Na riešenie tejto úlohy použijeme jednoduchý algoritmus na prehľadávanie grafu do hĺbky. Počas jeho behu budeme postupne tvoriť výsledné poradie činností. A to tak, že do čiastočne zostaveného zoznamu budeme činnosti pridávať nakoniec. Začneme s prázdnym zoznamom činností. Vrchol, ktorý sa už bude nachádzať vo výslednom poradí, si označíme ako „spracovaný“. Ďalej si pre každý vrchol budeme počas behu algoritmu pamätať, či sme ho už niekedy navštívili. Naviac si k nemu poznamenáme, z ktorého vrcholu sme sa doňho prvýkrát dostali.

Graf začneme prehľadávať z ľubovoľného vrcholu. Pozrieme sa na orientované hrany, ktoré z neho vedú k ďalším vrcholom. Vyberieme sa po ľubovoľnej hrane, ktorá nevedie do už spracovaného vrcholu.

Ak sme sa ocitli vo vrchole, v ktorom sme už raz boli, znamená to, že sme prešli po orientovaných hranách, ktoré tvoria kružnicu z nespracovaných vrcholov. Teda takýto zoznam činností tvorí kruhovú závislosť a tú usporiadať nejde. Prehľadávanie v tomto prípade ukončíme a oznámime neexistenciu riešenie.

Ak sme sa dostali do vrcholu, v ktorom sme ešte neboli, pozrieme sa opäť na susedov, do ktorých sa vieme dostať. Opäť si niektorého vyberieme a takto pokračujeme, až kým sa nedostaneme do vrcholu, z ktorého sa nedá pokračovať ďalej (a to už preto, že z neho nevedie žiadna hrana alebo preto, že všetci susedia sú už označení ako spracovaní). V takomto prípade sme sa dostali do vrcholu reprezentujúci činnosť, ktorá nemá žiadne nespracované závislosti. Keby mal nejaké nespracované závislosti, tak by musela existovať hrana z aktuálneho vrcholu niekam. Avšak neexistuje, a teda sme našli činnosť, ktorá bude mať určite splnené všetky závislosti (ak nejaké má) v už zostavenom výstupnom poradí činností.

Túto činnosť pridáme do aktuálne zostavujúceho sa poradia nakoniec za všetky už spracované činnosti. Vrchol, ktorým je činnosť reprezentovaná, sa v tomto momente stane spracovaným. Potom sa vrátime späť do vrcholu, z ktorého sme sa prvýkrát dostali do aktuálneho (ideme proti smeru orientácie hrany), a pokračujeme ďalej v prehľadávaní z toho vrcholu.

Ak skončíme prehľadávanie, tak graf buď bude celý prejdený, alebo v ňom nájdeme cyklus (a usporiadanie neexistuje), alebo ostanú v grafe ešte neprehľadané vrcholy. To, že v grafe ostanú neprehľadané vrcholy, môže nastať aj keď je graf súvislý. Napr. vtedy, ak si zvolíme začiatočný vrchol, z ktorého nevedú žiadne orientované hrany.

V takomto prípade, keď sme ešte nenavštívili všetky vrcholy (a teda výstupné poradie ešte nie je úplné), musíme spustiť prehľadávanie opäť z ďalšieho ešte nenavštíveného vrcholu. Máme stále zaručené, že činnosti, ktoré sú už vo výslednom poradí, nebudú závislé na nespracovaných činnostiach (nenavštívených vrcholoch). Ináč by viedla z nich hrana a pri prehľadávaní by sme ju spracovali.

Prehľadávanie budeme spúšťať vždy z niektorého nenavštíveného vrcholu, až kým nebudú všetky vrcholy navštívené a spracované. Na konci, keď už bude každý vrchol označený ako navštívený, bude zároveň každá činnosť vo výstupnom poradí, a teda už budeme mať hotové výsledné poradie.

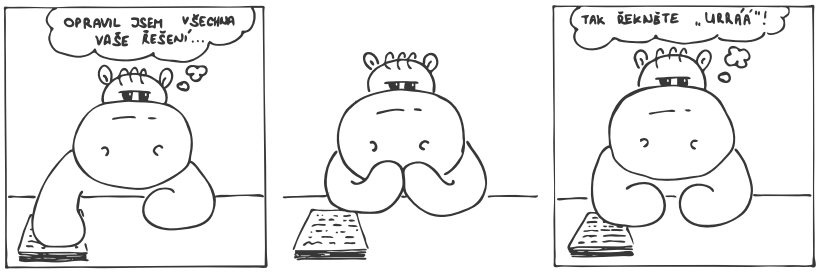
Keďže vrcholov je konečný počet a každý vrchol navštívime maximálne toľkokrát, koľko má susedov (z každého suseda sa doňho späť vrátime), a naviac každou hranou prejdeme maximálne dvakrát (raz v smere orientácie a raz proti smeru, keď sa budeme vracať), tak sa tento algoritmus určite raz zastaví. Z toho aj rovno vypozorujeme, že časová zložitosť algoritmu bude lineárna od počtu hrán a keďže navštívime každý vrchol, tak aj lineárna od počtu vrcholov.

Pre jednoduchosť si zadaný graf činností a ich závislostí budeme reprezentovať nezápornými celými číslami. V pamäti budeme mať pre každý vrchol uložený zoznam jeho susedov. Okrem toho si počas behu algoritmu budeme musieť pamätať, či sme daný vrchol už navštívili (a odkiaľ prvýkrát) a či už je vo výslednom poradí spracovaný. Čiže pamäťová zložitosť bude lineárna od počtu vrcholov a hrán. Výstup algoritmu bude tvoriť usporiadanie jednotlivých vrcholov, a teda nám pamäťovú zložitosť nezmení.

Ná záver malá poznámka. Problém popísaný v tejto úlohe sa označuje aj ako topologické usporiadanie orientovaného grafu a môžete sa o ňom dočítať aj v našej grafovej kuchárke.[5] V nej nájdete aj ďalší alternatívny algoritmus, ktorý rieši tento problém taktiež v lineárnom čase od veľkosti zadaného grafu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z4-6.py`

Pali Rohár



[5] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Chcete-li s námi komunikovat šifrovaně a bezpečně, můžete si ověřit náš HTTPS *certifikát – jeho SHA1 fingerprint je:* `0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3`.

Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

27. ročník | **KSP-Z** | **Květen 2015**

Skončila poslední, čtvrtá, série začátečnické kategorie KSP a my vám tedy naposled v tomto školním roce přinášíme autorská řešení. Doufáme, že se vám úlohy líbily. Pokud byste měli k čemukoliv otázky, třeba proč vaše řešení fungovalo či naopak nefungovalo, neváhejte se zeptat na našem fóru. Stejně tak nám můžete napsat email na adresu *ksp@mff.cuni.cz*.

Jsme rádi, že se vás do této série pustilo tolik. Gratulujeme všem, kdo získali nějaké body! Těm, kdo nezískali plný počet, doporučujeme si uvedená řešení projít a poučit se z nich. A i pokud jste úlohu vyřešili na plný počet bodů, můžete se pročtením řešení podívat na problém třeba z jiného úhlu pohledu.

Řešení čtvrté série začátečnické kategorie 27. ročníku KSP

27-Z4-1 Záhada Pražského orloje

Napsání programu pro tuto úlohu nebylo vůbec těžké, jak se můžete přesvědčit ve vzorovém řešení, nicméně bylo nutné uvědomit si jednu základní myšlenku.

Problém si můžeme představit tak, že naše dvě kolečka namočíme do barvy a uděláme s nimi stopy na papíře. Takto nám vzniknou dvě úsečky dlouhé jako obvody jednotlivých ozubených kol (délka obvodu, protože jsme s každým kolečkem otočili dokola). V této představě by náš problém byl jako vyskládání několika úseček délky obvodu prvního kolečka do jedné čáry a úseček délky obvodu druhého kolečka do druhé tak, aby obě čáry měly stejnou délku. Po chvíli bádání můžeme nahlédnout, že tento problém vyřeší nejmenší společný násobek daných dvou délek.

I řešení našeho původního problému je nejmenší společný násobek počtu zubů našich koleček. To proto, že se obě kolečka otočí o stejný počet zubů za jednotku času a pokaždé, kdy se kolečka potkají, se každé z nich otočí o celý počet otáček. Tedy když se potkají, tak první kolečko udělalo k svých otoček, druhé pak ℓ svých otoček. Pro předsavu třeba předpokládejme otočení o jeden zub za jednu sekundu. Každý čas setkání bude násobek počtu otočení a počtu zubů (tj. perioda) příslušného kolečka a toto pro obě kolečka bude stejné, tedy:

$$k \cdot \text{počet zubů prvního} = \ell \cdot \text{počet zubů druhého}.$$

Čas jejich prvního setkání nastane pro nejmenší možná ℓ a k a bude to nejmenší společný násobek obou period.

Jak spočteme nejmenší společný násobek? Pravděpodobně všichni známe rozklad na součin prvočísel, což ale v počítači není tak jednoduché a existuje mnohem rychlejší cesta. Ta vede přes Euklidův algoritmus zjištění největšího společného dělitele a vztah největšího společného dělitele (nsd) a nejmenšího společného násobku (nsn):

$$x \cdot y = \text{nsn}(x, y) \cdot \text{nsd}(x, y).$$

Tento vztah si můžete rozmyslet například právě díky zmíněnému prvočíselnému rozkladu.

Euklidův algoritmus funguje tak, že opakovaně odečítá od většího z čísel to menší, než se obě vyrovnají. Proč přesně funguje a jak ho zrychlit, se dozvíte v naší kuchařce o teorii čísel.[1] Zde prozradíme pouze to, že jeho časová složitost je $\mathcal{O}(x + y)$ a u zrychlené verze z kuchařky dokonce $\mathcal{O}(\log \min(x, y))$.

Celé řešení tedy načte N dvojic, pro každou dvojici spočítá nejmenší společný násobek a vypíše ho na výstup. Euklidův algoritmus tedy spouštíme N-krát, pro každou dvojici jednou. Paměti zabereme pouze konstantně, protože můžeme zpracovávat dotazy postupně, aniž bychom si je nejprve všechny načetli.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z4-1.py`

Martin Šerý & Štěpán Hojdar

27-Z4-2 Unavení u oken

Vida, rozsvícené okno. Jak zjistíme, do jaké souvislé svítící oblasti patří?

Začneme rozsvíceným oknem a prozkoumáme jeho sousedy. Pokud jsou také rozsvícení, započítáme je do oblasti a prozkoumáme i jejich sousedy. Jsou-li rovněž rozsvícení, zase je přidáme do oblasti a tak dále. Nechceme ovšem jedno okno započítat vícekrát, takže ledva nějaké započítáme, hned ho zhasneme.

Jak ale zařídit, abychom se v sousedech sousedů sousedů (atd.) neztratili? Pořídíme si *frontu,* v níž budeme skladovat všechna políčka, která jsme objevili, ale dosud jsme nezpracovali jejich sousedy. (Pokud se s frontou ještě neznáte, můžete si představit, že je to nějaké pole, ve kterém se nové prvky přidávají na konec a staré se odebírají ze začátku.)

Na počátku bude ve frontě jen to první rozsvícené okno. V každém kroku pak odebereme jedno okno z fronty a podíváme se na jeho sousedy. Je-li kterýkoliv z nich rozsvícený, zhasneme ho a přidáme do fronty. To opakujeme, dokud se fronta nevyprázdní.

Snadno si všimneme, že takto projdeme celou svítící oblast a bude nám to trvat řádově tolik času, kolik oken v oblasti leží. (Mimochodem, tomuto postupu se říká *prohledávání do šířky* a hodí se na ledacos dalšího.)

[1] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Jednu oblast tedy najít umíme. Zbývá domyslet, jak najít všechny. Budeme postupně procházet všechna okna a kdykoliv najdeme nějaké rozsvícené, prohledáme a zhasneme celou jeho oblast. Pak pokračujeme v hledání dalšího rozsvíceného okna atd.

Kolik času nám to celkem zabere? Hledání rozsvícených oken samo o sobě sáhne na každé okno právě jednou. Prohledávání všech oblastí dohromady sáhnou na každé okno nejvýše jednou (jedno okno nemůže ležet ve více oblastech současně). Náš program tedy má lineární časovou složitost s počtem všech oken.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z4-2.py`

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z4-2.c`

Martin Mareš & Jakub Marošek

27-Z4-3 Běžkaři v Praze!

Na tuto úlohu asi není žádný chytřejší postup, stačí si celou situaci odsimulovat. Nebudeme si ale posouvat figurkami závodníků po virtuálním okruhu, půjdeme na to chytřeji.

Mějme dva závodníky, kteří běží za sebou, a jejich rychlosti v_1 a v_2. Jediný spolehlivý způsob, jak poznat, jestli se na trati potkají, je spočítat časy, kdy by doběhli nezávisle (prozatím jako desetinná čísla v hodinách), a ty porovnat. Rovnice časů vypadají takto:

$$t_1 = \frac{S}{v_1} \qquad t_2 = \frac{1}{60} + \frac{S}{v_2}.$$

Budeme si udržovat seznam závodníků, u kterých neznáme čas. Jak toto udělat pohodlně najdete ve zdrojáku v Pythonu, jak to udělat rychle ve zdrojáku v C++. Vždy vezmeme prvního, který určitě závod dokončí, a vyhodíme jej ze seznamu. Následně přeskočíme všechny, co prvního doběhnou ($t_1 \geq t_2$, nezapomeňte ale na různou velikost zpoždění na startu). Podobně projdeme celý seznam, a celou proceduru opakujeme dokud nezměříme všechny závodníky.

V poznámce jsme psali, že se dá vyhnout počítání s desetinnými čísly – přesnost počítání s nimi není nekonečná, a pokud by se dva závodníci teoreticky potkali až těsně před cílovou páskou, mohlo by na přesnosti záležet. Také s nimi počítače počítají zpravidla o něco pomaleji. Pokud to lze, je dobré se jim vyhnout. Pojďme se podívat, jak na to.

Napišme si nerovnici pro dva závodníky, druhého zpožděného o m minut:

$$t_1 = \frac{S}{v_1} \;\geq\; \frac{m}{60} + \frac{S}{v_2} = t_2.$$

A trochu si ji upravme vynásobením $60 \cdot v_1 v_2$ (kladné číslo):

$$60 \cdot S\, v_2 \geq m\, v_1 v_2 + 60 \cdot S\, v_1.$$

A protože pouze násobíme a všechna čísla jsou celá, stačí nám už počítat s celými čísly. Pokud je tato nerovnice splněna, druhý závodník prvního doběhl.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/27-Z4-3.py`

Program (C++11):
`http://ksp.mff.cuni.cz/viz/27-Z4-3.cpp`

Ondra Hlavatý

27-Z4-4 Koňské skoky

Pro začátek zkusme vyřešit stejnou úlohu, akorát jenom s jedním koněm. Tedy dostaneme políčko, kde kůň začíná, a políčko, kam má doskákat. Naším úkolem je potom zjistit, na kolik skoků se tam dokáže dostat.

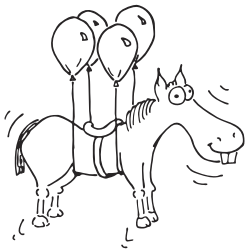
To není nic těžkého - použijeme jednoduchý algoritmus prohledávání do šířky, který je podrobně vysvětlen v naší grafové kuchařce.[2]

Jenže jak ho použít, když algoritmus povídá o grafech, ale my tu máme šachovnici? Prostě si podle šachovnice graf vytvoříme. Vrcholy grafu budou políčka šachovnice a hrana povede mezi dvěma políčky právě tehdy, když z jednoho na druhé může skočit kůň.

Prohledávání do šířky v našem případě dělá to, že nejprve zapíše jedničku do všech políček, kam se kůň může dostat jedním skokem ze začátečního políčka, potom dvojku na nenavštívená políčka, kam se může dostat z políčka s jedničkou, a tak dále, dokud nemáme na každém políčku napsáno, na kolik skoků se tam dokážeme dostat. Na konci si už jenom přečteme, v kolikátém kroku jsme byli v cíli.

Prohledávání do šířky skončí v čase $\mathcal{O}(N^2)$, protože šachovnice má N^2 políček a přibližně $4N^2$ hran mezi nimi.

Tak to by bylo. V zadané úloze je ale problém výrazně složitější v tom, že není jasné, který kůň má jít do kterého cíle. S tím se vypořádáme tak, že prostě vyzkoušíme všechny možnosti.



Zjistíme pro každou dvojici startovního a cílového políčka, jak dlouhá je mezi nimi cesta. Užitečné je, že prohledávání do šířky nám pro nějaké startovní políčko řekne vzdálenosti do všech ostatních políček, takže nám bude stačit pustit ho pětkrát (pro každé startovní políčko) a pokaždé si poznamenat vzdálenosti do všech pěti možných cílů.

Máme tedy tabulku 5×5 se vzdálenostmi mezi starty a cíli. Teď už stačí jenom vyzkoušet všechny možnosti, jak je spárovat. Možností je $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Tu nejlepší by šlo najít i s tužkou a papírem. Jako správní programátoři jsme ale líní, a tak to naprogramujeme.

Můžeme si třeba vygenerovat seznam všech permutací čísel od 1 do 5, neboli seznam způsobů, jak tato čísla seřadit. Každá permutace potom bude popisovat, jak přiřazovat starty k cílům. Pak už stačí jenom pro každou permutaci sečíst příslušné hodnoty z tabulky a najít nejnižší součet.

Ukážeme rekurzivní algoritmus, jak všechny permutace čísel od 1 do 5 vygenerovat: Vytvoříme funkci p, která dostane prvních několik pozic permutace pevně zadaných a vypíše všechny možnosti, jak pokračovat. Pokud nám zbyla jediná pozice, kam něco dát, a tedy i jediná hodnota, kterou tam dát, tak tuto jedinou permutaci zaznamenáme.

[2] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Jinak vyzkoušíme všechny možnosti, jak pokračovat na další pozici, a zbytek si vždy objednáme pomocí naší funkce p.

Stačí zavolat funkci p a nedat jí žádné omezení na to, co má být na začátku, a dostaneme všechny permutace.

Dodejme ještě, že se můžeme obejít bez generování permutací tak, že podobným rekurzivním způsobem budeme hledat rovnou minimální součet délek cest.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z4-4.c`

Martin Španěl

27-Z4-5 Poškolní trest

Nejprve se zamyslíme, jak bychom postupovali se skutečnou hromádkou papírů, a teprve poté toto řešení převedeme do řeči počítačů.

Všimněme si, že vytvářená hromádka papírů musí po celou dobu práce být správně uspořádaná. Tím myslíme to, že každý papír v ní má číslo o jedna větší než předchozí. Pokud by to neplatilo, pak v hromádce jsou dva papíry v sestupném pořadí (větší číslo před menším) nebo „díra“ (např. sousedí dvojice 1, 5). Ani jednu z těchto chyb už přidáváním dalších papírů nemůžeme odstranit, tedy ani na konci taková hromádka nebude správně uspořádaná.

Z toho už je vidět, jak musíme postupovat při třídění. Papír můžeme přidat na konec výstupní hromádky pouze tehdy, když má číslo o jedna větší než dosavadní koncový papír. Podobně pro přidávání na začátek, kde naopak musí být přidávané číslo o jedna menší. Zároveň si snadno rozmyslíte, že takovýmto přidáním nemůžeme nic pokazit. Z toho už je vidět výsledný postup: v každém kroku se podíváme na oba krajní papíry vstupní hromádky. Pokud některý z nich lze přidat na výstupní hromádku v souladu s popsanými pravidly, učiníme tak. Pokud je možné přidat oba, vybereme si libovolný (ten druhý můžeme přidat v příštím kroku).

Pokud se někdy dostaneme do situace, kdy ani jeden ze dvou krajních papírů nejde přidat na výstupní hromádku, pak setřídění není možné.

To vše má ovšem jeden háček. Popsali jsme si, jak postupovat, když už na výstupní hromádce nějaké papíry jsou. Ale jak začít? Například posloupnost papírů $3, 1, 2, 4, 5$ lze setřídit, pokud jako první na výstupní hromádku položíme číslo 5, ale nejde, pokud začneme trojkou (v takovém případě se zastavíme hned ve druhém kroku). Nejjednodušším řešením je prostě vyzkoušet obě možnosti. Nejprve začít prvním papírem, a pokud se to nepovede, znovu si vzít původní posloupnost a zkusit to z opačného konce. Tím se celý postup zpomalí jen dvakrát, což je ve světě algoritmů obvykle zanedbatelné.

Nyní se zamysleme, jak z toho všeho vyrobit program. Začněme tím, jak reprezentovat vstupní a výstupní hromádku. Vstupní si určitě musíme na začátku celou načíst (např. do pole), abychom se mohli dívat na oba konce. Poté z ní chceme odebírat papíry. To bychom mohli dělat tak, že prostě budeme z pole mazat prvky (např. v Pythonu příkazem `del pole[0]`), ale to je pomalé. Jediný způsob, jak smazat prvek ze začátku pole, je posunout všechny ostatní o jednu pozici níž. K tomu je potřeba řádově tolik operací, jako délka pole, tedy $\mathcal{O}(n)$, což je zbytečně moc.

Snadno jde toho samého dosáhnout v konstantním čase, a to hned dvěma způsoby. Prvním je použít místo pole strukturu zvanou spojový seznam, o které si můžete přečíst v naší základní kuchařce.[3] Z (obousměrného) spojového seznamu pak dokážeme odstraňovat prvky z libovolného konce v konstantním čase. Například v Pythonu můžete použít implementaci spojových seznamů ve třídě collections.deque.[4]

Pokud byste si je ale museli implementovat sami (např. v C), je to spousta práce, a v takovém případě je jednodušší druhé řešení: vůbec z pole nic nemazat. Namísto toho si budeme jen pamatovat, kterou pozici v poli aktuálně považujeme za první a poslední, a vždy pracovat pouze s touto částí pole. Ostatní prvky tam pořád budou, ale program je bude ignorovat. To je trik, který se při programování často hodí: někdy není třeba doopravdy měnit nějaká data, stačí změnit způsob, jakým se na ně díváme. Tuto variantu najdete i ve vzorovém programu.

Reprezentace výstupní hromádky je ještě jednodušší. Všimněme si, že nikdy nepracujeme s žádným jiným než prvním či posledním jejím prvkem. Stačí nám proto pamatovat si místo celé hromádky dvojici čísel: aktuální první a poslední prvek. Pokud chceme přidat papír na některý konec, prostě příslušné krajní číslo přepíšeme. To původní už stejně nikdy nebudeme potřebovat.

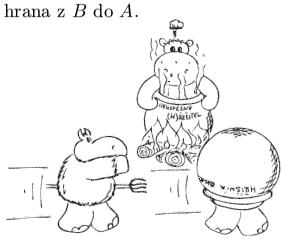
Každý krok třídění zvládneme v konstantním čase, tedy celý algoritmus bude mít lineární časovou složitost.

Program (C):
`http://ksp.mff.cuni.cz/viz/27-Z4-5.c`

Filip Štědronský

27-Z4-6 Příprava grilovačky

Cieľom tejto úlohy bolo usporiadať zadané činnosti podľa stanoveného pravidla. Každá činnosť, ktorá závisí od inej, sa musí vykonať až potom, čo sa vykonajú činnosti, na ktorých závisí. Pre jednoduchosť si jednotlivé činnosti a závislosti zakreslíme do grafu. Vrcholy grafu budú tvoriť činnosti a hrany budú šípky určujúce závislosť. Keďže šípky, ktoré vychádzajú od činností majú definovaný smer (podľa závislosti medzi činnosťami), tak aj hrany v grafe budú orientované v rovnakom smere. Teda, ak činnosť A musí byť vykonaná pred činnosťou B, potom v grafe bude existovať orientovaná hrana z B do A.



Na začiatok jednoduché pozorovania. Graf, ktorý tvorí závislosť činností, nemusí byť súvislý. To nastane vtedy, ak budeme mať nejakú množinu činností, ktorá nezávisí na ostatných a ani žiadna iná činnosť nezávisí na nej. Ak v grafe bude existovať orientovaný cyklus, znamená to, že činnosti obsahujú kruhovú závislosť (napr. A závisí na B, B na C a C na A). A takúto kruhovú závislosť nie je možné žiadnym spôsobom usporiadať.

[3] `http://ksp.mff.cuni.cz/viz/kucharky/uvodni`

[4] `https://docs.python.org/3/library/collections.html#collections.deque`