Korespondenční Seminář z Programování

37. ročník **KSP** **Září 2024**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám první číslo hlavní kategorie 37. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Dále na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál**, jehož díly mohou vycházet samostatně.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek.** Úspěšným řešitelem se stává ten, kdo získá za celý ročník (hlavní kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepku na notebook a možná i další překvapení.

Termín série: **neděle 3. listopadu 2024 ve 32:00 (tedy další ráno v 8:00)**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/h/odevzdavatko/*

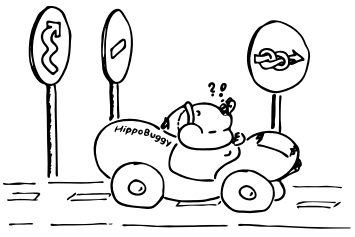
Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha; Experimentální (neobvyklá) úloha

Odměna série: **Odznáček do profilu na webu** si vysloužíte, když **z každé úlohy série získáte alespoň 2 body**.

První série třicátého sedmého ročníku KSP

37-1-1 Tramvajová zástavka — 10 bodů

Kevin si pozval na návštěvu několik kamarádů. Nikomu se od Kevina nechce odcházet, ale dříve nebo později budou muset jít. Každému kamarádovi jede domů nějaký spoj z blízké tramvajové zastávky, a tak Kevin navrhl, že na zastávku mohou jít všichni naráz v době, kdy se to všem „tak nějak“ bude hodit. Kdy to ale nastane?



Dostanete plán odjezdů příslušných tramvajových linek pro daný den a vaším úkolem bude najít co nejkratší časový úsek, kdy má každá linka alespoň jeden odjezd.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku dostanete dvě čísla K a N. K je počet různých tramvajových linek (čísel tramvají). Na dalších N řádcích dostanete vždy čas odjezdu tramvaje a její číslo. Čas bude v klasickém formátu `HH:MM:SS`, dvě číslice pro hodiny, dvě pro minuty a dvě pro sekundy, oddělené dvojtečkami.

Spoje budou vypsané chronologicky, tedy časy půjdou vzestupně. Může se stát, že ve stejný čas odjede víc tramvají. Slibujeme, že tramvaj každé linky odjede alespoň jednou za den.

Formát výstupu: Vypište čas začátku úseku (odjezd první tramvaje) a jeho délku v sekundách. Čas začátku zapište opět ve formátu `HH:MM:SS`. Ve vámi nalezeném intervalu musí odjíždět každá z K linek.

Pozor, hledaný úsek může trvat přes půlnoc (tramvaje jezdí každý den stejně)!

Ukázkový vstup:

```
3 6
09:10:00 1
09:13:40 3
09:13:40 1
09:14:32 1
09:15:00 2
09:22:15 3
```

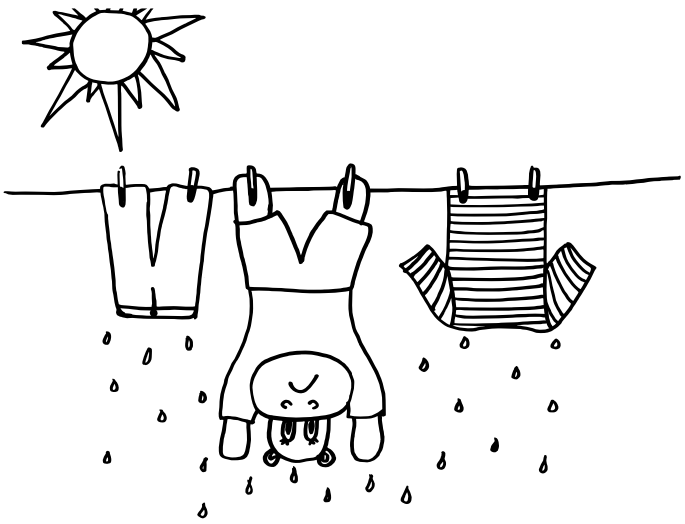
Ukázkový výstup:

```
09:13:40 80
```

37-1-2 Povodeň — 11 bodů

Po velkém požáru Smolince se místní obyvatelstvo rozhodlo zabránit budoucím požárům jednou provždy a postavilo megahydrant. Bohužel při slavnostním přestřihávání pásky omylem starosta zavadil o kohoutek a nyní se Smolincem šíří velká povodeň ...

Naštěstí je obyvatelstvo Smolince připraveno na katastrofy všeho druhu a mají zásobu protipovodňových bariér. Sami mají dost práce s evakuací, a tak se na vás obracejí s prosbou: Zjistěte, kam bariéry umístit.



Mapu Smolince si můžeme představit jako mřížku $R \times S$, kde každé pole je buď prázdné, zastavěné nebo megahydrant. Voda se šíří od megahydrantu pouze skrz prázdná pole. Na prázdné pole můžeme postavit protipovodňovou bariéru, čímž zabráníme šíření vody skrz něj. Nicméně bariéra musí mít pevné podpěry – Abychom na pole mohli postavit bariéru, buď severní a jižní pole musí být zastavěné, nebo východní a západní musí být zastavěné.

Najděte umístění bariér takové, že počet zatopených nezablokovaných polí bude co nejmenší. Pokud takových umístění existuje více, vypište to s nejmenším počtem použitých bariér.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku dostanete dvě čísla R a S: počet řádků a počet sloupců. Na každém z dalších R řádků naleznete S znaků popisujících mapu Smolince. Znak `H` značí megahydrant, `.` značí prázdné pole a `#` značí zastavěné pole.

Formát výstupu: Na prvním řádku vypište číslo M – počet použitých bariér. Na dalších M řádcích vypište čísla x_i, y_i – sloupec a řádek i-té postavené bariéry.

Ukázkový vstup:

```
6 5
##.#.
#H.##
#....
#.###
#.#..
.....
```

Ukázkový výstup:

```
3
3 1
4 3
2 4
```

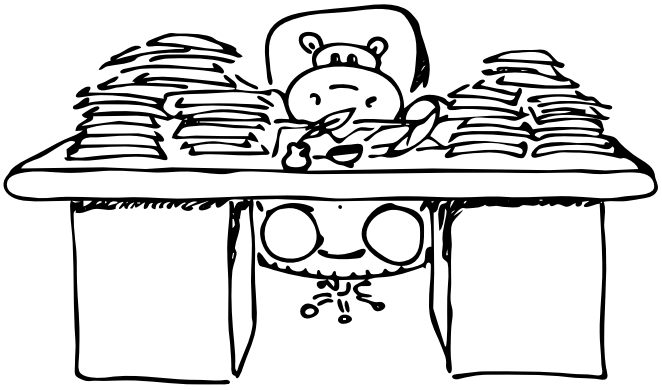
Při postavení bariér na příslušná pole bude Smolinec zatopen následovně (`B` značí bariéru, `W` vodu):

```
##B#.
#WW##
#WWB.
#B###
#.#..
.....
```

Nelze mít méně než 4 zatopená pole. Není možné dosáhnout 4 zatopených polí s méně jak 3 bariérami.

37-1-3 Zpřeházená fronta — 12 bodů

Na studijní oddělení Hroší univerzity v Hroším Týnci dorazila skupina N studentů, kteří se dopředu objednali pro vyzvednutí studentského průkazu. Každý z nich stojí na nějaké z vyznačených pozic, které na studijním zůstaly z protipandemických opatření.



Studenti ale dorazili v jiném pořadí, než ve kterém byli objednáni. Nyní je potřeba je správně seřadit, aby se paní úřednici nerozbil informační systém. Každý student ví, na kterou pozici byl předem objednán a na jaké pozici nyní stojí.

Aby se organizace přesunu studentů ve frontě zjednodušila, chtěla by paní úřednice pro každou pozici na studijním oddělení určit, odkud se na ni má patřičný student přemístit.

Je tu však menší problém. Paní úřednice, která studenty potřebuje přeřadit, trpí anterográdní amnézií a nedokáže si pamatovat údaje o všech studentech najednou.

Trochu formálněji: Vyznačené pozice na podlaze studijního oddělení si můžete představit jako pole A, které máte již načtené v paměti a můžete ho měnit. Na začátku výpočtu $A[i]$ říká, na kterou pozici se má přesunout student, který původně stál na na pozici i. Na konci výpočtu má být $A[i]$ rovno původní pozici studenta, který se má přesunout na pozici i. Po celou dobu výpočtu smí pole obsahovat pouze celá čísla od 1 do N. Jediná další paměť, kterou máte k dispozici, je konstantně mnoho proměnných pro celá čísla velká $\mathcal{O}(N)$.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

37-1-4 Nuly a jedničky — 12 bodů

Barnabáš se začetl do Pohádek tisíce a jedné noci. Při tom si uvědomil, jak zajímavé je číslo 1001 – je dělitelné nejen 7, ale také 11 a 13. Tak začal hledat jiná čísla z nul a jedniček, která mají zajímavé dělitele.

Napište mu program, který pro zadané $N \geq 1$ najde nejmenší celé kladné číslo dělitelné N, v jehož desítkovém zápisu se vyskytují jen nuly a jedničky. Případně odpoví, že žádné takové číslo neexistuje.

Například pro $N = 4$ je řešením číslo 100, pro $N = 5$ je to 10, pro $N = 7$ je to 1001 a pro $N = 11$ číslo 11.

37-1-X1 Bomberman — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Už to bylo chvíli, co slavný Bomberman pokládal výbušniny… Po vlně velké slávy si koupil vzducholoď, se kterou procestoval svět. Poté ale začal stárnout, jeho bomby se začaly rozbíjet a už nemohl chodit jako dřív. Nyní se však rozhodl všem ukázat, že ještě bomby pokládat umí.

Bomberman se rozhodl vyprázdnit mřížku $R \times S$, kde každé políčko je buď prázdné, nebo zničitelné. Momentálně se nachází ve své vzducholodi, a tak může pokládat bomby na úplně **libovolné** pole (i zničitelné). Nicméně jeho bomby částečně nefungují. Bomba dokáže zničit buď celý sloupec, ve kterém je položena, nebo celý řádek, ve kterém je položena, ale ne obojí zároveň. Bomberman by měl jít co nejdříve do postele, a proto chce položit co nejméně bomb tak, aby zničil všechna zničitelná políčka. Poraďte mu, kam je má položit.

37-1-S Návrat virtuálního stroje — 15 bodů

Letošním ročníkem vás bude provázet seriál. V každé sérii se objeví jeden díl, který bude obsahovat nějaké povídání a navíc úkoly. Za úkoly budete moci získávat body podobně jako za klasické úlohy série. Abyste se mohli zapojit i během ročníku, seriál bude možné odevzdávat i po termínu za snížený počet bodů. Vzorové řešení seriálu proto před koncem celého ročníku KSP uvidí jen ti, kdo už seriál odevzdali.

V tomto ročníku jsme si pro vás připravili praktický seriál o překladačích programovacích jazyků. Programovací jazyky a překladače představují obrovskou krajinu k prozkoumávání. Spousta informatiků strávila celé kariéry tím, že se touto zemí procházeli, aniž by někdy dosáhli jejího konce. Pokud do této země právě vstupujete poprvé, vítejte.

V průběhu příštích několika měsíců si ukážeme část této krajiny. Konkrétně si spolu sestrojíme jednoduchý překladač našeho vymyšleného programovacího jazyka. Přiblíží nám to vhled do produkčních překladačů a jazyků, které překládají. Se základními znalostmi jazyka a místních zvyků může každý z vás pokračovat svou vlastní cestou.

Úkolem překladače je přeložit zdrojový kód do kódu strojového. Zdrojovým kódem pro nás bude text v našem vymyšleném programovacím jazyce (který si představíme příště). Strojový kód je sekvence primitivních instrukcí, které nějaký stroj dokáže vykonat. Jeden strojový kód si ukážeme dnes.

Práce překladače se dá rozdělit do několika fází. Každá fáze transformuje vstupní program z jedné reprezentace do jiné (na začátku zdrojový kód, na konci strojový kód). Všechny reprezentace jsou přitom dost různé.

Transformace se dají rozdělit do dvou skupin. První analyzují a vytváří strukturu – těm se říká frontend překladače. Někde v polovině překladu nastane zlom a překladač se pustí do druhé skupiny transformací – backend. Ty pak tuto analyzovanou strukturu rozkládají a tvoří finální reprezentaci.

Celý průběh překladu si můžeme představit jako výlet začínající v údolí Zdrojového Kódu, který pokračuje výlezem na vrcholek hory Programové Analýzy, ze které pak zase musíme slézt dolů na druhou stranu do údolí Strojového Kódu.

V rámci tohoto seriálu si celou tuto stezku projdeme. Inspirovali jsme se ale Georgem Lucasem a dneska začneme uprostřed naší cesty: na útulné horské chatě těsně za vrcholem.

Návrat virtuálního stroje

Skutečné stroje dnešní doby jsou pro naše účely zbytečně složité. Dnes tedy začneme tím, že si vyrobíme simulátor jednoduššího stroje. V příštích dílech si postavíme překladač do tohoto virtuální stroje a na konci seriálu opustíme virtuální stroj a přesuneme se ke strojům fyzickým.

Ačkoliv se může zdát, že virtuální stroj má využití pouze jako mezikrok ke skutečným strojům, interprety jsou v dnešní době velmi populární. Python, JavaScript, a do jisté míry i C# nebo Java jsou všechny založené na nějakých virtuálních strojích. Výhodou tohoto přístupu je například, že stejný kód může běžet na různých fyzických strojích bez jakýchkoliv modifikací. Virtuální stroj slouží jako abstrakce nad skutečným hardwarem a poskytuje identické prostředí na různých platformách. Na druhou stranu virtuální stroj s sebou nese i velkou cenu. Například simulace virtuálního stroje nám často řádově zpomalí běh programu. Stejně jako všechno v životě, je to kompromis. Zpátky k našemu virtuálnímu stroji.

Od našeho virtuální stroje budeme chtít, aby uměl vykonávat primitivní instrukce které operují (zatím pouze) s čísly. Všechny hodnoty, se kterými stroj pracuje, si budeme ukládat na zásobník a instrukce budou uložené v poli. Operace stroje bude vypadat takto:

1. přečti aktuální instrukci
2. vykonej ji
3. posuň se o 1 instrukci dál („instruction pointer“ zvyš o 1)
4. opakuj

Vykonání instrukce bude často vypadat přibližně takto:

1. odeber několik hodnot ze zásobníku
2. proveď na nich danou operaci
3. přidej výsledek na zásobník

Jak jste měli možnost si vyzkoušet v 36-2-4 a 35-4-1, programování čistě zásobníkového stroje není úplně přívětivé. Bez instrukce pro indexování zásobníku je to dokonce prakticky nemožné. Kromě zásobníku tedy naučíme náš stroj zároveň pracovat s pojmenovanými proměnnými a přidáme instrukce na zkopírování hodnoty ze zásobníku do proměnné a naopak. Zásobník pak budeme používat primárně pro mezivýsledky a ostatní věci ukládat do proměnných.

Data, se kterými bude náš virtuální stroj pracovat, můžeme reprezentovat například takto:

(V průběhu seriálu budeme používat C++23. Řešení akceptujeme prakticky v libovolném jazyce, ale pokud chcete plný počet bodů, tak svoje řešení napište v jazyku, který sám není interpretovaný. Speciálně považujeme za rozumné: C++

(*ehm*), Rust, Zig, Go, C#, F#, ML, OCaml, Java, Kotlin, Scala, Nim, Swift, D, Odin, Crystal.)

```
#include <string>
#include <unordered_map>
#include <variant>
#include <vector>

using namespace std;

enum Opcode {
  OP_PRINT,
  OP_PUSH,

  OP_LOAD,
  OP_STORE,

  OP_ADD,
  OP_SUB,
  OP_MUL,
  OP_DIV,
  OP_DUP,
};

struct Instruction {
  // typ instrukce
  Opcode op;

  // některé instrukce si potřebují
  // pamatovat další hodnoty
  variant<monostate, int, string> value = monostate{};
};

vector<Instruction> instrukce;
vector<int> stack;
unordered_map<string, int> promenne;
int ip;
```

Úkol 1 – **Aritmetické operace** [5b]:

Prvním úkolem bude naimplementovat instrukce pro základní aritmetické operace (sčítání, odčítání, násobení a dělení) a instrukci `DUP`. Aritmetické instrukce dělají, co byste ze jména čekali. Odeberou dvě hodnoty ze zásobníku, provedou na nich operaci, kterou mají ve jméně, a výsledek přidají zpátky na zásobník.

Například pokud máme na zásobníku hodnoty `6 12 33 -3`, po provedení instrukce `OP_MUL` bude zásobník vypadat takto: `6 12 -99`. Pokud by následovala instrukce `OP_SUB`, bude zásobník obsahovat hodnoty `6 111`.

Budeme chtít, aby druhý operand byla první hodnota ze zásobníku a první operand druhá hodnota ze zásobníku. To se může zdát lehce neintuitivní, ale usnadní nám to práci v příštích dílech seriálu.

Instrukce `OP_DUP` odebere hodnotu ze zásobníku a ihned jí tam zase dvakrát přidá. Vypadá to, jako kdyby se vrchní hodnota zásobníku duplikovala.

Dnešní úkoly můžete implementovat do následující šablony virtuálního stroje. Nezapomeňte si zkopírovat definice `Opcode` a `Instruction` výše.

```
#include <print>

using namespace std;

void interpret(
    vector<Instruction> const &instrukce,
    vector<int> &zasobnik,
    unordered_map<string, int> &promenne) {
  int ip = 0; // index aktuální instrukce
              // (instruction pointer)
  while (true) {
    if (!(ip >= 0 && ip < ssize(instrukce)))
      break;

    auto ins = instrukce.at(ip);

    switch (ins.op) {
    case OP_PRINT: {
      println("PRINT: {}", zasobnik.back());
    } break;

    case OP_LOAD: {
      zasobnik.push_back(
          promenne[get<string>(ins.value)]);

    } break;

    case OP_STORE: {
      promenne[get<string>(ins.value)] =
          zasobnik.back();
      zasobnik.pop_back();
    } break;

    case OP_PUSH: {
      zasobnik.push_back(get<int>(ins.value));
    } break;

    case OP_ADD: {
      // TODO
    } break;
    case OP_SUB: {
      // TODO
    } break;
    case OP_MUL: {
      // TODO
    } break;
    case OP_DIV: {
      // TODO
    } break;
    case OP_DUP: {
      // TODO
    } break;
    }
    ip += 1;
  }
}
```

Funkci `interpret` bychom pak mohli volat takto:

```
int main() {
  vector<int> zasobnik;
  vector<Instruction> instrukce(
      {Instruction{.op = OP_PUSH, .value = 8},
       Instruction{.op = OP_STORE, .value = "x"},
       Instruction{.op = OP_LOAD, .value = "x"},
       Instruction{.op = OP_PRINT}});
  unordered_map<string, int> promenne;

  interpret(instrukce, zasobnik, promenne);
}
```

Pro příjemnější vývoj doporučujeme zapnout sanitizátory a varování:

```
c++ -std=c++23 -Wall -Wextra \
    -fsanitize=address,undefined ./main.cpp
```

V každém pořádném programovacím jazyce se lze donekonečna zacyklit, ale náš virtuální stroj to zatím vůbec nepodporuje. Pokud jste zvyklí, že podmínky i cykly vyžadují nějaké vnořování bloků, tak se teď možná bojíte, že jejich přidáním všechno zkomplikujeme. Naštěstí je to přesně opačně. Nic jako `OP_SLOŽENÁ_ZÁVORKA` nebudeme potřebovat a cykly i podmínky vyřešíme jednou triviální instrukcí.

Bude to instrukce podmíněného skoku, typicky nazývaná „branch“. Taková instrukce sebere vrchní hodnotu ze zásobníku a podle ní se rozhodne, jestli bude prostě pokračovat na další instrukci v pořadí, anebo skočí a program bude pokračovat někde jinde. `If` pomocí této instrukce můžeme

nasimulovat tak, že daný blok přeskočíme, pokud podmínka neplatí. `While` cyklus uděláme velmi podobně, akorát na konec bloku přidáme nepodmíněný skok zpět.

Úkol 2 – Podmíňený skok [5b]:

Přidejte instrukci `OP_BRANCH`, která skočí na instrukci s indexem uvedeným ve `value` za podmínky, že hodnota sebraná ze zásobníku není nula. Kde bude interpret pokračovat, ovlivníte změnou instruction pointeru ip.

Bude se nám hodit několik dalších instrukcí pro práci s hodnotami `TRUE` a `FALSE`. Speciální datový typ nepotřebujeme, `FALSE` budeme reprezentovat jako nulu a `TRUE` jako jedničku. Přidejte také následující „logické“ operace:

- `OP_NOT` – Sebere hodnotu ze zásobníku. Pokud to byla nula, přidá 1, jinak přidá 0.
- `OP_EQ` – Sebere dvě hodnoty ze zásobníku. Pokud se rovnají, přidá 1, jinak přidá 0.
- `OP_LT` – Sebere ze zásobníku hodnoty B a A. Pokud je $A < B$, přidá 1, jinak přidá 0.

Úkol 3 – Jednoduchý program [5b]:

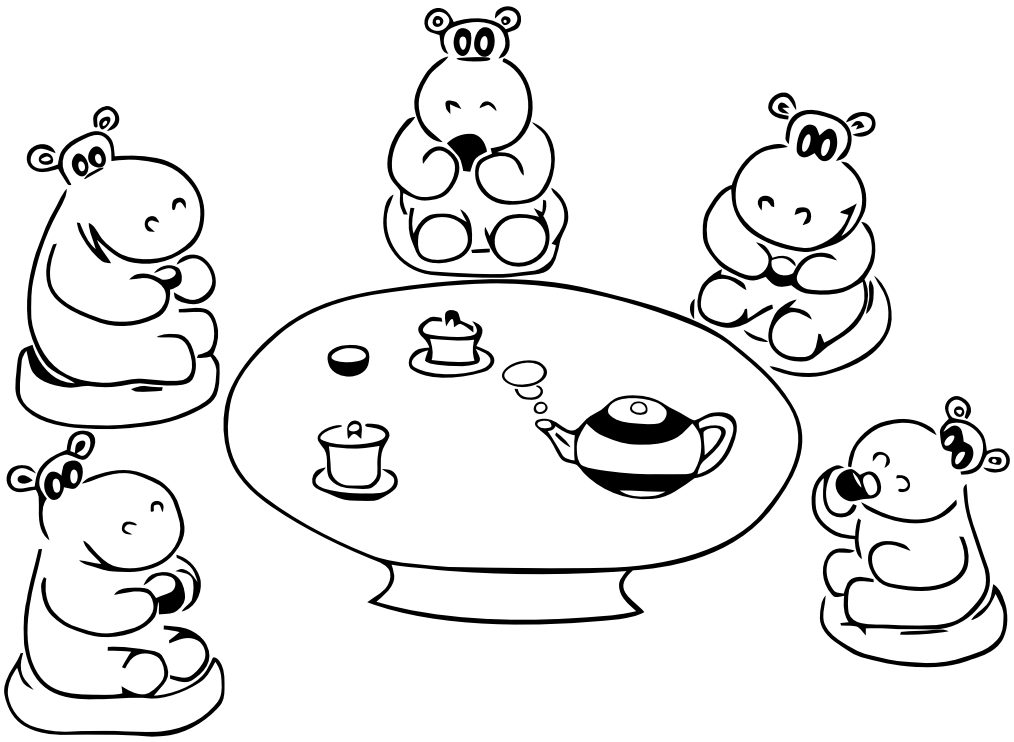
Dnešním posledním úkolem bude využít všechny funkce virtuálního stroje, které jsme si připravili, a napsat si nějaký jednoduchý program.

Napište program pro váš virtuální stroj na počítání faktoriálů. Můžete si vybrat, jestli vstup zadrátujete jako proměnnou, nebo jako hodnotu na zásobníku. Využijte přitom instrukce pro přesun hodnot mezi proměnnými a zásobníkem, aritmetiku a skoky.

Předpokládáme, že váš program bude i se vstupem zadrátovaný ve funkci `main` a že výsledek vypíše do konzole pomocí instrukce `OP_PRINT`.

Všechny 3 úkoly odevzdejte najednou jako jeden ZIP soubor obsahující všechen váš zdrojový kód. Oceníme i slovní popis zvoleného řešení, bohatě by měly stačit komentáře v kódu. Když budete mít jakýkoliv dotaz nebo problém, tak se nebojte zeptat na našem Discordu či pomocí emailu.

Prokop Randáček, Standa Lukeš & Ondra Machota

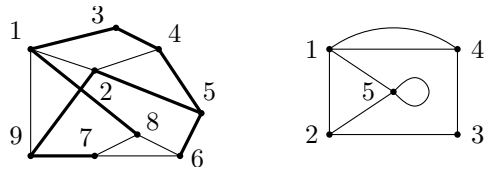


Recepty z programátorské kuchařky: Grafy

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran E, což jsou neuspořádané dvojice vrcholů. Hrana $e = \{x, y\}$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* budeme říkat takové posloupnosti vrcholů a hran tvaru v_1, e_1, v_2, e_2, …, e_{n-1}, v_n, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, totiž obsahuje nějaký vrchol u dvakrát. Existuje tedy $i < j$ takové, že $u = v_i = v_j$. Pak ale můžeme z našeho sledu vypustit posloupnost e_i, v_{i+1}, …, e_{j-1}, v_j a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

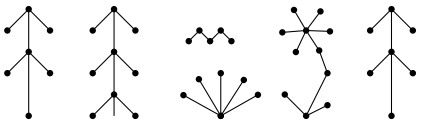
Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty navíc platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé.* Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti.*

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy,* jelikož každá komponenta souvislosti takového grafu je strom.



Les, jak ho vidí matematici

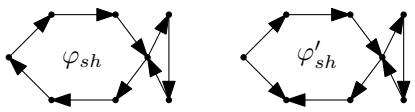
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu říkáme každému jeho podgrafu, který je stromem a spojuje všechny vrcholy grafu. Můžeme ji například získat tak, že dokud jsou v grafu kružnice, odebíráme hrany ležící na nějaké kružnici. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je jedna z koster levého grafu znázorněna silnými hranami.

Cvičení: Zkuste si dokázat, že stromy jsou právě grafy, které jsou souvislé a mají o jedna méně hran než vrcholů.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf.* Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro grafy orientované, jen si musíme dát pozor na směr hran.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme slabou a silnou souvislost: *slabě souvislý* je graf tehdy, pokud se z něj zapomenutím orientace hran stane souvislý neorientovaný graf. *Silně souvislým* ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x a y orientovaná cesta v obou směrech. Pokud je graf silně souvislý,

je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Ohodnocené grafy

Další možností, jak si graf „vyzdobit“, je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny*. Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

Podobně můžeme přiřazovat ohodnocení i vrcholům, ale raději si všechny operace s ohodnocenými grafy necháme na některé z dalších dílů Kuchařky. I tak budeme mít práce dost a dost.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. To můžeme udělat například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

```
  123456789
1 011000011
2 100110001
3 100100000
4 011010000
5 010101000
6 000010110
7 000001011
8 100001100
9 110000100
```

- *matice sousednosti* – to je pole A velikosti $N \times N$. Na pozici $A[i,j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j hrana vede (1), nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.
- *seznam sousedů* je obvykle tvořen dvěma poli: polem sousedů $S[1 \ldots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \ldots N]$, v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do $Z[N+1]$ uložíme $M+1$, bude platit, že sousedé vrcholu i jsou uloženi v $S[Z[i]], \ldots, S[Z[i+1]-1]$. Tato reprezentace má tu výhodu, že zabírá pouze prostor $\mathcal{O}(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2

i	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$S[i]$	4	6	5	7	8	6	8	9	1	6	7	1	2	7

i	1	2	3	4	5	6	7	8	9	10
$Z[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

- *půlhranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půlhrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici a na vrchol, ze kterého vychází.

V následujících receptech budeme vždy používat seznamy sousedů, poli S budeme říkat `sousedi`, poli Z `zacatky`. Pole `sousedi` bude mít rozsah od 0 do $M-1$, pole `zacatky` od 0 do N (kde N je počet vrcholů a M počet hran).

Prohledávání do hloubky

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. Anglický název je (překvapivě) *last in, first out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme do zásobníku a označíme.
4. Kroky 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládejme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M, čili $\mathcal{O}(N + M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
oznacen = [False] * N

def projdi(v):
  oznacen[v] = True
  for i in range(zacatky[v], zacatky[v+1]):
    if not oznacen[sousedi[i]]:
      projdi(sousedi[i])
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu, a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $\mathcal{O}(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $\mathcal{O}(N + M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $\mathcal{O}(N + M)$.

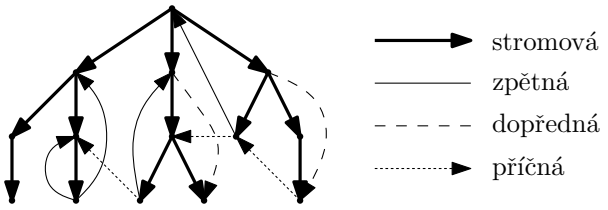
```
def projdi(v):
  komponenta[v] = nova_komponenta
  for i in range(zacatky[v], zacatky[v+1]):
    if komponenta[sousedi[i]] == -1:
      projdi(sousedi[i])

...
for i in range(N):
  komponenta[i] = -1
nova_komponenta = 1
for i in range(N):
  if komponenta[i] == -1:
    projdi(i)
    nova_komponenta += 1
...
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom – podle anglického názvu Depth-First Search pro prohledávání do hloubky). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit. Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *zpětné hrany*. *Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla doprava, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; doleva rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.

Prohledávání do hloubky lze tedy také využít k nalezení kostry neorientovaného grafu, což je strom, který jsme prošli. Rovnou při tom také zjistíme, zda graf neobsahuje cyklus: to poznáme tak, že nalezneme zpětnou hranu různou od té stromové, po níž jsme do vrcholu přišli.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromu shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou zprava doleva, čili pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).



Strom prohledávání do hloubky a typy hran

Prohledávání do šířky

Prohledávání do šířky je založené na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá frontu:

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jehož $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u]+1$.
4. Kroky 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n + 1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do v. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a její předposlední vrchol z. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, a to je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $\mathcal{O}(N+M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
...
for i in range(N):
  H[i] = -1
prvni = 1
posledni = 1
fronta[prvni] = pocatecni_vrchol
H[pocatecni_vrchol] = 0

while True:
  v = fronta[prvni]
  for i in range(zacatky[v], zacatky[v+1]):
    if H[sousedi[i]] < 0:
      H[sousedi[i]] = H[v] + 1
      posledni += 1
      fronta[posledni] = sousedi[i]
  prvni += 1
  if prvni > posledni: break
...
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy aby pro každou hranu $e = (v_i, v_j)$ bylo $i > j$. Představme si to jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu $v_1, \ldots, v_n$, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp. z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_n$ než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což nelze splnit.

Cyklus je ovšem to jediné, co může existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat *stok*). Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujme po ní do nějakého vrcholu v_2, z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.
- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě, a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolovat si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $\mathcal{O}(N+M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili nižší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $\mathcal{O}(N+M)$.

```
def projdi(v):
  # zatím V jen označíme
  ocislovani[v] = 0
  for i in range(zacatky[v], zacatky[v+1]):
    if ocislovani[sousedi[i]] == -1:
      projdi(sousedi[i]]
  posledni += 1
  ocislovani[v] = posledni

...
for i in range(N):
  ocislovani[i] = -1
posledni = 0
for i in range(N):
  if ocislovani[i] == -1:
    projdi(i)
...
```

Hranová a vrcholová 2-souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most.*

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky a DFS strom. Všimněme si, že mostem může být jedině stromová hrana – každá jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“

pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany.*

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $\mathcal{O}(N+M)$. Zde jsou důležité části programu:

```
def projdi(v, nova_hladina):
  hladina[v] = nova_hladina
  spojeno[v] = hladina

  for i in range(zacatky[v], zacatky[v+1]):
    w = sousedi[i]
    if hladina[w] == -1:
      projdi(w, nova_hladina + 1)
      if spojeno[w] < spojeno[v]:
        spojeno[v] = spojeno[w]
      if spojeno[w] > hladina[v]
        dvoj_souvisle = False
    else:
      if hladina[w] < nova_hladina - 1
         and hladina[w] < spojeno[v]:
        spojeno[v] = hladina[w]

...
for i in range(N):
  hladina[i] = -1
dvoj_souvisle = True
projdi(1, 0)
...
```

Další formou souvislosti je *vrcholová souvislost*. Graf je *vrcholově 2-souvislý*, právě když:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

Artikulace je takový vrchol, který když odebereme, zvýší se počet komponent souvislosti grafu.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou; sami zkuste najít, které nerovnosti.

Dnešní menu servírovali

Martin Mareš, David Matoušek a Petr Škoda

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*