Korespondenční Seminář z Programování

33. ročník **KSP** **Duben 2021**

Přinášíme vám řešení čtvrté série tohoto ročníku KSP. Čtení vzorových řešení je skvělý způsob, jak se naučit nové způsoby řešení úloh, a tak neváhejte a začtěte se do nich.

Připomínáme, že **seriál můžete za menší bodové ohodnocení stále odevzdávat** a to až do konce ročníku, takže jeho vzorové řešení vám vydáme až po skončení celého ročníku. Také odevzdávání úlohy **33-4-X1 Koření a recepty** jsme **prodloužili až do konce 5. série**, zatím vám namísto řešení přinášíme nápovědu.

Vzorová řešení čtvrté série třicátého třetího ročníku KSP

33-4-1 Ohňostroj

Odpalovací zařízení mají definované pořadí, říkejme mu třeba „zleva doprava“. Nejprve zařídíme, aby kdykoliv někdo zmáčkne tlačítko na kterémkoliv zařízení, dozvědělo se o tom nejlevější zařízení. To zařídíme vysláním *signálu* od tlačítka doleva. V každém tiku se každé zařízení podívá na svého pravého souseda. Pokud už je soused ve stavu „zmáčknuto“, samo také přejde do stavu „zmáčknuto“. Signál po nejvýše N tikách doputuje k nejlevějšímu zařízení.

Dvě místa současně

Teď vyřešíme, jak zasynchronizovat nějaká dvě zařízení ℓ a p, kde ℓ je nalevo od p. Tedy jak způsobit, aby nějaká událost v zařízení ℓ způsobila jinou událost současně v zařízeních ℓ a p. Provedeme to takto: Vyšleme z ℓ směrem k p dva signály, říkejme jim třeba A a B, přičemž A se bude šířit plnou rychlostí a B rychlostí poloviční (posune se o jeden krok za dva tiky). Jakmile A dorazí do p, „odrazí se“ a bude pokračovat jako A' stejnou rychlostí zpět k ℓ. Všimněte si, že A' se vrátí do ℓ přesně v okamžiku, kdy B dorazí do p.

Program (C++):
`http://ksp.mff.cuni.cz/viz/33-4-1-dvojice.cpp`

Simulátor odpalovacích zařízení (C++):
`http://ksp.mff.cuni.cz/viz/33-4-1-simulator.cpp`

Všechna místa současně

Když už umíme synchronizovat dvojice zařízení, můžeme použít metodu Rozděl a panuj a vyřešit problém pro celou posloupnost. Uvažujme interval zařízení $\langle \ell, p)$, na počátku je ℓ nejlevější zařízení a p těsně za nejpravějším (takové zařízení doopravdy neexistuje, ale poslední si na něj může hrát). Nalezneme zařízení s v polovině intervalu a to zasynchronizujeme s ℓ. Tím se nám interval rozpadne na poloviny $\langle \ell, s)$ a $\langle s, p)$. V obou polovinách pak současně spustíme tentýž algoritmus.

Pokud bude počet zařízení $N = 2^k$, po k iteracích se dobereme k jednoprvkovým intervalům (ke všem současně), a tehdy už může každé zařízení rovnou odpálit.

Zbývá dořešit hledání středu. K tomu můžeme použít opět šíření signálů různými rychlostmi: z levého okraje intervalu vyšleme doprava signál C plnou rychlostí a signál D rychlostí třetinovou. Signál C se od pravého okraje odrazí jako C' a na cestě zpět se potká se signálem D přesně v polovině intervalu (pokud si vrcholy očíslujeme $0, \ldots, N-1$ a N bude sudé, potkají se ve vrcholu $N/2$).

Každá iterace tohoto algoritmu potřebuje lineární počet tiků vzhledem k délce intervalů, které se zrovna zpracovávají. Celková časová složitost tedy bude $\mathcal{O}(N+N/2+N/4+\ldots+1) = \mathcal{O}(N)$.

Kdyby N nebylo mocninou dvojky, museli bychom se vyrovnat s nerovnoměrným dělením intervalů. To ale není nijak těžké: při hledání středu v intervalu liché délky se zastavíme na $(N-1)/2$, a dokonce poznáme, že se nám to stalo, podle toho, že se signály nepotkají přesně na hranici periody signálu D. Pak můžeme prostřední vrchol intervalu vynechat a zpracovat zbylé stejně velké části identicky. Nakonec nám zbudou jednoprvkové intervaly a mezi některými z nich osamocené vrcholy. Stačí tedy, aby zařízení místo odpálení ohňostroje nastavila nějakou značku „připravit se“ a v následujícím tiku odpálí každé zařízení, které buďto tuto značku má, nebo ji vidí u svého souseda. Program si tím ovšem komplikovat nebudeme.

Program (C++):
`http://ksp.mff.cuni.cz/viz/33-4-1-odpal.cpp`

Úlohu připravili: Jirka Kalvoda, Martin „Medvěd“ Mareš

P.S.: Medvěd děkuje svému mladšímu já za inspiraci v podobě úlohy P-II-2 z 56. ročníku MO kategorie P.

33-4-2 Firewall

V této úloze bylo úkolem zpracovávat tři události – blokaci nebo povolení nějakého rozsahu IP adres a dotaz na to, jestli je daná IP adresa povolená. Nejdříve si ukážeme dvě pomalejší řešení, která jsou ale zajímavá nápadem, a nakonec předvedeme řešení pomocí *líných stromů,* které je již dostatečně rychlé na vyřešení všech vstupů.

Nejprve si však povíme, jak s IP adresami efektivně pracovat. Už v zadání jsme zmínili, že IPv4 adresy jsou vlastně 32bitová čísla a jejich běžná reprezentace v programu je buď čtveřice bajtů, nebo přímo jedno čtyřbajtové číslo (třeba `uint32_t` v Céčku). Pokud načítáme IPv4 adresu jako čtveřici čísel oddělených tečkami, tak si je na 32bitové číslo můžeme převést třeba pomocí funkce `ip` níže (každý bajt jen posuneme na správné číslo a uděláme bitový OR).

Stejně tak si můžeme z počtu bitů v CIDR formátu (číslo K za lomítkem) vyrobit *masku,* která bude mít prvních K bitů jedničky a zbytek nuly. Můžeme to udělat třeba jako ve funkci `mask` níže.

```
typedef unsigned char byte;
uint32_t ip(byte a, byte b, byte c, byte d) {
  return (a << 24) | (b << 16) | (c << 8) | d;
}
uint32_t mask(int K) {
  if (K > 0)
    return (0xFFFFFFFF << (32-K)) & 0xFFFFFFFF;
  else
    return 0;
}
```

K čemu nám takové reprezentace jsou? Můžeme s nimi rychle provádět bitové operace. Například test, jestli adresa patří do nějaké sítě, provedeme tak, že uděláme bitový AND adresy a masky sítě. Pokud se výsledné číslo shoduje s prefixem sítě, adresa do ní skutečně patří:

```
if ((adresa & maska) == prefix) { je v síti... }
```

A nakonec, pokud chceme otestovat, jestli je K-tý bit adresy jednička, můžeme to udělat třeba takto (indexujeme od jedničky):

```
if (adresa & (1 << (32-K))) { K-tý bit je 1... }
```

Teď již máme nějaký základní arzenál pro rychlé operace s IPv4 adresami, pojďme se podívat na řešení. Abychom si zjednodušili popis, tak odteď budeme používat výraz *operace* pro blokaci nebo povolení.

Přímočaré řešení

Protože se operace navzájem přepisují, tak nás pro každý dotaz zajímá jen ta poslední operace, která dotazovanou IP adresu ovlivnila. Můžeme si tedy všechny operace ukládat do seznamu a pro každý dotaz celý seznam projít. Pro každou operaci otestujeme, jestli dotazovaná adresa patří do podsítě, na které byla operace provedena – pokud ano, tak si zapamatujeme typ operace a pokračujeme dál. Nakonec jen vypíšeme poslední zapamatovanou operaci.

Lze lehce nahlédnout, že toto řešení vždy vrátí správnou odpověď, ale bude muset pro každý dotaz projít všechny doposud provedené operace. Pokud si jako N označíme počet řádků na vstupu a budeme předpokládat, že polovina z nich jsou operace a druhá polovina dotazy, tak můžeme odhadnout, že zpracování jednoho dotazu bude trvat řádově $\mathcal{O}(N)$ a celkově se tak dostaneme na čas $\mathcal{O}(N^2)$.

Zlepšit to sice můžeme procházením seznamu odzadu a zastavením se u první operace ovlivňující dotazovanou adresu (předchozí operace nás již nezajímají), ale není těžké vymyslet vstup, na kterém i tak musíme pro podstatnou část dotazů projít skoro všechny doposud provedené operace (takové vstupy jsme v úloze skutečně vyráběli). Tudy správná cesta nevede.

Pole všech adres

V prvním řešení bylo provedení operace okamžité, ale dotaz nám trval až $\mathcal{O}(N)$, pojďme dotazy zrychlit… třeba až na $\mathcal{O}(1)$. Možných IPv4 adres je 2^{32} (neboli něco přes 4 miliardy). Když si budeme pro každou adresu držet v poli jednobajtové `true` nebo `false` udávající, jestli je adresa aktuálně povolená nebo blokovaná, tak nám na to stačí přesně 4 GiB paměti, což dnes většinou není problém. A i jeden bajt je plýtvání, ve skutečnosti nám stačí jen jeden bit informace a spotřebovanou paměť tak můžeme snížit na 512 MiB (ale je potřeba umět přistupovat k jednotlivým bitům, typicky pomocí nějakých bitových triků s ANDy a ORy, záleží na jazyku).

Na začátku si celé pole vynulujeme, abychom reprezentovali, že všechny adresy jsou blokované. Pro operaci blokace pak přepíšeme správnou část pole nulami, pro operaci povolení jedničkami. Dotaz je pak jen vytažení správného prvku pole – jednička značí povolenou adresu, nula blokovanou.

U tohoto řešení je asi na první pohled jasné, že bude také vracet správné výsledky, ale s rychlostí na tom nebude nejlépe. Pokud si jako U označíme velikost univerza (v tomto případě 4 miliardy adres), tak jedna operace může trvat až čas $\mathcal{O}(U)$, celkově jsme tak skončili na časové složitosti $\mathcal{O}(NU)$. To na vyřešení všech vstupů této úlohy nestačilo (a například pro IPv6 adresy by taková složitost byla úplně nepoužitelná).

Líné stromy

Problémem předchozího řešení bylo to, že jsme zbytečně trávili čas s „tapetováním“ velkých částí paměti nulami nebo jedničkami. Přitom by nám pro některé části stačilo vědět „od X do Y jsou adresy zakázané“. Na takové věci se bude hodit reprezentovat IP adresy pomocí stromů.

IPv4 adresa rozepsaná na bity reprezentuje průchod binárním stromem o hloubce 32 – nechť třeba nulový bit znamená *doleva* a jedničkový *doprava*. Podsíť s prefixem délky K v takovém stromě je vlastně podstrom začínající vrcholem v hloubce K, ke kterému jsme došli pomocí prefixu této sítě.

Operace si ve stromě budeme reprezentovat jako vrcholy říkající „všechno pode mnou je blokováno/povoleno“. Musíme ale vyřešit, jak při dotazu na nějakou IP adresu poznat poslední operaci, která se této IP adresy týkala. To se dá udělat více přístupy, ale my si ukážeme řešení pomocí *líného stromu*.

Idea je vlastně triviální: vrcholy stromu si budeme vyrábět až ve chvíli, kdy budou potřeba, a nikdy si nedovolíme mít ve vrcholu operaci, která by byla přepsaná nějakou novější operací pod ní. Při dotazu pak díky tomu bude platit, že první operace, kterou při procházení stromu potkáme, je platná a můžeme ji rovnou vrátit jako odpověď.

V momentě, kdy budeme přidávat novou operaci, vydáme se stromem k vrcholu představujícímu podsíť této operace. Pokud cestou potkáme neexistující vrchol, tak ho založíme. Pokud potkáme vrchol s jinou operací, musíme tento jiný vrchol „rozhrnout“. Jednoduše zrušíme operaci v současném vrcholu, ale předtím zkopírujeme tuto operaci do obou jeho synů (které případně založíme, pokud ještě neexistují). Pak můžeme pokračovat v původním průchodu (který pravděpodobně do jednoho ze synů sestoupí a bude se tak opakovat podobná situace, dokud nedojdeme až k vrcholu, kde se chceme zastavit – je to něco jako když sněžný pluh rozhrnuje sníh na obě strany :)).

Jak rychlé takové řešení je? Každá operace i dotaz projde stromem jen jednou, hloubka stromu pro univerzum o velikosti U je $\log U$, takže celkový čas je $\mathcal{O}(N \log U)$. To již stačilo k vyřešení všech vstupů v řádu jednotek sekund. Na implementaci se můžete podívat v přiloženém programu.

Poznámka: Tento strom je nazývaný líný proto, že operace na intervalech ukládáme vždy co nejblíže ke kořeni a tím si ušetřujeme práci. Poctivě je propíšeme do nižších vrstev až ve chvíli, kdy je to nutné.

Program (C):
`http://ksp.mff.cuni.cz/viz/33-4-2.c`

Úlohu připravili: Michal Kodad, Martin Koreček, Jirka Sejkora, Jirka Setnička

33-4-3 Mraveniště na kopci

Nejprve si uvědomíme, že jednotliví mravenci jsou nezávislí (a taky určitě i silní, ale to je pro naši úlohu nepodstatné). Tedy když vyšleme nějakého mravence z mraveniště, tak pravděpodobnost toho, že skončí v nějakém vrcholu, nijak nezávisí na tom, kam došli předcházející mravenci (a ani kolik jich bylo).

Představme si, že známe pravděpodobnost p, se kterou mravenec dorazí do cíle. Pokud bychom vyslali jen tohoto jednoho mravence, bude střední hodnota počtu mravenců dorazivších do cíle právě p.

Dále využijeme linearity střední hodnoty: Vyšleme-li K/p mravenců, dorazí jich do cíle ve střední hodnotě přesně požadovaných $(K/p) \cdot p = K$. Protože vyslaných mravenců musí být celočíselně, vyšleme jich $\lceil K/p \rceil$ (nejbližší celé číslo $\geq K/p$).

Nyní se podívejme, jak spočítat onu pravděpodobnost p.

Využijeme dynamického programování. Pro každý vrchol grafu si spočítáme, jaká je pravděpodobnost toho, že jím mravenec projde. Mraveništěm projde určitě, tedy s pravděpodobností 1. Pro ostatní vrcholy sečteme pravděpodobnosti podle toho, odkud mravenec přijde. Předpokládejme, že z vrcholu V vede do aktuálního vrcholu hrana. Pravděpodobnost toho, že mravenec přijde po hraně z vrcholu V je p_V/s_V, kde p_V je pravděpodobnost průchodu mravence vrcholem V a s_V je počet hran, které z V vedou. Když tedy známe pravděpodobnosti pro všechny předky, zvládneme spočítat i pravděpodobnost aktuálního vrcholu.

Jak ale zařídit, abychom počítali vrcholy tak, že vždy budeme mít spočítány všechny předky? V případě, že cestičky jsou stromem, stačí udělat průchod do šířky či do hloubky z mraveniště. V obecném případě budeme hledat možné rozložení nadmořských výšek tak, aby všechny hrany vedly z kopce. Tedy můžeme najít topologické uspořádání grafu, o němž se můžete dočíst v kuchařce Grafy.[1] Jelikož graf neobsahuje cyklus, tak musí alespoň jedno existovat. V tomto pořadí pak už můžeme bez problémů spočítat pravděpodobnosti.

Pro naši úlohu ale stačí použít i jednodušší postup než hledání topologického uspořádání. Uděláme průchod do hloubky z cílového vrcholu, který bude procházet po hranách v opačném směru než mravenci. U každého vrcholu nejprve provedeme rekurzi do všech předků (v případě, že jsme je ještě nenavštívili) a pak spočteme pravděpodobnost pro aktuální vrchol. Každého předka jsme tedy buď spočítali již někdy dříve, nebo jsme ho spočetli teď rekurzí. Budeme mít tedy dostatek dat pro výpočet pravděpodobnosti aktuálního vrcholu.

Úlohu připravili: Jirka Kalvoda, Jirka Setnička

33-4-4 Prohledávání KSP webu

Úloha byla implementačně založená – nemuseli jste nic světoborného vymýšlet, ale spíše si osahat komunikaci s webem. Zdrojový kód a popsané řešení se vztahuje k řešení psanému v jazyce Python. Jiné jazyky můžou mít řešení této úlohy trochu lehčí či těžší nebo mít jiné záludnosti.

Řešení se bude skládat ze dvou částí. První část stáhne všechny stránky z webu, předpřipraví je do nějakého formátu, který bude příjemný na odpovídání dotazů, a uloží výstup do souboru. Druhá část programu bude už jen číst tento soubor a odpovídat na dotazy.

První krok je zjištění všech URL, ze kterých chceme stahovat. Při krátkém průzkumu našeho webu zjistíme, že URL má tento tvar: Začátek bude `/z/ulohy` nebo `/h/ulohy` (pro dvě různé kategorie), následovaný číslem ročníku a nakonec určením, jestli chceme zadání, řešení či komentáře spolu s číslem série. Nějaké nástroje dokonce umí stáhnout úplně vše z daného kořenové URL, ale to využít nepotřebujeme.

Dále dané stránky musíme stáhnout. Na toto využijeme knihovnu `requests` (standardní Pythoní knihovna je pro jednoduché použití spíše nevhodná a lehce se můžete něčím střelit do nohy, `requests` je lepší).

Nyní na každé stránce stačí najít `div` s identifikátorem `content`. Pro obecné parsování HTML bychom doporučili nějaký parser, například knihovnu `BeautifulSoup`, ale protože teď potřebujeme jen vyseknout `div` se správným identifikátorem ze stránek, které mají všechny stejnou strukturu, je jednodušší to v tomto případě udělat *regexem* (neboli regulárním výrazem). Dejte si však pozor, že obecné parsování HTML jen pomocí regexu napsat nejde[2] a nedoporučujeme se o to ani pokoušet.[3]

Víme, že chceme zahodit vše, co je před tagem `div` s identifikátorem `content` a poté zahodit vše od momentu, kde začíná `div` s identifikátorem `sidebar-wrapper`.

Nyní chceme odstranit HTML entity. Ty odstraníme tímto regexem `&.*?;` (otazník za hvězdičkou říká, že chceme hledat nehladově – najít nejkratší úsek splňující kritérium). Uvědomme si, že znaky `&<>` a další jsou speciální znaky HTML, takže pokud je chceme mít v textu, tak je musíme mít napsané HTML entitou. Nakonec dalším regexem `<.*?>` smažeme všechny tagy. Pozor, v Pythonu standardně tečka nebere znak nového řádku (newline), proto musíme tuto funkci explicitně zapnout nepovinným parametrem `flags`.

Poté již stačí procházet jednotlivá slova regexem `\w+` a házet si je třeba do slovníku.

První část našeho programu se spouští s parametrem `-ini`, protože tento krok děláme většinou jednou a pak již program pracuje „offline“.

Druhá část programu je již jednoduchá – načtu si předpřipravený soubor třeba do slovníku a odpovídám na dotazy. Celkový počet slov je vcelku malý – kolem 60 tisíc slov, takže se moc nevyplatí vymýšlet něco speciálního (třeba slova binárně vyhledávat).

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/33-4-4.py`

Úlohu připravili: Michal Kodad, Martin „Medvěd“ Mareš, Jirka Sejkora, Jirka Setnička

33-4-X1 Koření a recepty

Toky a řezy

Zopakujeme si část algoritmů a teorie okolo toků a řezů. Budeme se řídit podle kapitoly o tocích v Průvodci labyrintem algoritmů.[4]

Mějme nějakou síť s vrcholy V, orientovanými hranami E, nezápornými kapacitami hran $c(e)$, a konečně zdrojem z a stokem neboli spotřebičem s.

Pro množiny vrcholů A, B označíme množinu hran vedoucích z A do B jako $E(A,B) = \{(a,b) \in E \mid a \in A, b \in B\}$.

[1] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`
[2] regulární jazyky jsou v Chomského hierarchii jazyků slabší, než bezkontextové jazyky, do kterých lze řadit HTML
[3] `https://stackoverflow.com/a/1732454`
[4] `http://pruvodce.ucw.cz/`

Řez v síti budeme říkat každé množině hran $R \subseteq E$, jejichž odebrání přeruší všechny cesty ze zdroje do stoku. (Pokud v síti žádná taková cesta nebyla, i prázdná množina je řez.)

Pokud vrcholy beze zbytku rozdělíme na množiny A a B tak, aby zdroj ležel v A a stok v B, hrany z A do B budou tvořit řez. Každá cesta ze zdroje do stoku totiž někde musí přejít z A do B, takže obsahuje aspoň jednu hranu z $E(A,B)$. Takovým řezům říkáme *elementární řezy.*

Navíc platí, že *podmnožinou každého řezu je nějaký elementární řez.* To dokážeme snadno: Mějme nějaký řez F. Označme A množinu všech vrcholů dosažitelných ze zdroje v grafu $G - F$. Zřejmě $z \in A$ a $s \notin A$, jinak by totiž řez nepřeřízl všechny cesty ze z do s. Přitom všechny hrany elementárního řezu $E(A,B)$ musí ležet v F (jinak by množinu A šlo rozšířit).

Řezům můžeme také přiřazovat kapacity. Pro libovolnou podmnožinu hran $F \subseteq E$ (tedy i pro řez) položíme $c(F) := \sum_{e \in F} c(e)$. Speciálně se nám to bude hodit pro elementární řezy, tak zavedeme zkratku $c(A,B) := c(E(A,B))$.

Základní úlohou týkající se řezů je nalezení *minimálního řezu,* tedy řezu o nejmenší možné kapacitě. Všimněte si, že *alespoň jeden z minimálních řezů je elementární.* To proto, že každý minimální řez obsahuje jako podmnožinu nějaký elementární řez, jehož kapacita nemůže být větší (hrany mají nezápornou kapacitu).

Nyní mějme nějaký tok f. Definujme *průtok* přes libovolný elementární řez $E(A,B)$ tak, že tok po hranách z A do B započítáme kladně a tok v protisměru záporně:

$$f^{\Delta}(A,B) := \left(\sum_{e \in E(A,B)} f(e) \right) - \left(\sum_{e \in E(B,A)} f(e) \right).$$

Pak platí (důkazy najdete v Průvodci):

- Velikost toku $|f|$ měřená u zdroje je $f^{\Delta}(\{z\}, V \setminus \{z\})$.
- Velikost toku měřená u stoku je $f^{\Delta}(V \setminus \{s\}, \{s\})$.
- Je jedno, kterou definici velikosti si vybereme, protože $f^{\Delta}(A,B)$ vyjde stejně pro všechny elementární řezy $E(A,B)$.
- Proto je $|f| = f^{\Delta}(A,B) \leq c(A,B)$ pro všechny řezy. Tedy velikost každého toku je omezena kapacitou každého řezu.
- Pokud je pro jakýkoliv řez $f^{\Delta}(A,B) = c(A,B)$, pak tok f je maximální a řez $E(A,B)$ minimální.
- Nechť se Fordův-Fulkersonův algoritmus zastaví a vydá jako výsledek nějaký tok f. Označme A množinu vrcholů dosažitelných ze zdroje po zlepšujících cestách a B její doplněk. Pak je $f^{\Delta}(A,B) = c(A,B)$. Našli jsme tedy řez stejně velký jako nalezený tok, takže tok je maximální a řez minimální.

Koření a recepty

Vztahu mezi toky a řezy teď využijeme k řešení úlohy. Nejprve vytvoříme vhodnou síť, složenou ze čtyř vrstev: zdroj, vrstva koření (každý druh koření zde bude mít svůj vrchol), vrstva receptů (vrcholy pro jednotlivé recepty), stok. Přitom:

- Ze zdroje povedou hrany do všech koření. Těm budeme říkat *hrany koření.*
- Z každého koření povedou hrany do všech receptů vyžadujících toto koření. To jsou *hrany závislostí.*
- Z receptů povedou hrany do stoku. To jsou *hrany receptů.*

Než nastavíme kapacity, uvědomíme si následující vlastnost grafu: Nechť vybereme nějaká koření a nějaké recepty. Sestrojíme množinu hran R tak, že do ní dáme hrany *vybraných* koření a *nevybraných* receptů. Nahlédneme, že vybraná koření jsou dostatečná k uvaření vybraných receptů právě tehdy, když R je řez. To proto, že kdykoliv by vybraný recept závisel na nevybraném koření, vedla by nějaká nepřeříznutá cesta ze zdroje do stoku (totiž ze zdroje hranou koření, pak hranou závislosti a hranou receptu do stoku). Naopak jakákoliv nepřeříznutá cesta odpovídá porušené závislosti. Tady se hodí všimnout si, že kvůli orientaci hran jsou všechny cesty tvořené třemi hranami.

Nyní nastavíme kapacity takto:

- Hrany koření ohodnotíme náklady na nákup koření.
- Hrany receptů ohodnotíme ziskem z receptů.
- Hrany závislostí dostanou kapacitu $+\infty$.

Minimální řez určitě nebude mít nekonečnou kapacitu (protože všechny hrany koření tvoří řez o konečné kapacitě), takže bude používat jenom hrany koření a receptů. Minimální řezy tedy odpovídají výběru koření a receptů podle požadavků úlohy. Jejich kapacita minimalizuje výraz

$$\text{(cena použitých koření)} + \text{(cena nepoužitých receptů)}.$$

Jelikož součet cen všech receptů je konstanta, je to ekvivalentní s minimalizací

$$\text{(cena použitých koření)} - \text{(cena použitých receptů)}.$$

A jelikož $\min(-x,-y) = -\max(x,y)$, je to ekvivalentní s maximalizací

$$\text{(cena použitých receptů)} - \text{(cena použitých koření)},$$

což je přesně cílem úlohy.

Zbývá dořešit, jak minimální řez nalezneme. Nejdřív najdeme maximální tok. Jelikož Fordův-Fulkersonův algoritmus je pro obecné kapacity velmi pomalý, použijeme Dinicův algoritmus. Na nalezený tok pak spustíme Fordův-Fulkersonův algoritmus. Ten se nutně zastaví po prvním průchodu – maximální tok už nejde zlepšit. I tak nám řekne něco užitečlného: množina vrcholů, do kterých se dostal, určuje jeden z minimálních řezů.

Nakonec rozebereme složitost. Označme K počet druhů koření, R počet receptů a Z počet závislostí. Náš graf má $n = K + R + 2$ vrcholů a $m = K + R + Z$ hran. Dinicův algoritmus běží v čase $\mathcal{O}(n^2 m)$. Jeden průchod F-F algoritmu stojí $\mathcal{O}(m)$, což je asymptoticky méně. Časová složitost celého algoritmu je tedy $\mathcal{O}(n^2 m) = \mathcal{O}((K+R)^2(K+R+Z))$. Pokud samostatně ošetříme koření a recepty, které se neúčastní žádné závislosti, bude platit $K + R \in \mathcal{O}(Z)$, takže odhad složitosti můžeme zjednodušit na $\mathcal{O}((K+R)^2 \cdot Z)$. Paměti nám stačí lineárně s velikostí grafu, tedy $\mathcal{O}(K+R+Z)$.

Úlohu připravili: Martin „Medvěd“ Mareš, Tom Sláma

33-4-S Raytracing

Odkazy na referenční implementace úkolů ze všech dílů seriálu naleznete v řešení posledního dílu seriálu.[5]

[5] `http://ksp.mff.cuni.cz/viz/33-5-S/reseni`

Výsledková listina čtvrté série třicátého třetího ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*4-1*	*4-2*	*4-3*	*4-4*	*4-S*	*4-X1*	*série*	*KSP-X*	*celkem*
0.					12	10	11	12	15	10	60,0	40,0	240,0
1.	Kristýna Petrlíková	SPŠJičín	3	14	12	10	10,5	12	15		59,5	0,0	240,5
2.	Jan Adámek	GKepleraPH	4	9	9,5	10	11	12	15	0	57,5	18,0	218,5
3.	Jiří Kvapil	GTomkovaOL	3	18	0,5	10	10	12	15		47,5	3,0	215,5
4.	Daniel Skýpala	GTomkovaOL	3	19		10	10,5	12	15		47,5	5,0	215,0
5.	Ondřej Skácel	GTomkovaOL	2	4	12	10	10,5	12	15		59,5	0,0	209,5
6.	Filip Hejsek	GPísnickáPH	4	6	11	10	11	12			44,0	0,0	195,5
7.	Robert Jaworski	GÚstavníPH	3	6		10	10,5	12	15		47,5	2,0	191,5
8.	Viktor Fukala	GKepleraPH	4	8	12	10	11	12			45,0	19,0	171,5
9.	Jan Kotovský	GPísnickáPH	2	5		10	5,5	12	15		42,5	0,0	170,5
10.	Lukáš Veškrna	GKepleraPH	3	4	6	4	9	12	15		46,0	4,0	169,0
11.	Vít Skalický	GPísnickáPH	3	18	0	10		12	14,5		36,5	0,0	160,5
12.	Jakub Surga	ParkLane	3	4		10	10,5	12			32,5	0,0	154,5
13.	Jakub Ondroušek	GTomkovaOL	1	4		4	10	12	15		41,0	0,0	145,5
14.–15.	Janek Hlavatý	GJirsíkaČB	2	8		1	5,5	12	12,5		31,0	0,0	133,0
	Eliška Macáková	CENADA BA	1	3							0,0	20,0	133,0
16.	Patrik Herman	GTomkovaOL	2	5		4	5	12	6,5		27,5	0,0	114,5
17.	Matej Štencel	GPošKošice	4	6		10		12	15		37,0	0,0	114,0
18.	Dominik Farhan	GMikulášPL	4	8		10	6	12			28,0	0,0	113,0
19.	Vladimír Chudý	G Chrudim	4	18							0,0	4,5	110,0
20.	Ondřej Sladký	GMikulášPL	4	11			10			3	10,0	22,0	91,0
21.	Adam Kolník	SSŠVTPraha	2	4	11	10	9				30,0	0,0	88,0
22.	Václav Janáček	GJarošeBO	4	6							0,0	16,0	80,0
23.	Prokop Randáček	GFXŠaldyLI	2	5		10			15		25,0	0,0	78,0
24.	Pavel Jordán	GPOA Znojmo	2	2							0,0	0,0	66,0
25.	Šimon Genčur	GBBr	1	6		1	10,5	12			23,5	0,0	64,5
26.	Klára Grinerová	GZborovPH	4	4		1	6				7,0	0,0	34,5
27.	David Holas	SPŠEMasLI	1	1							0,0	1,0	32,5
28.	Martin Havelka	Gym Třeboň	3	3							0,0	0,0	28,0
29.	Robert Gemrot	GKomHavíř	4	4		10	5,5	12			27,5	0,0	27,5
30.	Petr Šícho	GKepleraPH	3	2	8	7		9			24,0	0,0	24,0
31.	Jiří Bartošík	SUHr	3	2							0,0	0,0	22,0
32.	Albert Kučera	GNadŠtolPH	4	4							0,0	0,0	21,0
33.	Kristýna Umlaufová	SPŠOstrov	4	2							0,0	0,0	20,0
34.	Jáchym Tuma	G FrýdlNOs	0	2							0,0	0,0	19,0
35.	Andrej Thomas Dobrev	GJHroncaBA	4	1							0,0	0,0	18,0
36.	Kryštof Maxera	GJírovcČB	0	1			5	12			17,0	0,0	17,0
37.	Michal Žáček	MensaG	4	1							0,0	0,0	15,0
38.	Daniel Šoltýs	GTřeKošice	3	3		4					4,0	0,0	14,0
39.	Adam Húšťava	EupSchoolLux	3	4							0,0	0,0	13,0
40.	Tomáš Kašpárek	G FrýdlNOs	3	1							0,0	0,0	12,0
41.–46.	Vojtěch Březina	GCoubTábor	4	4							0,0	0,0	10,0
	Petr Filip	GLovosice	2	1							0,0	0,0	10,0
	Vojtech Gadurek	PORGPha	4	1							0,0	0,0	10,0
	Štěpán Kovář	GNadKavaPH	4	1							0,0	0,0	10,0
	Michal Pavlíček	MendelGOP	3	1							0,0	0,0	10,0
	Filip Úradník	GyMimoň	4	1							0,0	0,0	10,0
47.	Jan Ráček	SPŠEMasLI	1	1							0,0	0,0	7,0
48.–49.	Bohumil Kulvejt	G Sokolov	3	1							0,0	0,0	6,0
	Josef Malý	GPísnickáPH	2	1							0,0	0,0	6,0
50.	Veronika Jůzková	MensaG	3	2		1					1,0	0,0	3,0
51.	Matěj Strnad	ZŠRiegraSM	0	1							0,0	0,0	2,0

Bonusové úlohy označené „X“ mají svou vlastní výsledkovou listinu a nepočítají se do normálního bodování ročníku.

Výsledková listina KSP-X po čtvrté sérii třicátého třetího ročníku

	řešitel	*škola*	*ročník*	*sérií*	*1-X1*	*2-X1*	*3-X1*	*4-X1*	*celkem*
0.					10	10	10	10	40,0
1.	Ondřej Sladký	GMikulášPL	4	11	9		10	3	22,0
2.	Eliška Macáková	CENADA BA	1	3	9	1	10		20,0
3.	Viktor Fukala	GKepleraPH	4	8	9	10			19,0
4.	Jan Adámek	GKepleraPH	4	9	7	6	5	0	18,0
5.	Václav Janáček	GJarošeBO	4	6	8		8		16,0
6.	Daniel Skýpala	GTomkovaOL	3	19	4	1	0		5,0
7.	Vladimír Chudý	G Chrudim	4	18	4,5				4,5
8.	Lukáš Veškrna	GKepleraPH	3	4	4				4,0
9.	Jiří Kvapil	GTomkovaOL	3	18	2	1			3,0
10.	Robert Jaworski	GÚstavníPH	3	6	1	1			2,0
11.	David Holas	SPŠEMasLI	1	1			1		1,0

Bonusové úlohy z jednotlivých sérií se nepočítají do bodování ročníku. Mají svou vlastní výsledkovou listinu a za jejich úspěšné vyřešení (alespoň polovina bodů za úlohu) udělujeme speciální odměny.