a preto by neboli dva rôzne podstromy), preto odstránením koreňa vznikne viacej komponentov. V opačnom prípade odstránime iba list stromu a to zachová súvislosť.

Jediné čo nám zostáva je vedieť vypočítať hodnotu $ll(v)$. To však nie je ťažké, vieme to (neprekvapivo) spraviť v jednom DFS prechode. Z vrchola v totiž buď zoberieme priamu spätnú hranu, alebo sa vydáme jednou stromovou do nižšieho (už spracovaného) vrcholu w. Tam sa nám však neoplatí spraviť nič iné, ako použiť hodnotu $ll(w)$. Ak sa do takéhoto vrcholu vie dostať vrchol w, tak sa tam vie dostať aj vrchol v. Zo všetkých takýchto možností zoberieme tú najmenšiu.



Obr. 2: Obrázok ukazuje možné spätné hrany a im prislúchajúce low linky pre podstromy vrcholu v. Spätná hrana ľavého podstromu ide nad vrchol v a zabezpečuje súvislosť pri odstránení v. Stredný aj pravý podstrom však budú po odstránení v samostatné komponenty.

Obrázok 2 ilustruje možnosti pre low linky podstromov. Program naviac ukazuje implementáciu tohto algoritmu. Je treba si dať pozor, že hrana do otca nie je spätná a tiež treba špeciálne ošetriť koreň.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
using namespace std;

vector<vector<int> > G; //zoznam susedov reprezentujici graf
vector<bool> T;         //oznacenie navstivenia
vector<int> Index;
vector<int> LL;         //low link
int n,m;                //pocet vrcholov a hran
int cas=0;

void dfs(int v, int otec) { //vrchol v a jeho otec v strome
    T[v]=true;
    Index[v]=cas++;
    LL[v]=Index[v];
    bool artikulacia=false;
    int pocet_synov=0;
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];
        if(w == otec) continue;
        if(T[w]) {LL[v]=min(LL[v],Index[w]); continue;} //spatna hrana
        pocet_synov++;
        dfs(w,v);          //rekurzive sa zavolam na nenavstiveny vrchol
        LL[v]=min(LL[v],LL[w]);
        if(LL[w]>=Index[v]) artikulacia=true;
    }
    if(otec==-1 && pocet_synov>1) {          //koren
        printf("Vrchol %d je artikulacia\n",v);
    }
    else if(otec!=-1 && artikulacia) {       //nekoren
        printf("Vrchol %d je aritkulacia\n",v);
    }
}

int main() {
    T.resize(n,false);
    Index.resize(n); LL.resize(n);
    //nacitaj graf G
    dfs(0,-1);              //pusti prehladavanie z vrcholu 0
}
```

Podobne ako sme si zadefinovali vrchol, ktorého odstránením znesúvislými graf, si môžeme zadefinovať aj hranu. **Most** je taká hrana grafu G, že po jej odstránení sa graf rozpadne na viacero komponentov.

Možných riešení, ako nájsť všetky mosty grafu, je viacero. V jednom z nich si doprostred každej hrany vložíme nový vrchol. Následne nájdeme všetky artikulácie a tie nové vrcholy, ktoré sú označené za artikulácie boli vložené do hrán, ktoré boli mostami.

Druhé riešenie je použiť postup podobný ako pri hľadaní artikulácií. Presnejšie, hrana medzi vrcholmi v a w je most práve vtedy, ak platí $Index[v] < ll(w)$. Všimnite si ostrú nerovnosť. Dôkaz je analogický.

Silno súvislé komponenty

Nech G je **orientovaný** graf. **Silno súvislý komponent** je taká podmnožina vrcholov S, že existuje cesta z a do b pre ľubovoľné $a, b \in S$ a zároveň nevieme do tejto množiny pridať ďalší vrchol tak, aby táto vlastnosť platila naďalej.

Inak povedané, je to najväčšia množina vrcholov, v ktorej sa dá z každého vrcholu dostať do každého iného. Ako uvidíme neskôr, takáto množina môže byť užitočná, preto by sme chceli rozdeliť graf G na takéto množiny – rozdeliť ho na silno súvislé komponenty.

Uvedomme si, že ak skontrahujeme všetky takéto množiny do jedného vrcholu, dostaneme nový graf, ktorý bude acyklický a orientovaný. Ak by totiž obsahoval cyklus, tak všetky vrcholy (čo sú vlastne množiny) by tvorili väčší silno súvislý komponent, pretože by existovala cesta odvšadiaľ všade. Aj tento kontrahovaný graf sa nám neskôr zíde.

Ako však tieto silno súvislé komponenty nájsť? Opäť raz využijeme prehľadávanie do hĺbky. Začnime prehľadávať graf z ľubovoľného vrcholu. Postupne objavujeme nové vrcholy, ktoré môžu patriť rôznym komponentom. V nejakom momente však narazíme na vrchol v, ktorý leží v silno súvislom komponente S a pre tento komponent platí, že z neho nevedia žiadna hrana do iného komponentu. Toto musí nastať práve z dôvodu acyklickosti kontrahovaného grafu.

Uvedomme si nasledovnú vec. Vrchol v má spomedzi vrcholov množiny S najmenší index. A žiadna hrana vedúca z vrchola z množiny S nevedie do vrchola s menším indexom, pretože tieto hrany vedú iba v rámci množiny S. To znemená, že DFS algoritmus nájde vrchol v, potom postupne obíde všetky vrcholy množiny S (lebo z v vedie nejaký cesta do nich) a potom bude chcieť vyjsť von z vrchola v.

V tomto momente však vieme zistiť, že index vrchola v je rovnaký ako low link vrchola v. Naviac, pre žiaden iný vrchol $w \in S$ neplatí, že $ll(w) = Index[w]$. Pretože z vrchola w vedie cesta do v a preto jeho low link vedie do vrchola s menším indexom. Vrchol v je teda **prvý vrchol**, ktorý keď opúšťame, tak jeho low link sa rovná jeho indexu. V tom momente vieme, že všetky vrcholy ležiace pod ním patria do toho istého silno súvislého komponentu.

V tomto momente môžeme odstrániť množinu vrcholov S z grafu a pokračovať v prehľadávaní ďalej. Po nejakom čase sa opäť dostaneme do komponentu, z ktorého nevedú hrany do iného **ešte neobjaveného** komponentu. Je dôležité, že ešte neobjaveného. Môže sa totiž stať, že z neho vedú hrany napríklad do množiny S. Tú sa nám však už podarilo uzatvoriť a akoby v našom grafe neexistovala.

Asi najjednoduchšiu implementáciu vyššie spomenutého postupu navrhol Tarjan a funguje nasledovne. Púšťaj algoritmus na hľadanie silno súvislých komponentov postupne zo všetkých ešte nenavštívených vrcholov (môže sa stať, že nebude stačiť jedno prehľadávanie). Keď začneš spracovávať vrchol v, nastav mu index a hoď tento vrchol na samostatný stack, poznač si, že vrchol v je na stacku. Postupne prechádzaj všetkých susedov vrchola v označených w. Ak je w ešte nenavštívený, zavolaj túto funkciu rekurzívne. Ak je w navštívený, ale nie je na stacku, patrí do už spracovaného silno súvislého komponentu a môžeš ho ignorovať. Ak je navštívený, ale je na stacku, chovaj sa k tejto hrane ako k spätnej hrane. Pre vrchol v vypočítaj hodnotu $ll(v)$ pomocou hrán do synov. Ak je $ll(v)$ rovnaké ako index v, našiel si silno súvislý komponent. Všetky vrcholy, ktoré doň patria ležia na vrchu stacku a keďže vrchol v bol navštívený prvý, leží v stacku najhlbšie. Postupne vyberaj vrcholy zo stacku, až kým nenarazíš na vrchol v. Pre tieto vrcholy si zapamätaj, že už nie sú na stacku a poznač si, že patria do spoločného silno súvislého komponentu.

Túto nie až tak ťažkú myšlienku môžeme implementovať nasledovne v čase $O(n + m)$.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <stack>
using namespace std;

vector<vector<int> > G; //zoznam susedov reprezentujici graf
vector<int> T;          //oznacenie navstivenia (0-nenavstiveny, 1-na stacku, 2-navstiveny nie na stacku)
vector<int> Index;
vector<int> LL;         //low link
stack<int> S;           //stack
int n,m;                //pocet vrcholov a hran
int cas=0;

void dfs(int v) {
    S.push(v); T[v]=1;      //vlozim vrchol v na stack
    Index[v]=cas++;
    LL[v]=Index[v];
    for(int i=0; i<G[v].size(); i++) {
        int w=G[v][i];
        if(T[w] == 2) continue;
        if(T[w] == 1) {LL[v]=min(LL[v],Index[w]); continue;}
        dfs(w);
        LL[v]=min(LL[v],LL[w]);
    }
    if(LL[v] == Index[v]) {
        printf("Novy silno suvisly komponent obsahuje vrcholy:");
        int x;
        do {
            x=S.top(); S.pop();
            T[x]=2;
            printf(" %d",x);
        } while(v!=x);
        printf("\n");
    }
}

int main() {
    T.resize(n,0);
    Index.resize(n); LL.resize(n);
    //nacitaj graf G
    for(int i=0; i<n; i++)
        if(T[i] == 0) dfs(i);   //prehladava postupne zo vsetkych nenavstivenych vrcholov
}
```

2-SAT

Nech x_1 až x_n sú logické premenné nadobúdajúce iba hodnoty `True` (1) a `False` (0). Tieto premenné a ich negácie $\overline{x_1}$ až $\overline{x_n}$ nazývame literály. Klauzulou nazývame disjunkciu (or) niekoľkých literálov. Formulu nazývame konjunkciu (and) niekoľkých klauzúl. Problém SAT (satisfiability) je problém rozhodnuteľnosti, či sa dajú priradiť hodnoty premenným x_1 až x_n tak, aby bola splnená konkrétna formula φ. Naviac, ak každá klauzula formuly φ obsahuje najviac c literálov, voláme tento problém c-SAT.

Ako už možno viete (alebo sa dozviete), problém SAT je NP-úplný, nepoznáme preň polynomiálny algoritmus, ktorý by ho riešil. Dokonca už problém 3-SAT je NP-úplný. Napriek tomu je jednoduchší 2-SAT riešiteľný v polynomiálnom, dokonca lineárnom čase. V tejto časti sa pozrieme na to, ako.

Každá klauzula formuly φ v 2-SAT sa skladá z dvoch literálov, má preto tvar $a \vee b$. Aby bola formula φ splnená, musia byť splnené všetky klauzuly. S operáciou `or` sa pracuje pomerne zle. Oveľa vhodnejšia je implikácia. Uvedomme si však, že klauzula $a \vee b$ určuje dve implikácie $\bar{a} \Rightarrow b$ a $\bar{b} \Rightarrow a$ (ku každej implikácie vieme pridať aj jej obrátenú). Dostávame teda množinu implikácií, ktoré všetky musíme splniť.

Túto množinu implikácií si vieme pekne predstaviť ako graf. Každý literál x_1 až x_n a $\overline{x_1}$ až $\overline{x_n}$ bude predstavovať jeden vrchol. Následne ak máme implikáciu $a \Rightarrow b$ tak pridáme orientovanú hranu z vrcholu a do vrcholu b.

Uvedomme si, čo takýto graf reprezentuje. Ak existuje orientovaná cesta medzi dvoma vrcholmi c a d, tak to znamená, že $c \Rightarrow d$, pretože máme postupnosť na seba nadviazaných implikácií. Ešte dôležitejšie sú však cykly a obojsmerné cesty. Ak totiž $c \Rightarrow d$ a $d \Rightarrow c$, tak literály c a d sú **ekvivalentné**. To znamená, že im musíme priradiť rovnakú pravdivostnú hodnotu. No a z tohto pozorovania jasne vyplýva, že všetkým vrcholom v rovnakom silno súvislom komponente impikačného grafu musíme priradiť rovnakú hodnotu.

Prvý krok riešenia je teda vytvoriť silno súvislé komponenty a dostať skontrahovaný graf, kde každý vrchol predstavuje jeden silno súvislý komponent. Zamyslime sa teraz, kedy sa formula φ nedá splniť. Jedna zjavná podmienka je, že by sa premenná x_i a aj jej negácia $\overline{x_i}$ nachádzala v tom istom komponente. Tento komponent mi tvrdí, že sú ekvivalentné, ale to nie je možné, lebo sú navzájom negované. V takomto prípade je formula φ nesplniteľná. A ako sa ukáže, toto je jediný takýto prípad.

Prezreli sme teda každý silno súvislý komponent a zistili, že žiaden z nich neobsahuje premennú aj jej negáciu. Chceli by sme preto nájsť priradenie hodnôt premenným. Treba si uvedomiť dôležitú vec. Graf implikácií a aj skontrahovaný graf sú istým spôsobom symetrické, lebo implikácia má aj svoju obmenu. Presnejšie, ak platí implikácia $a \Rightarrow b$, tak platí aj $\bar{b} \Rightarrow \bar{a}$. Ak sú teda dve premenné ekvivalentné, aj ich negácie musia byť ekvivalentné. To znamená, že silno súvislé komponenty sa vyskytujú v dvojiciach. Ku každému silno súvislému komponentu S existuje silno súvislý komponent $\bar{S}$ obsahujúci negácie literálov v S. Naviac, ak máme silno súvislý komponent S, z ktorého nevedú žiadne hrany do iných komponentov, tak do $\bar{S}$ nevchádzajú žiadne hrany z iných silno súvislých komponentov (môžete si to odvodiť sporom pomocou obmeny implikácie).

V našom skontrahovanom grafe si preto zoberme silno súvislý komponent S, z ktorého nevychádzajú hrany a komponent $\bar{S}$, do ktorého hrany nevchádzajú. Vrcholom S priraďme hodnotu 1, z čoho vyplýva, že vrcholom $\bar{S}$ priradíme 0. Avšak $x \Rightarrow 1$ aj $0 \Rightarrow x$ sú vždy pravdivé. Takéto priradenie preto nemohlo porušiť žiadnu implikáciu a všetky implikácie týkajúce sa týchto vrcholov sú splnené bez ohľadu na hodnotu priradenú druhému koncu (ktorý je možno v inom silno súvislom komponente). Môžeme tieto vrcholy odstrániť a pokračovať rovnakým spôsobom s menším grafom. Nakoniec dostaneme validné ohodnotenie premenných x_1 až x_n spĺňajúce formulu φ. Na to všetko nám stačil čas $O(n + m)$, kde m je počet klauzulí.