Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

32. ročník | **KSP-Z** | **Červen 2020**

Řešení čtvrté série začátečnické kategorie 32. ročníku KSP

32-Z4-1 Jednobarevné praní

Pro zaznamenání počtu triček jednotlivých barev v hromadě si zavedeme slovník. Při přidání trička na hromadu buď vytvoříme nový klíč s danou barvou, pokud jsme ji ještě nepotkali, a k němu přiřadíme hodnotu 1, nebo hodnotu u existujícího klíče zvýšíme o 1.

Dále potřebujeme dvě proměnné – jedna udává aktuální velikost hromady. Druhá ukazuje, jakou maximální výšku hromada zatím měla. Tyto dvě proměnné budeme porovnávat při každém přidání trička do hromady, aby v druhé proměnné byla vždy uložena maximální výška.

Začneme procházet vstup a pro každé písmeno ze vstupu inkrementujeme hodnotu u příslušného klíče ve slovníku nebo vytvoříme nový klíč. Pokud hodnota u nějakého klíče dosáhne čísla 5, tak se v hromadě nachází pět triček stejné barvy. Zkontrolujeme velikost hromady – pokud je její výška větší než maximální uložená výška, tak tuto hodnotu opravíme. Potom snížíme aktuální výšku o 5 a k danému klíči zapíšeme hodnotu 0.

Na konci vypíšeme maximální velikost hromady. Vše zvládneme spočítat na jeden průchod vstupními hodnotami – časová složitost proto bude $\mathcal{O}(N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z4-1.py`

Lucka Vomelová & Zuzka Urbanová

32-Z4-2 Hoří chleba?

Než se pustíme do tvorby algoritmu, povšimněme si tří důležitých věcí:

1. Prázdná políčka pro nás nejsou zajímavá a nemá smysl si je ukládat. Důležitá je pouze vzdálenost a tu vyčteme ze souřadnic.
2. Vstup je seřazený podle souřadnic, tedy sousední objekty (nepočítáme prázdná políčka) spolu sousedí i na vstupu.
3. Úloha bude fungovat shodně, pokud se nebude z trub šířit kouř, ale z detektorů „detekční paprsek“, přičemž detektor spustí poplach, jakmile jeho paprsek zabírá alespoň dvě trouby.

Teď k algoritmu. Nejjednodušší je samozřejmě dřevorubecká varianta s využitím poznatků 1 a 2. Všechny objekty si naskládáme za sebe do pole a budeme vteřinu po vteřině simulovat šíření kouře a kontrolovat, kolik se jej již dostalo ke každému detektoru. Nakonec detektory seřadíme podle času a vypíšeme.

Takový přístup je ale velmi neefektivní, dlouhé vstupy bychom zpracovávali hrozně dlouho. Můžeme využít poznatek 3 a tedy namísto simulace kouře pro každý detektor hledat dvě nejbližší trouby; čas spuštění poplachu pak odpovídá vzdálenosti k vzdálenější z nich. Stále jsme si moc nepomohli (pro vstup, kde na krajích chodby je vždy 1 trouba a zbytek je tvořen detektory, děláme $\mathcal{O}(N^2)$ kroků), ale otevřeli jsme si cestu k významné optimalizaci.

Hrozně moc času trávíme hledáním nejbližších trub. Nemohli bychom si je nějak předpočítat? Pořiďme si ke každému políčku s troubou nebo hlásičem ještě dvě informace a to, kde se nachází nejbližší trouba směrem nalevo a směrem napravo. To zvládneme spočítat jednoduchým průchodem zleva, resp. zprava (nezapomeňme, že nejlevější/nejpravější trouba už žádnou levější/pravější nemá).

Této pomůcky teď využijeme. Pro každý detektor si pořídíme až 3 dvojice kandidátů – trub, jež v něm spustí poplach. Jedná se o troubu nejblíže nalevo a další troubu nalevo od ní, obdobně pro směr vpravo a následně o troubu nejblíže vlevo a nejblíže vpravo. Musíme si ohlídat, že daná dvojice skutečně existuje, a také správně pracovat v případě, že na políčku se zkoumaným detektorem zároveň stojí trouba. Pak již jen vybereme dvojici, jejíž vzdálenější trouba je nejblíže zkoumanému detektoru. Ta nám též prozradí, za jak dlouho detektor sepne.

Zbývá již jen detektory seřadit dle časů. V kuchařce o třídění[1] se můžete dozvědět, že rychlé třídící algoritmy zvládnou seřadit posloupnost N čísel v čase $\mathcal{O}(N \log N)$ (v Pythonu zavoláním `sort` použijete právě takový algoritmus). Jelikož je třídění nejdelší operací celého algoritmu, tak celkově poběží algoritmus s časovou složitostí $\mathcal{O}(N \log N)$. Co se paměti týče, načetli jsme do ní celý vstup a ke každému políčku si připojili konstantně mnoho proměnných, takže spotřeba je lineární s N.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z4-2.py`

Vojta Káně

32-Z4-3 Esej do bloku

Nejdříve se podíváme na to, jak zjistit maximální počet slov, který se nám vejde na jeden řádek. K tomu si potřebujeme pamatovat aktuální délku načteného řádku, označme ji *akt_délka*. Dále si potřebujeme pamatovat seznam dosud nezapsaných slov. Postupně budeme načítat slova ze vstupu a jejich délku přičítat k proměnné *akt_délka*. Pokud slovo přidáváme k nenulovému řádku, musíme ještě připočíst jedničku za mezeru oddělující slovo od zbytku řádku. Pokud je *akt_délka* menší než číslo K udávající požadovanou šířku, přidáme načtené slovo do seznamu dosud nezapsaných slov. V momentě, kdy *akt_délka* přesáhne číslo K, zpracujeme aktuálně načtená slova, nastavíme *akt_délka* na délku posledního načteného slova a přepíšeme seznam nezapsaných slov tak, aby obsahoval pouze poslední načtené slovo.

Otázkou zůstává, jak budeme dosud nezapsaná slova zpracovávat. Nejprve si spočítáme celkovou délku zpracovávaných slov. Aby měl řádek správnou délku, musíme ho doplnit $K -$ *celková délka slov* mezerami. Potom mezi dvěma

[1] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

následujícími slovy bude buďto

$$\lfloor (K - \textit{celková délka slov}) / (\textit{počet slov} - 1) \rfloor + 1$$

mezer, nebo o jednu méně. Teď už zbývá jen mezery mezi jednotlivá slova správně doplnit a zpracovaný řádek vypsat.

Tento postup jde jistě naprogramovat s lineární časovou složitostí vzhledem k délce zpracovávaného vstupu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z4-3.py`

Maruška Kalousková & Klárka Tauchmanová

32-Z4-4 Bomberman uklízí

Ze vstupního souboru vytvoříme pole, které obsahuje mapu hrací plochy a najdeme polohu hráče.

Jakmile hráče najdeme, spustíme z jeho pozice prohledávání do hloubky a průběžně vypisujeme, kudy prochází. (Všimněte si, že prohledávání do šířky nemůžeme použít, protože by po mapě neustále skákalo.) Kdykoliv narazíme na políčko sousedící se zdí, zavoláme podprogram, který se postará o zbourání zdi. Tento podprogram přitom skončí na stejném políčku, kde začal, takže pak budeme moci plynule pokračovat v prohledávání.

Bourací podprogram začne prozkoumávat políčka ve všech 4 směrech, jestli náhodou nemají volné sousední políčko, kam se můžeme schovat před výbuchem. Pokud ano, položíme na políčko sousedící se zdí bombu, vypíšeme cestu k bezpečnému políčku a odpálíme bombu. Nakonec se vrátíme na místo, kde jsme přerušili prohledávání do hloubky a vypíšeme cestu, kterou jsme se vrátili. Například na následujícím obrázku položíme bombu na políčko `B`, načež se jdeme schovat na políčko `O`. Provedeme tedy posloupnost příkazů `BPPPPPDONLLLLL`.

```
##############
#...B.......#
####*####O###
```

Existují ovšem případy, kdy tento postup selže. Třeba:

```
####.###    ########
####*###    #...*..#
#P.....#    #####.##
####.###    #####P##
########    ########
```

Pokud bychom položili bombu hned vedle zdi, nezbylo by nám žádné bezpečné místo, kam se schovat před výbuchem. V takovém případě se budeme snažit položit bombu do chodby naproti zdi a schovat se do chodby, která je na ni kolmá.

```
####.###    ########
####*###    #...*.B#
#..O...#    #####O##
####B###    #####.##
########    ########
```

V jakých případech selže i tento postup? Jen pokud dosažitelná část mapy vypadá jako jedna dlouhá chodba:

```
#########
#.P.....#
###*#####
```

Tehdy je ale vše marné – zeď zbourat nelze.

Přesvědčili jsme se tedy, že náš algoritmus funguje. Zbývá určit jeho časovou složitost. Nechť N je delší strana mapy. Prohledávání do hloubky projde nejvýše N^2 políček, takže to stihne v čase $\mathcal{O}(N^2)$. Nalezne nejvýše N^2 zdí a pro každou z nich spustí bourací podprogram. Ten projde nejvýše N políček vertikálně a N horizontálně, takže trvá $\mathcal{O}(N)$. Celková časová složitost tedy je $\mathcal{O}(N^2)$ za prohledávání krát $\mathcal{O}(N)$ za bourání, čili $\mathcal{O}(N^3)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z4-4.py`

Honza Černý

32-Z4-5 Kopečková zmrzlina

Dobrým způsobem, jak začít řešit úlohu, je uvědomit si, co se v ní skrývá (pod štědrou vrstvou zmrzliny s polevou) za data. Máme zde různé komponenty, ze kterých se skládá zmrzlina. Dále mezi těmito komponentami jsou nějaké vztahy, konkrétně která komponenta může ležet na které jiné. Navíc tyto vztahy nejsou obousměrné, tedy může se stát, že na čokoládovou zmrzlinu může přijít jahodová, ale ne naopak.

A tohle už je Vám možná povědomé – jedná se o typickou grafovou úlohu.[2] Komponenty zmrzliny jsou vrcholy a vztahy mezi nimi, tedy která příchuť smí přijít na kterou, tvoří hrany, a protože tento vztah není symetrický, jsou to hrany orientované. Hrany budeme vést tak, že vedou z nižší položky do vyšší, tedy z kornoutu do zmrzlin a ze zmrzlin do jiných zmrzlin nebo do polevy.

V zadání je zaručeno, že podle Zuzčiných pravidel se nikdy nedá vrátit ke stejné komponentě znovu. To po přeložení do grafové terminologie znamená, že se v tomto grafu nenachází cykly. Jedná se tedy o acyklický orientovaný graf, často zkracován na *DAG,* anglicky *directed acyclic graph.*

Umíme tedy pravidla reprezentovat grafem, a chceme zjistit, kolik různých zmrzlin lze těmito pravidly vytvořit. Přeloženo do řeči grafů, chceme zjistit, kolik různých cest existuje z vrcholu kornoutu do vrcholu polevy.

Prohledávání do hloubky

Můžeme zkusit hledat všechny možné cesty z kornoutu do polevy pomocí prohledávání do hloubky. Implementovali bychom ho pomocí rekurzivní funkce, která dostane na vstupu nějaký vrchol. Pokud je tento vrchol polevou, zvýší počet nalezených cest o jedna, pokud není polevou, tak zavolá sama sebe na všechny následovníky aktuálního vrcholu. Následovníky vrcholu u myslíme takové vrcholy v, že existuje hrana vedoucí z u do v, ale nikoliv vrcholy, pro které existuje pouze hrana z v do u, ale už ne obráceně.

Z acykličnosti grafu máme zaručeno, že se tato rekurze nikdy nezacyklí.

Tento algoritmus jistě najde každou validní cestu, a to nejen do vrcholu polevy, ale i z kornoutu do úplně každého vrcholu. Lze to dokázat indukcí podle délky cesty.

Cesty délky 0 nalezneme spuštěním algoritmu na vrchol kornoutu. Pokud umíme nalézt všechny cesty délky N a chceme dokázat, že nalezneme i nějakou cestu délky $N+1$, víme, že do předposledního vrcholu této cesty existuje cesta délky N, kterou umíme nalézt, a z tohoto vrcholu se rekurzí dostaneme do posledního vrcholu delší cesty a tudíž ji také najdeme.

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

Zároveň algoritmus žádnou cestu nezapočítá vícekrát. Ale co kdyby se tak stalo a algoritmus by nějakou cestu započítal dvakrát? Potom by se naše rekurzivní funkce musela na poslední vrchol cesty spustit podruhé z předposledního vrcholu.

To se při jednom běhu funkce stát nemůže (na každého následovníka se spustí jen jednou), tudíž by to muselo stát potom, co se funkce na předposledním vrcholu podruhé rekurzivně spustila z nějakého předchozího vrcholu. Ale protože i předpředposlední vrcholy obou (stejných) cest jsou stejné, opět se na nich funkce musela znovu spustit rekurzivně. Takto postupně dojdeme až k prvnímu vrcholu cesty. Ale protože jemu už žádný vrchol v cestě ani v grafu nepředchází, tak na něm se funkce nemohla nikdy spustit podruhé a celý tento případ tedy nemůže nastat.

Tento algoritmus je nicméně příšerně pomalý, protože každou cestu započítá zvlášť, a cest může být v grafu až exponenciálně mnoho vzhledem k jeho velikosti. Lze například vytvořit graf, ve kterém je asi $\mathcal{O}(2^N)$ různých cest, kde N je počet vrcholů. Ale jde to i rychleji. Co kdybychom započítávali více cest naráz?

Hledáme z druhého konce

Mějme nějaký vrchol v, pro který chceme zjistit, kolik existuje různých cest z kornoutu do v. Podíváme se na všechny jeho *předchůdce,* čímž myslíme každý takový vrchol u, že existuje cesta z u do v. Co když pro každé u víme, kolik vede cest z kornoutu do u?

Toto nám výpočet velice usnadní, protože počet cest do v je právě součet počtů různých cest do všech takových u (ke každé cestě do nějakého u přidáme hranu z u do v).

Vytvoříme si funkci, která pro svůj vstupní vrchol v vrátí počet cest z kornoutu do v. To udělá tak, že se rekurzivně zavolá na každého u předchůdce v a výsledky sečte. Pokud je v vrchol kornoutu, tak vrátí jedna. Tuto funkci zavoláme na vrchol polevy a tím získáme řešení úlohy.

Tím jsme si ale vůbec nepomohli co se rychlosti algoritmu týče. Tato nová funkce stále započítá každou cestu zvlášť jako naše předchozí, jen to dělá pozpátku. Vrácení jedničky u vrcholu kornoutu je vlastně to samé jako zvýšení počítadla u polevy u předchozí funkce.

Zrychlujeme

Všimněme si ale toho, že tuto novou funkci sice budeme pro nějaký vrchol spouštět mnohokrát, ale vždy vrátí stejný výsledek.

Proto provedeme jednoduchou úpravu: poté, co hodnotu funkce pro nějaký vrchol spočítáme, tak si ji k tomuto vrcholu poznamenáme a pokud bychom ji chtěli někdy použít, nebudeme funkci spouštět znovu, ale rovnou vrátíme tuto hodnotu.

Jak rychlý bude vylepšený algoritmus s ukládáním mezivýsledků? Funkci spustíme pro každý z N vrcholů nejvýše jednou. Každé spuštění této funkce projde všechny hrany vedoucí do daného vrcholu a sečte hodnoty funkcí vrcholů na druhé straně (ať už musíme funkci spouštět znovu či jen čteme z paměti), projdeme tedy každou z M hran grafu nejvýše jednou. Celková časová složitost je $\mathcal{O}(N+M)$, tedy lineární vzhledem k velikosti grafu.

Prostorová složitost činí $\mathcal{O}(N)$ na uložení mezivýsledků a $\mathcal{O}(N+M)$ pro uložení grafu, dohromady tedy $\mathcal{O}(N+M)$.

Kuba Pelc

32-Z4-6 Trojjediné mince

Máme N mincí a zadaný seznam řádově N výsledků vážení nějakých dvou mincí. Naším úkolem je rozdělit mince do tří skupin tak, aby v každé skupině byly mince se stejnou hmotností.

Seznam vážení projdeme dvakrát. Při prvním průchodu se budeme dívat jen na ta vážení, kde hmotnost obou mincí vyšla stejná. Pro každou minci si vyrobíme spojový seznam a když narazíme na minci, která váží stejně, přidáme ji na konec tohoto seznamu.

Nyní máme vlastně takový graf. Jednotlivé mince jsou vrcholy a hrana z jedné mince vede do každé mince, která je v jejím spojovém seznamu. Uvědomíme si, že všechny mince v jedné komponentě grafu (tedy takové části grafu, která je navzájem propojená hranami, a zároveň z ní nevedou hrany do jiných částí) musí mít stejnou hmotnost. Toho využijeme v následujícím kroku.

Projdeme seznam vážení podruhé a budeme se naopak dívat na ta vážení, kde vyšla jedna mince těžší než druhá. Takové vážení nám sice neřekne, ve které hromádce přesně mince skončí, ale řekne nám něco o tom, kde která mince nebude. Pokud nám vyšlo, že mince A je lehčí než mince B, pak jistě mince A nebude v nejtěžší hromádce. Pokud navíc víme, že B není v nejtěžší hromádce, tak A nebude ani v prostřední. Naopak B určitě nebude v nejlehčí hromádce a pokud víme, že A není v nejlehčí, tak B nebude ani v prostřední.

Když tedy zjistíme novou informaci o tom, ve které hromádce mince není, zapíšeme si ji k této minci. Protože víme, že všechny mince, které má tato mince ve svém spojovém seznamu, jsou stejně těžké, zapíšeme si tuto informaci i k nim, a pokud ji ještě nevěděly, tak i k jejich sousedům a tak dál (takto vlastně používáme prohledávání do šířky). Tímto tuto novou informaci rozšíříme po celé jedné komponentě našeho grafu.

Takovýchto šiřitelných informací bude řádově stejný počet jako vážení, tedy N. Jak dlouho bude takové šíření jedné informace trvat? Protože mincí máme N, může se stát, že rozšířit jednu informaci nás bude stát až N kroků. Uvědomíme si ale, že jakmile mince už informaci má, podruhé tu stejnou informaci šířit nebude. Podíváme se tedy raději na celkový součet všech šíření informací: Protože každá mince má dvě hromádky, ve kterých není, bude během celého algoritmu šířit maximálně dvě informace. Celková doba šíření nám tedy zabere jen $\mathcal{O}(N)$ času. Proto bude celý algoritmus lineární. Paměťová složitost je $\mathcal{O}(N)$.

Zuzka Urbanová