


Vzorová řešení třetí série třicátého sedmého ročníku KSP

37-3-1 Robot

 Najdôležitejšie pri riešení tejto úlohy je uvedomiť si, aké prípady môžu nastať. Keď im budeme rozumieť, môžeme už úlohu vyriešiť pomocou jednoduchého prehľadávania grafu.

Zaujíma nás, ktoré políčka skladu sú dosiahnuteľné, teda na ktoré vieme robota prinútiť ísť. Určite to platí pre políčko, na ktorom začína. Ďalej teda musíme ísť z neho. Pokiaľ má toto políčko jedno alebo dve voľné susedné políčka, môžeme robotovi prikázať ísť na opačné pole, než chceme aby šiel, a prinútiť ho tak spraviť krok požadovaným smerom.

Pokiaľ má však susedných políčok viac, má robot po každom príkaze aspoň dve možnosti kam sa pohnúť, a teda nevieme ho dostať ani na jedno susedné políčko, pretože sa vždy pohne inak, než chceme.

Stačí nám teda spustiť BFS alebo DFS z počiatočného políčka a pokračovať len po políčkach, ktoré majú najviac dvoch susedov. Vždy keď takto nájdeme pascu, určite je dosiahnuteľná, takže všetky pasce, ktoré nájdeme, budú označené správne. Funguje takéto riešenie? Bohužiaľ nie.

Problémom je, že stále môžu existovať pasce, ktoré sme týmto procesom neoznačili. V nasledujúcom príklade môžeme vysávač naviesť do každej pasce napriek tomu, že doterajším algoritmom by sme neoznačili ani jednu:

```
##.P.  
V..#P  
##.P.
```

Ale v tomto podobnom príklade môžeme vysávač naviesť iba do prvej pasce v strednom riadku, nevieme ho naviesť do žiadnej zo zvyšných troch.

```
##.P.##  
V..#P.P  
##.P.##
```

Predošlým algoritmom by sme označili len prvé tri políčka cesty k pasciam (vrátane začiatočného). Využijeme teda tieto príklady a pridáme pozorovanie: okrem predošlých označených pascí dokážeme vysávač naviesť aj do takých, ktoré

1. ležia na ceste, ktorá spája oba východy z križovatky.
2. ležia na križovatke, kde sa stretávajú všetky vetvy, na ktoré sa nejaká cesta delí (ale nie do takých, ktoré ležia na ceste do tejto križovatky, pretože vysávač si vždy vie vybrať inú z ciest, aby sa im vyhol).

Tieto pasce vieme nájsť tak, že z každej pasce spustíme BFS prehľadanie, ktorým sa pokúsime dostať na štartovné políčko. Pokiaľ navštívime križovatku (čo vieme zistiť z počtu susedov daného políčka), označíme si, že na to, aby sme z nej mohli pokračovať, musíme ju navštíviť ešte raz za každú ďalšiu cestu, ktorá do nej vedie, okrem tej, ktorou chceme pokračovať (teda o dva menej, ako je počet jej susedných políčok, pretože z jedného sme práve prišli a jedným odídeme).

Vždy, keď danú križovatku navštívime znova, jej počítadlo znížime. Ak sa počítadlo križovatky dostane na nulu, pridáme ju do fronty pokračujeme z nej pri prehľadávaní. Ak narazíme na štartovné políčko, označíme pascu, z ktorej sme začali prehľadanie, ako dosiahnuteľnú.

Toto prehľadanie má časovú zložitosť $\mathcal{O}(R \times S \times P)$, kde P je počet pascí, pretože teoreticky môže byť potrebné prejsť pre každú pascu celý sklad.


Program (Python):

<http://ksp.mff.cuni.cz/viz/37-3-1.py>

Úlohu pripravili: Daniel Culliver, Ján „Jančí“ Plachý



37-3-2 KSP-Nim-Vim

 Úloha se ukázala být výrazně zákeřnější, než jsme plánovali. Proto nejprve vyřešíme její jednodušší variantu. Pak si uvědomíme, proč přímočarý zobecnění tohoto řešení na celou úlohu nefunguje. Následovat bude několik pozorování o struktuře optimální posloupnosti příkazů, ze kterých nakonec vykoukáme, jak řešení opravit.

Jednodušší úloha

Nejprve si rozmyslíme, jak by se úloha řešila s méně svérázným textovým editorem, který by vkládal a mazal samostatné znaky. Všimneme si, že takové příkazy jsou na sobě nezávislé – v optimální posloupnosti příkazů se jistě nestane, že bychom vložený znak později smazali. Díky tomu můžeme příkazy seřadit zleva doprava: dostaneme posloupnost vkládání, mazání a posunů kurzoru doprava.

Označíme α vstupní text a n jeho délku, podobně β výstupní text a m jeho délku. Dále bude $\alpha[i]$ značit i -tý znak řetězce α (indexováno od 0) a $\alpha[:i]$ prefix tvořený prvními i znaky řetězce α , tedy $\alpha[0]\alpha[1]\dots\alpha[i-1]$.

Definujme číslo $D(i, j)$ jako minimální počet příkazů, kterými z $\alpha[:i]$ vytvoříme $\beta[:j]$. Rozebereme, čím může optimální posloupnost příkazů končit:

- *vkládáním* – tehdy jsme z $\alpha[:i]$ vytvořili $\beta[:j-1]$, načež jsme vložili znak $\beta[j-1]$. Proto je $D(i, j) = D(i, j-1) + 1$.
- *mazáním* – tehdy jsme z $\alpha[:i-1]$ vytvořili $\beta[:j]$, načež jsme smazali znak $\beta[i-1]$. Tedy je $D(i, j) = D(i-1, j) + 1$.
- *posunem kurzoru doprava* – z prefixu $\alpha[:i-1]$ jsme vytvořili $\beta[:j-1]$, načež jsme přeskočili znaky $\alpha[i-1]$

a $\beta[j - 1]$, které se musely rovnat. Proto je $D(i, j) = D(i - 1, j - 1)$, jelikož v naší úloze jsou pohyby kurzorem zadarmo.

Z těchto tří možností si vybereme tu, která nám dá nejmenší výsledek:

$$D(i, j) = \min(D(i - 1, j) + 1, D(i, j - 1) + 1, D(i - 1, j - 1)),$$

přičemž poslední člen se do minima zahrne jen tehdy, je-li $\alpha[i - 1] = \beta[j - 1]$.

To nám dává návod, jak tabulku všech $D(i, j)$ vyplňovat po řádcích. Na okrajích bude $D(0, 0) = 0$, $D(i, 0) = i$ a $D(0, j) = j$. Náš vztah pro $D(i, j)$ se odkazuje jen na hodnoty vlevo a/nebo nahoře od pozice (i, j) , které už máme spočítané.

V čase $\mathcal{O}(nm)$ vyplníme celou tabulku. Pak nám $D(n, m)$ řekne, jak z celé α vyrobit celou β . Zkratka dynamického programování jak z učebnice (kapitola 12.3 v Průvodci).

Totéž bychom mohli interpretovat jako graf: Vrcholy budou uspořádané dvojice (i, j) s $(0, 0)$ v levém horním rohu. Hrany povedou o 1 doprava (mazání znaku), o 1 dolů (vlození znaku) a šikmo doprava dolů (přeskočení znaku, shoduje-li se). Cesty z $(0, 0)$ do (n, m) odpovídají hledaným posloupnostem příkazů. Ohodnotíme-li svislé a vodorovné hrany 1 a šikmé 0, stačí najít nejkratší cestu. To jde provést indukci podle topologického pořadí, neboť graf je acyklický. Průchod po řádcích dá topologické pořadí. Výsledný algoritmus bude takřka identický s předchozím dynamickým programováním.

Nečekaný zádrhel

Nabízí se podobným způsobem vyřešit zadanou úlohu, v níž máme příkazy na vložení nebo smazání dvojice nebo trojice znaků a opět zadarmo pohyby kurzorem. Tyto příkazy můžeme zase popsat vztahy typu $D(i, j) = D(i, j - 2) + 1$ pro vložení dvojice znaků, $D(i, j) = D(i - 3, j) + 1$ pro smazání trojice atd.

Co když potřebujeme vložit jediný znak znak x ? To můžeme nahradit vložím xyz a následným smazáním yz . To vede na vztah $D(i, j) = D(i, j - 1) + 2$.

Tím se nám ovšem rozbil předpoklad, že příkazy jsou nezávislé. Ve skutečnosti se mohou překrývat docela komplikovaně (jedno mazání může ležet přes 2 různá vložení, mezi nimiž se mohou vyskytovat původní znaky řetězce α). Proto už vůbec není jasné, co znamená seřadit příkazy zleva doprava, a celá myšlenka řešení se rozpadne.

Struktura optimálních řešení

I v této temné chvíli našťěstí mihotá světélko naděje. Ukážeme, že v optimálních řešeních většina překryvů nenastane, nebo se jim alespoň dá vyhnout.

Budeme uvažovat nějakou optimální posloupnost příkazů a postupně ji budeme upravovat, aby měla čím dál jednodušší strukturu, a přitom se neprodložila. Pohyby kurzorem zanedbáme, protože se za ně neplatí, a u příkazů vkládání a mazání si budeme pamatovat, na které pozici v řetězci nastanou.

Označíme I příkaz pro vkládání (*insert*), D příkaz pro mazání (*delete*). I_k a D_k znamená verzi pro vložení/smazání k znaků.

Předchází/následuje bude popisovat vztahy v čase (pozice v posloupnosti příkazů), *nalevo/napravo/překrývá se* vztahy v prostoru (pozice v editovaném řetězci).

Krok 1: Všechny inserty lze provést před všemi deletey.

Mějme posloupnost příkazů, pro kterou tvrzení neplatí. Náchází se tam tedy *inverze*: delete D , po kterém bezprostředně následuje insert I .

Představme si, co se stane, když I a D v nějaké inverzi prohodíme. I zjevně proběhne buď nalevo nebo napravo od D , takže I neovlivní, jak D proběhne, kromě toho, že může posunout indexy. Ty ale můžeme snadno opravit.

Pokud se tímto způsobem budeme zbavovat *první* inverze v pořadí příkazů, bude se postupně pozice nejlevější inverze posouvat ke konci posloupnosti, až budou všechny inserty před všemi deletey. Počet operací tím nezměníme, takže posloupnost bude nadále optimální.

Krok 2: Dva příkazy téhož druhu se nepřekrývají.

Dva deletey se překrývat nemohou.

Pokud se překrývaly dva inserty, znamená to, že nejprve vložíme jeden řetězec a pak dovnitř něj druhý. Tím vznikne podřetězec z 4–6 znaků, který ale umíme vyrobit i dvěma po na sebe navazujícími inserty.

Krok 3: Delete se překrývá s nejvýše 1 insertem.

Delete se může překrývat s nejvýše 2 inserty. Mezi těmito inserty mohou být nějaké znaky původní posloupnosti. Uvažme nejlevější takový delete.

Pokud mezi inserty nejsou žádné původní znaky, nejprve dva inserty vytvoří 4–6 znaků, pak z nich delete smaže 2–3, takže zbude 1–4 znaků. Každý řetězec této délky ovšem umíme vyrobit jednodušeji: délku 2–3 pomocí I_2 nebo I_3 , délku 4 dvěma I_2 , délku 1 pomocí I_3 , jehož zbytek smaže D_2 . Tím dostaneme kratší posloupnost příkazů se stejným výsledkem, takže takové v optimální posloupnosti tento případ nemohl nastat.

Pokud mezi inserty leží nějaký původní znak, musí být jediný a delete musí být D_3 (jinak by delete nemohl zasáhnout do dvou insertů současně), tím pádem delete z každého insertu smaže jeden znak. Uvažujme možné kombinace délek insertů.

V rozbořech případů budeme dodržovat následující konvence: Velká písmena pochází z původního řetězce α , malá písmena z nějakého předchozího insertu, tučná písmena maže delete. Svislé čárky oddělují jednotlivé inserty jak od sebe navzájem, tak od původních znaků.

Možné kombinace tedy jsou:

- $pq | \mathbf{A} | xy$, což dá výsledek py . Tentýž výsledek ovšem můžeme dosáhnout pomocí $pyy | \mathbf{A}$ (to je I_3 a D_2).
- $pqr | \mathbf{A} | xy$, což dá výsledek pqy . Nahradíme za $pq | \mathbf{A} | xy$ (tedy dva I_2 a pak D_2).
- $pq | \mathbf{A} | xyz$, což dá výsledek pyz . Nahradíme za $py | \mathbf{A} | qz$ (dva I_2 a pak D_2).
- $pqr | \mathbf{A} | xyz$, což dá výsledek $pqyz$. Nahradíme za $pq | \mathbf{A} | xyz$ (I_2 , I_3 a pak D_2).

Pokaždé tedy umíme dvojitý překryv nahradit jednoduchým, aniž by příkazů přibýlo. Žádný nový dvojitý překryv nevznikne, takže po konečně mnoha opakováních se všech násobných překryvů zbavíme.

Blížíme se do finále

Kombinací předchozích tří kroků získáme, že aspoň jedna z optimálních posloupností splňuje to, že se skládá z nepřekrývajících se částí těchto typů:

- přeskočení znaku společného pro α a β ,
- samostatné I_2, I_3, D_2, D_3 ,
- dvojice insertu s překrývajícím se deletem.

Tyto části opět můžeme seřadit zleva doprava a získat rekurentní vzorec. (Pořadí z kroku 1 už dodržujeme jen v rámci části.)

Jen je ještě potřeba dořešit, jak mohou vypadat dvojice insertu s deletem (příčemž delete může mazat i znaky původního textu). Budeme se držet značení z důkazu předchozího kroku:

- Levější z dvojice příkazů je I_2 , pravější D_2 :
 - $p\mathbf{q} = \varepsilon$ (prázdný řetězec): neprovede nic
 - $p\mathbf{q} \mid \mathbf{A} = p$: to se chová jako D_1
- Nalevo I_2 , napravo D_3 :
 - $p\mathbf{q} \mid \mathbf{A} = \varepsilon$: to je jako D_1
 - $p\mathbf{q} \mid \mathbf{AB} = p$: to je jako I_1D_2
- Nalevo I_3 , napravo D_2 :
 - $p\mathbf{qr} = p$: to je jako I_1
 - $p\mathbf{qr} \mid \mathbf{A} = pq$: to je jako I_2D_1
- Nalevo I_3 , napravo D_3 :
 - $p\mathbf{qr} = \varepsilon$: neprovede nic
 - $p\mathbf{qr} \mid \mathbf{A} = p$: to je jako I_1D_1
 - $p\mathbf{qr} \mid \mathbf{AB} = pq$: to je jako I_2D_2
- Kombinace s deletem nalevo od insertu vyjdou zrcadlově.

Dvojice, které neprovedou nic nebo jsou ekvivalentní s D_2 , nemusíme brát v úvahu – téhož efektu jde dosáhnout méně příkazy. Variantu I_2D_2 umíme provést i bez překryvu. Ostatní dvojice nám dvěma příkazy dokáží simulovat $I_1, D_1, I_1D_1, I_2D_1, I_1D_2$ (nepřekrývající se).

Hotový algoritmus

Teď už můžeme dát dohromady celý algoritmus. Z předchozího rozboru případů odvodíme:

$$D(i, j) = \begin{cases} D(i-1, j-1) & \text{pokud } \alpha[i-1] = \beta[j-1] \\ D(i-1, j-1) + 2 & \text{simulovaný } I_1D_1 \\ D(i-1, j) + 2 & \text{simulovaný } D_1 \\ D(i-2, j) + 1 & D_2 \\ D(i-3, j) + 1 & D_3 \\ D(i, j-1) + 2 & \text{simulovaný } I_1 \\ D(i, j-2) + 1 & I_2 \\ D(i, j-3) + 1 & I_3 \\ D(i-1, j-2) + 2 & \text{simulovaný } I_2D_1 \\ D(i-2, j-1) + 2 & \text{simulovaný } I_1D_2 \end{cases}$$

Varianty, kterým vyjdou indexy mimo tabulku, vynecháme.

Tabulku můžeme opět vyplnit po řádcích v čase $\mathcal{O}(nm)$, jako okrajovou podmínku postačí položit $D(0, 0) = 0$.

Ještě je potřeba domyslet, jak rekonstruovat posloupnost příkazů. Na to stačí pamatovat si při výpočtu tabulky, pro kterou variantu se minimum nabývalo. To umožní rekonstruovat posloupnost pozpátku od $D(n, m)$.

Program (Python):

<http://ksp.mff.cuni.cz/viz/37-3-2.py>

Úlohu připravili: Honza Černohorský, Martin „Medvěd“ Mareš, Vladimír Sklenář, Dan Skýpala

37-3-3 Hledání podmatice

Kdybychom tuto úlohu řešili v 1D světě, šlo by o klasický problém hledání v textu. Na ten existuje například algoritmus KMP, o kterém se dočtete v kuchařce.¹ My ale potřebujeme hledat ve 2D. Celkem přímočarý způsob, jak převést algoritmus do více dimenzí, je zpracovávat jednotlivé osy postupně.

Nejdříve budeme hledat po řádcích. Jelikož hledáme výskyt několika různých řádků malé matice, použijeme algoritmus Aho-Corasicková. Ten má navíc tu výhodu, že pokud jsou některé řádky malé matice identické, tak je najdeme, protože jim bude odpovídat stejný vrchol v trii. Projdeme každý řádek velké matice a na všech políčkách si zaznamáme, v jakém stavu (vrcholu trie) jsme byli. Tím dostaneme novou matici, která nám říká, jestli a jaký malý řádek na daném místě končí.

Poté budeme hledat po sloupcích. Sloupec obsahuje pravý kraj výskytu malé matice, právě když jsou v něm ve správném pořadí konce řádků malé matice. To je opět hledání v textu, jen s jinou abecedou tvořenou vrcholy trie. Jelikož je tentokrát jenom jedna jehla, použijeme algoritmus KMP. Projdeme každý sloupec zpracované velké matice. Každý výskyt jehly pak odpovídá pravému dolnímu rohu výskytu malé matice.

Celková časová složitost algoritmu je $\mathcal{O}(N^2 + K^2)$. Trii si připravíme v $\mathcal{O}(K^2)$, KMP tabulku v $\mathcal{O}(K)$ a obě prohledávání trvají $\mathcal{O}(N^2)$. Pohyb v trii sice závisí na abecedě, ale ta je v prvním prohledávání konstantní. Ve druhém hledání máme abecedu velikosti až K^2 , ale KMP našťastí umí pracovat bez zpomalení s libovolně velkou abecedou. V druhém prohledávání můžeme vynechat prvních $K-1$ sloupců, protože nemůžou obsahovat konec malého řádku, ale asymptoticky to nic nezmění.

Program (C++):

<http://ksp.mff.cuni.cz/viz/37-3-3.cpp>


Program (Python):

<http://ksp.mff.cuni.cz/viz/37-3-3.py>

Úlohu připravili: Jan Adámek, Ján „Jančí“ Plachý

37-3-4 Výhybky

(Ne)obvyklé hradlo

 Nejdřív si ukažme, že z výhybkového hradla jdou sestavit hradla, která všichni známe. NOT sestrojíme jednoduše – vybereme nulu, pokud je vstupní hodnota jedna, a opačně:

NOT: X 0 1

U ANDu si všimneme, že pokud první vstup je nula, výsledek je určitě nula. Jinak stačí vrátit druhý vstup:

AND: X Y 0

OR vyřešíme obdobně:

OR: X 1 Y

Zbývá vyřešit XOR. Pokud oba vstupy jsou jedna, vrátíme nula. Jinak vrátíme jejich OR:

AND: X Y 0

OR: X 1 Y

XOR: AND 0 OR

¹ <http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu>

Čas řešit

Nyní pojďme řešit jednotlivé podúlohy. V první podúloze stačí udělat AND všech vstupů. Abychom dosáhli logaritmické hloubky, tak stačí udělat ANDy dvou sousedních hodnot. (Pokud je počet vstupů lichý, tak nám jedna zůstane.) A to stačí dělat opakovaně, čímž skončíme s jednou hodnotou. Protože v každé vrstvě vydělíme počet hodnot dvěma (a zaokrouhlíme nahoru), hloubka bude logaritmická.

Druhou podúlohu vyřešíme obdobně – stačí vyměnit AND za OR.

V třetí podúloze chceme zjistit, jestli máme právě jednu jedničku. Na to zobecníme náš postup. Máme nějaké hodnoty reprezentující úseky. (V předchozích podúlohách to byl AND nebo OR daného úseku.) Poté v jedné vrstvě zkombinujeme sousední dvojice do jedné, čímž dané hodnoty budou odpovídat úseku o dvojnásobné délky. (Až na případy, kdy počet hodnot je lichý.)

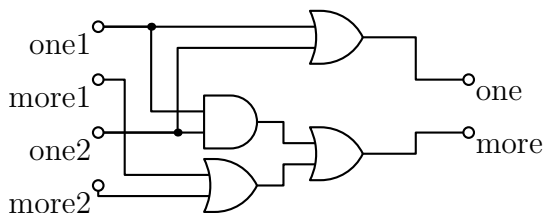
A v této podúloze si budeme posílat následující dvojici:

- Máme alespoň jednu jedničku (**one**)
- Máme alespoň dvě jedničky (**more**)

Na začátku v každém úseku o máme dvojici (i -tý vstup, konstantní nula). Kombinovat dvojice s již zkonstruovanými hradly zvládneme následovně:

$one = one1 \text{ OR } one2$

$more = (one1 \text{ AND } one2) \text{ OR } more1 \text{ OR } more2$

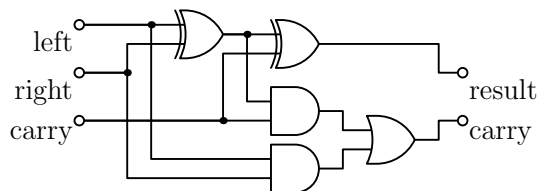


Na konci stačí pouze zkontrolovat, že výsledná dvojice je (1, 0). To můžeme udělat následovně: Ze všech hodnot, co mají být 0, uděláme NOT, a následovně všechny hodnoty zANDujeme.

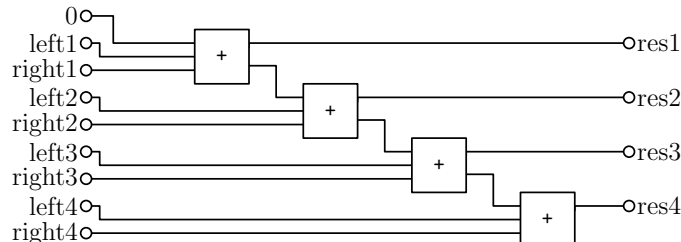
Vzhůru na čtvrtou úlohu – zde je už doopravdy zapotřebí počítat počty jedniček. Naše úseky budou reprezentovány $\log n$ bity, udávajícími počet jedniček v daném úseku. (Na začátku jen vstupní hodnoty zleva doplníme konstantními nulami.) A zkombinování dvou úseku odpovídání sečtení dvou $\log n$ bitových čísel.

Na to si nejdříve sestrojme primitivní obvod, který sečte tři bity – bit z levého úseku, bit z pravého úseku a bit za pře-

nos. Výsledek bude tedy dvoubitové číslo (**carry**, **result**):



Když už máme sčítačku, tak jsme je schopni zapojit za sebe pro libovolně dlouhá čísla (příklad pro 4 bity):



Kde poslední **carry** bit prostě zahodíme – víme, že tolik jedniček v našem vstupu prostě nemůže být. Na závěr stačí opět znegovat bity, co mají být nulové, a udělat AND.

Nicméně tohle použije moc hradel. Uděláme proto ještě jednu optimalizaci – všimneme si, že v i -té vrstvě jedničkových může být pouze nejnižších i bitů. Když tyto nepotřebné části obvodu odstraníme, už se do limitu počtu hradel vlezeme.

Program (Python):

<http://ksp.mff.cuni.cz/viz/37-3-4.py>

Magnum opus

Na závěr zmiňme, že řešení čtvrté podúlohy jde použít na řešení všech ostatních. – Stačí spočítat počet jedniček a porovnat ho s požadovaným počtem. Pouze u podúlohy na alespoň jednu jedničku stačí porovnat s nulou a udělat negaci.

Program (Python):

<http://ksp.mff.cuni.cz/viz/37-3-4-mo.py>

Úlohu připravili: Dan Skýpala, Ben Swart

37-3-S Parsery vrací úder

Seriál lze odevzdávat za snížený počet bodů až do konce školního roku. Řešitelům, kteří už třetí sérii odevzdali, rozešleme vzorové řešení napřímo. Pokud chceš tuto sérii přeskočit a začít řešit další díl, můžeš si o vzorové řešení napsat na ksp@mff.cuni.cz.