Korespondenční Seminář z Programování

32. ročník **KSP** **Červen 2020**

Milé řešitelky a řešitelé!

Poslední termín odevzdání už uplynul a my postupně vydáváme vzorová řešení zbývajících úloh 32. ročníku. Konec školního roku oslavíme řešeními čtvrté série. Brzy se k ní přidá i pátá a poněkud opožděný seriál ze třetí série (kdo mohl čekat, že zpoždění postihuje nejen vlaky, ale i úlohy o nich?).

Za zmínku určitě stojí řešení úlohy 32-4-3, ze které se vyklubala záhada hodná Sherlocka Holmese, a praktická úloha 32-4-6 (sudoku), u které navzdory očekávání známe optimální odpověď. Ale i ostatní řešení stojí za přečtení. Samostatné komentáře opravujících tentokrát nevydáváme.

Také už brzy očekávejte pozvánky na podzimní soustředění.

Vzorová řešení čtvrté série třicátého druhého ročníku KSP

32-4-1 Výbušné koťátko

Tato úloha měla původně (dnes už zcela politicky nekorektní) název Josefův problém.[1]

V celém tomto řešení budeme předpokládat, že počítáme modulo počet aktivních hráčů. Taktéž měl autor přesvědčení, že detaily, jak počítat indexy při změnách kroužku, snad každý čtenář dokáže odvodit sám. Rozepisovat je zde by jen zhoršilo čitelnost.

Jak by se implementovalo simulující řešení zmíněné v zadání si, doufejme, každý zvládne rozmyslet vlastní hlavou.

Lepší možné řešení používá intervalový strom, stačí implementace pomocí pole, nebude třeba vkládat prvky. Listy budou odpovídat hráčům v pořadí, v jakém stojí v kroužku. Hodnoty v listech budou 0, či 1. 1 bude znamenat, že hráč na tomto místě ještě nevypadl. Na vyšší úrovně se pak bude ukládat počet prvků v levém podstromu. Algoritmus v každém kroku odsimuluje jedno vypadnutí, se znalostí počtu hráčů v levém podstromě umíme sestupem z kořene najít hledaného hráče.

Toto řešení má složitost $\mathcal{O}(n \log n)$, existuje však ještě rychlejší řešení.

Zamyslíme se, zda by k určení toho, kdo bude vítězem nešla použít znalost relativní pozice vítěze vůči začínajícímu hráči v kroužku, kde je o jednoho hráče méně. Ukazuje se, že ano. Před začínajícího hráče v menším kroužku vložíme dalšího hráče a aktivním se stane hráč k pozic před ním. Protože si pamatujeme pouze pozici vítěze a aktivního hráče, tyto operace umíme provést v konstantním čase.

Začneme tedy u kolečka se dvěma hráči, kde už přímo určíme, kdo je vítěz. Pak postupně přidáváme další hráče, až se dostaneme ke kroužku požadované velikosti. Nakonec umíme určit relativní pozici vítěze vzhledem k počátečnímu hráči.

Toto ještě zvládneme vylepšit pro $k \in o(n/\log n)$. Nahlédneme, že během jednoho kroku vlastně nemusíme přidávat do kroužku jen jednoho hráče. Všechny změny během jednoho oběhu koťátka po kroužku dokážeme promítnout do jednoho kroku. Stále totiž dokážeme pozice aktivního hráče a vítěze po rozšíření v konstantním čase.

Za každých k hráčů tak v jednom kroku přibude nový. Jinak řečeno, počet hráčů se tak v jednom kroku zvýší $(k+1)/k$-krát. Počet kroků je tedy

$$\log_{(k+1)/k}(n) = \frac{\log n}{\log\left((k+1)/k\right)}.$$

S k jdoucím do nekonečna se výraz $\left(\frac{k+1}{k}\right)^k$ blíží k Eulerovu číslu $\mathrm{e} \doteq 2.718$. Odtud odmocněním $e^{1/k} \approx \frac{k+1}{k}$, použitím logaritmu $1/k \approx \log\left(\frac{k+1}{k}\right)$.

Výsledná složitost po dosazení tedy činí $\mathcal{O}(k \log n)$.

Nakonec se ještě hodí zmínit, že by se postup posledního řešení dal implementovat i pomocí rekurze na menší kroužky, ale v takovém případě bude potřebná paměť lineární s počtem hráčů.

Jiří Škrobánek

32-4-2 Jeden výstřel

Přeloženo do terminologie grafů chceme odstranit hranu tak, abychom zmenšili minimální kostru. (Pokud graf není souvislý, kostra nebude strom, nýbrž les.)

Mějme tedy nějaký graf G a jeho minimální kostru K. Pokud smažeme nějakou hranu, dostaneme nový graf G' s minimální kostrou K'. Přitom kostra K' se nachází i v původním grafu G. Takže nemůže být menší než původní minimální kostra K, a proto úloha nemá žádné řešení. Aha!

To by bylo jednoduché, jenže není to pravda. Kostra K' se sice v grafu G nachází (jako podgraf), ale to neznamená, že i tam musí být kostrou. Může se totiž stát, že se graf G smazáním hrany rozpadne na více komponent – pak kostra K' nepropojuje celý graf G, ale jen každou komponentu zvlášť. Tedy není kostrou grafu G.

Naší jedinou šancí tedy je smazat takovou hranu, aby se graf G rozpadl. Hranám s touto vlastností se říká *mosty* a existuje hezký lineární algoritmus na jejich hledání. Najdete ho třeba v naší grafové kuchařce[2] v oddílu „Vrcholová a hranová 2-souvislost“, případně v kapitole 5.7 Průvodce labyrintem algoritmů.[3]

Smažeme-li nějaký most, nová minimální kostra vznikne smazáním mostu z původní minimální kostry (to vyplývá

[1] Úloha má vskutku starobylý původ: poprvé ji zmínil v 1. století n. l. Flavius Josephus v knize Židovská válka.

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

[3] `http://pruvodce.ucw.cz/`

třeba z kteréhokoliv algoritmu na hledání minimální kostry). Čím větší energii most má, tím menší energii má výsledná kostra.

K řešení úlohy tedy vůbec nepotřebujeme hledat kostru – postačí smazat most s největší energií. Umíme ho najít v čase $\mathcal{O}(n+m)$ pro graf s n vrcholy a m hranami.

Ještě dodejme, že kdybychom v grafu bez mostů opravdu museli do nějaké hrany vystřelit, vybrali bychom si maximální hranu. Ta určitě není součástí minimální kostry: o tom se přesvědčíme spuštěním algoritmu z kuchařky na hledání minimální kostry. Algoritmus vezme maximální hranu do ruky až jako poslední a tehdy už musí být její konce spojené – jinak by tato hrana byla mostem. Takže pokud smažeme maximální hranu, minimální kostra se nezmění. A to je nejlepší, čeho můžeme v grafu bez mostů dosáhnout.

Standa Lukeš & Martin „Medvěd“ Mareš

32-4-3 Sběrnice na pájivém poli

Tato úloha se ukázala jako mnohem zákeřnější, než jsme plánovali. Měli jsme vymyšlené krásné lineární řešení (které se vzápětí dočtete), ale když jsme ho sepisovali, uvědomili jsme si, že je špatně. Úplně stejně se nachytali i všichni účastníci, kteří úlohu řešili. Z toho, proč je řešení rozbité a co si s tím počít, se nakonec vyklubal napínavý příběh. Tak si uchystejte pohodlné křeslo a popcorn, jdeme na to!

Čtverec v bludišti

Cílem úlohy je najít co nejširší pruh cestiček vedoucích od horní hrany ke spodní, které se nikde nerozcházejí. Uvědomíme si, že toto lze vyřešit nalezením největšího čtverce, který lze od horní hrany k té spodní protáhnout. Pokud najdeme takovou cestu pro čtverec o velikost $k \times k$, jistě se do jeho trasy vejde k jednotlivých cestiček.

Bude se nám hodit přepočítat si pro každé políčko (r, s) hodnotu $C(r, s)$, která nám řekne, jaký největší čtverec bez překážek může mít pravý dolní roh na tomto políčku (pro jednoduchost říkejme, že na tomto políčku *končí*). Všimneme si, že pro políčko s překážkou je $C(r, s) = 0$, a pro prázdné políčko platí

$$C(r, s) = \min(C(r-1, s), C(r, s-1), C(r-1, s-1)) + 1.$$

Proč je to pravda? Pokud čtverec $k \times k$ končí na políčku (r, s), pak nalevo od tohoto políčka končí nějaký čtverec $(k-1) \times (k-1)$ a podobně na políčku nahoře a šikmo vlevo nahoře. A naopak: pokud na těchto políčkách končí čtverce $(k-1) \times (k-1)$, pak na políčku (r, s) končí čtverec $k \times k$.

Podle tohoto vztahu můžeme spočítat $C(r, s)$ pro všechna políčka, stačí postupovat po řádcích. Jen musíme $C(r, s)$ dodefinovat na okrajích. Tak k destičce pomyslně přidáme nultý sloupec plný překážek ($K(r, 0) = 0$) a nultý řádek, nad kterým je samé volné místo ($K(0, s)$ ovšem není $+\infty$, ale s, protože čtverec se zarazí o nultý sloupec).

Předvýpočet tedy zvládneme v čase lineárním s počtem políček, tedy $\mathcal{O}(RS)$, kde R a S jsou rozměry destičky.

Nyní můžeme v čase $\mathcal{O}(RS)$ pro konkrétní k otestovat, zda se čtvercem $k \times k$ dostaneme od horní hrany ke spodní. Na to nám stačí třeba obyčejné prohledávání do šířky, kde budeme uvažovat jen políčka, na která se vejde dostatečně velký čtverec.

Chceme-li najít největší takové $k \in \{0, \ldots, S\}$, pomůže nám binární vyhledávání. Provedeme celkem $\mathcal{O}(\log S)$ kroků, v každém z nich spustíme předchozí algoritmus, takže to celkem zabere čas $\mathcal{O}(RS \log S)$.

Lineární řešení

Předchozí algoritmus můžeme zrychlit. Začneme s $k = S$ a prohledáním do šířky najdeme všechna políčka, kam se dá od horní hrany dostat čtvercem $k \times k$. Pak budeme k postupně snižovat po 1 a přepočítávat množinu dosažitelných políček. Jakmile začne být dosažitelné nějaké políčko na dolní hraně, skončíme.

Abychom množinu zvládli přepočítávat rychle, budeme si udržovat všechna políčka sousedící s dosažitelnou částí a rozdělíme si je do přihrádek podle hodnoty $C(r, s)$. Jakmile snížíme k o 1, přidáme do fronty všechna políčka z $(k-1)$-ní přihrádky a necháme prohledávání do šířky běžet dál.

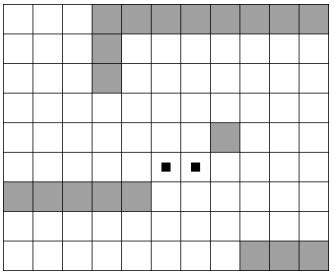
Takto každým políčkem strávíme celkově konstantní čas: nejprve ho inicializujeme jako nenalezené, pak se stane sousedním a buď se dostane rovnou do fronty prohledávání, nebo ještě bude chvíli čekat v přihrádce. Pak ho označíme jako dosažitelné a už se jím nebudeme dál zabývat.

Algoritmus tedy celkem poběží v čase $\mathcal{O}(RS)$.

Vyvrácení (a jiné špatné zprávy)

Jak už jsme přiznali v úvodu, uvedený algoritmus nefunguje. Co je špatně? Inu, hned počáteční převod vedení sběrnice na protažení čtverce bludištěm.

Skutečně je pravda, že po každé sběrnici šířky k lze protáhnout čtverec $k \times k$. Ovšem opačně to neplatí: když někudy protáhneme čtverec $k \times k$, ještě to neznamená, že tudy můžeme vést sběrnici šířky k. Uvažujme třeba následující příklad (šedivá políčka jsou blokovaná součástkami):



Po tomto pájivém poli je možné protáhnout čtverec 3×3, ovšem maximální šířka sběrnice je 2, protože sběrnice šířky 3 by musela protnout sama sebe na políčkách označených černým čtverečkem, což evidentně neodpovídá validní sběrnici.

Náš elegantní lineární algoritmus tedy řeší poněkud jinou úlohu, než kterou jsme zadali. Co se původní úlohy týče, bohužel vás zklameme – žádný efektivní algoritmus na ni neznáme (efektivním teď myslíme jakýkoliv, jehož složitost je polynomiální v počtu políček). A co víc, máme důvodné podezření, že ho nezná ani nikdo jiný.

Ona totiž naše úloha patří mezi takzvané NP-úplné problémy. To je skupina problémů, pro které není známý žádný algoritmus s polynomiální složitostí. Navíc lze tyto problémy na sebe převádět, takže efektivní řešení jedné z nich by se dalo použít i k vyřešení všech ostatních. Jestli nějaké efektivní řešení mají, nevíme – je to jeden z největších otevřených problémů současné informatiky. Blíže se o tomto tématu dočtete v naší kuchařce o těžkých problémech.[4]

[4] `http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy`

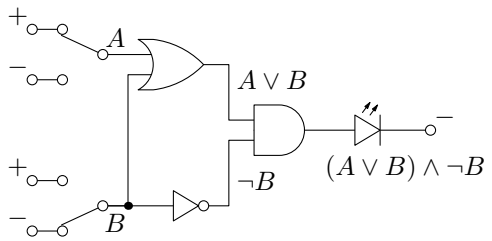
Důkaz NP-úplnosti

NP-úplnost naší úlohy se samozřejmě sluší dokázat. Nejdřív se ale musíme vypořádat s tím, že NP-úplné problémy se definují jako *rozhodovací* – tedy odpověď může být pouze ANO nebo NE. Proto zadání maličko přeformulujeme a budeme se ptát jen na to, zdali lze protáhnout sběrnici konkrétní šířky k. Původní úlohu můžeme vyřešit pomocí dotazů na všechna k od 0 do R a vypsat největší, na které je odpověď ANO. Pro zmenšení počtu dotazů lze využit binárního vyhledávání.

Pomocí našeho problému bychom rádi vyřešili některý z již známých NP-úplných problémů. Tedy chceme vytvořit algoritmus, který v polynomiálním čase z libovolného zadání NP-úplného problému vytvoří zadání naší úlohy se stejnou odpovědí. Jako zdrojový NP-úplný problém si vybereme SAT neboli splnitelnost logické formule. V něm na vstupu dostaneme logickou formuli složenou z proměnných a logických spojek $\neg$ (negace, NOT), $\wedge$ (konjunkce, AND) a $\vee$ (disjunkce, OR). Máme zjistit, zda je možné přiřadit proměnným hodnoty 1 a 0 tak, aby formule byla pravdivá.

Například formule $(A \vee B) \wedge \neg B$ je splnitelná (nastavíme $A = 1$ a $B = 0$), zatímco $A \wedge \neg A$ splnitelná není.

SAT si pro potřeby této úlohy můžeme představit jako elektrický obvod. Na jeho začátku je několik přepínačů představující jednotlivé proměnné. Na ně jsou v nějakém pořadí napojena v několika vrstvách logická hradla (tedy součástky s jedním nebo dvěma vstupy a jedním výstupem, které se chovají jako logické spojky). Na konci je jedno hradlo, které je napojeno na LED, která se rozsvítí, když hradlo bude mít na výstupu 1. Úkolem je zjistit, zdali je možné nastavit přepínače tak, aby se dioda rozsvítila.



Ukážeme, že libovolný takový obvod zvládneme simulovat pomocí pájivého pole s překážkami. Většina pole bude zaplněná. Volné zůstanou jenom úzké koridory šířky 3. Ty se budou větvit a občas do sebe rohem zasahovat, aby sběrnice o stejné šířce nemohla procházet oběma koridory současně. Zajistíme přitom, že polem půjde protáhnout sběrnice šířky 3 právě tehdy, když logický obvod bude splnitelný. To je přesně převod SATu na naši úlohu.

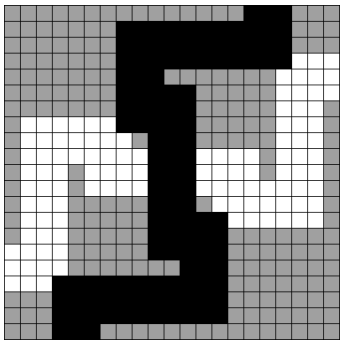
Postupně tedy ukážeme, jak odsimulovat všechno, co se v obvodu vyskytuje: drátky, jejich křížení, hradla a LED.

Signály. Každý drátek spojující hradla obvodu bude odpovídat dvojici paralelních koridorů. Sběrnice bude procházet právě jedním z nich. Podle toho, kterým bude procházet, bude reprezentovat jedničku nebo nulu. Každé takové dvojici koridorů budeme říkat *signál.*

V místech začátku a konce signálů mohou být koridory svedeny zase do jednoho. Mezi ně pak můžeme napojit jednoduché koridory tak, aby jediná spojnice horní a dolní hrany pole postupně procházela všemi signály. Tyto spojovací části označme jako *napájení.*

Signál může začít přivedením napájení do jednoho z koridorů (v tomto případě vlastně máme přepínač přepnutý do určité polohy). Také můžeme napájení rozvětvit do obou koridorů. V tento moment dáváme sběrnici možnost vybrat si koridor, čili zvolit polohu přepínače. Nakonec u LED určíme, že signál musí mít pravdivou hodnotu – zablokujeme nepravdivý koridor. Obvodem tedy bude možné protáhnout sběrnici šířky tři jen tehdy, když bude možné nastavit přepínače tak, aby LED svítila, tedy bude SAT řešitelný.

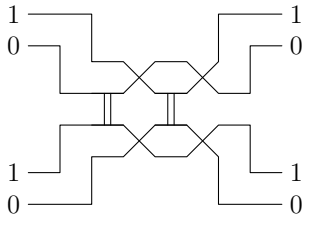
Křížení koridorů. Nejprve začneme křížením dvou koridorů (pozor, to není to samé jako křížení dvou signálů). Křížení koridorů (viz obrázek) funguje tak, že přes něj smí procházet pouze jedna sběrnice, která musí spojovat protilehlé strany (druhé dvě strany pak budou zablokovány). Každá sběrnice šířky 3, která se dostane do prostřední oblasti, zablokuje oba sousední koridory, tedy musí opustit křížení protilehlým koridorem.



Křížení můžeme také přímo použít jako *negaci* signálů. Stačí prohodit oba koridory tvořící signál.

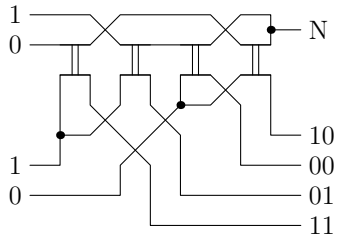
Pro vybudování dalších součástek budeme často využívat křížení. Pro názornost budeme kreslit symbolické náčrtky. Koridory šířky tři budeme značit jako čárky. Křížení dvou čárek znamená křížení koridorů, zatímco křížení tří nebo čtyř čárek s tečkou v jejich průsečíku je symbolem prostého spojení koridorů (tedy rozdvojka či roztrojka). Kolizi dvou koridorů budeme značit spojením daných koridorů pomocí dvou čárek. (V takovémto místě sdílí koridory společné políčko. Sběrnice se sice nemůže dostat z jednoho koridoru do druhého, ale nemůže procházet oběma současně, protože by na dané políčko vstoupila dvakrát).

Rovnítko. Na následujícím obrázku je naznačené, jak můžeme vynutit pomocí dvou kolizí koridorů, aby dva signály měly stejnou hodnotu. Když je v jednom signálu nějaká hodnota, v druhém nemůže být opačná, tedy tam musí být stejná. Můžeme umístit každý ze signálů na jednu stranu součástky. Tedy levý horní vstup vyvedeme na jeho horní hranu a pravý dolní vstup na hranu spodní. Levý spodní a pravý horní zůstanou vlevo, respektive vpravo. Tím v podstatě dostaneme křižovatku signálů, kde musí mít signály stejnou hodnotu. Tuto součástku proto nazveme *rovnítko.*

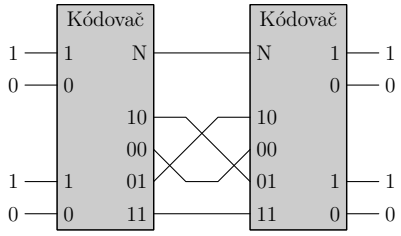


Rovnítko můžeme také použít k rozdvojení signálu: rozdvojením napájení si pořídíme nový signál bez určené hodnoty, načež rovnítkem vynutíme jeho rovnost s původním signálem.

Kódovač. Zařízení na následujícím obrázku budeme nazývat *kódovač*. Z levé strany do něj lze napojit dva signály. Na pravé straně pak vychází jedno napájení a čtyři koridory, z nichž může sběrnice procházet maximálně jedním. Lze nahlédnout, že pro každou kombinaci hodnot signálů na vstupu existuje jenom jedna varianta, kudy může vést sběrnice do pravých čtyř koridorů. Ostatní totiž buď nejsou napojené na koridor, kudy přichází spodní sběrnice, nebo jsou zablokovány horní sběrnicí. Ovšem i opačně, když sběrnice prochází jedním ze čtyř pravých koridorů a zároveň pravým napájením, tak je jasně dáno, jakou hodnotu musí mít levé signály. Zařízení tedy funguje i jako dekódovač.

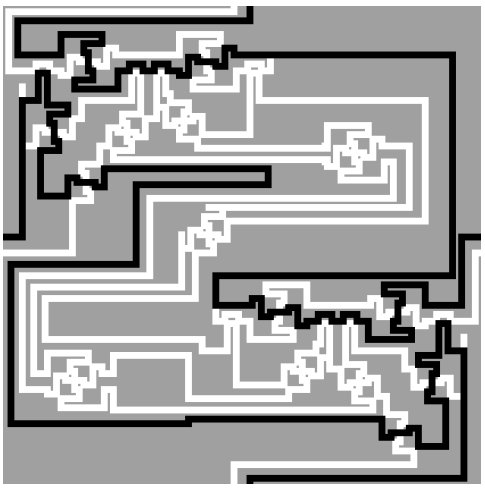


Křížení dvou signálů lze udělat následovně: Nejprve jedním kódovačem z nich uděláme napájení a čtveřici koridorů. Tu pak vhodně prohodíme. Nakonec dalším kódovačem z prohozených koridorů a napájení zase sestrojíme dva signály. U prohození si koridory odpovídající hodnotám 01 a 10 vymění místa; k tomu můžeme použít dříve sestrojené křížení koridorů, protože sběrnice může vést nejvýše jedním z těchto koridorů.



Dále budeme uvažovat, že křížení signálů má jeden vstup z každé strany. Tedy levý horní vstup vyvedeme na jeho horní hranu a pravý dolní vstup na hranu spodní. Levý spodní a pravý horní zůstanou vlevo, respektive vpravo.

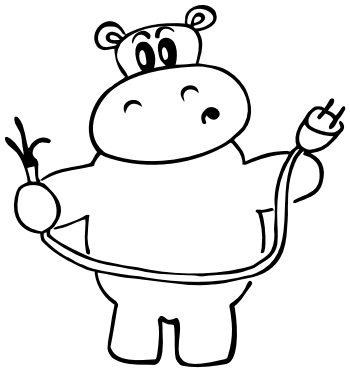
Na následujícím obrázku je zobrazena tato konstrukce přímo pomocí překážek na pájivém poli. Bílé pixely značí volný prostor a černé jeden z možných způsobů vedení sběrnice.



Pozor na to, že křížení signálů a také rovnítko kříží pouze logické hodnoty, nikoliv samotné trasy sběrnic. Na to je třeba myslet při napojování signálů na sebe tak, aby sběrnice musela projít všude, kde je potřeba.

Převrácením křížení či rovnítka kolem vertikální osy vznikne součástka se stejnými vlastnostmi, přes kterou bude sběrnice procházet jinak. V původní součástce totiž vedla sběrnice mezi horní a pravou stranou a druhá část mezi dolní a levou. V upravené součástce však povede mezi horní a levou stranou a jiná část bude spojovat dolní a pravou stranu.

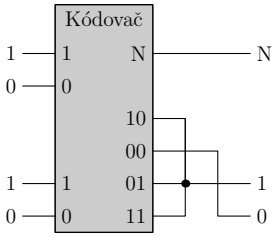
Ještě dodejme, že křižítko signálů se dá také využit pro křížení signálu a napájení. Napájení totiž můžeme považovat za signál: prostě ho napojíme na jeden z koridorů signálu.



Hradla. S pomocí kódovače lze také vytvořit všechna hradla se dvěma vstupy. Koridor odpovídající jedničce na výstupu přípojíme na všechny kombinace, pro které má být výstup pravdivý, a ostatní připojíme na koridor odpovídající nule. K tomu mohou být zapotřebí další křížení koridorů.

Z hradla kromě výstupního signálu vychází i jedno napájení, které bude potřeba někam odvést.

Na následujícím obrázku je načrtnut OR.



Rozmístění. Posledním problémem je rozmístění součástek a napojení jednotlivých signálů a napájení. Nejprve si pro každou neznámou a mezivýsledek (tedy výstup z hradla) připravíme jeden řádek, v němž povede signál odpovídající hodnoty.

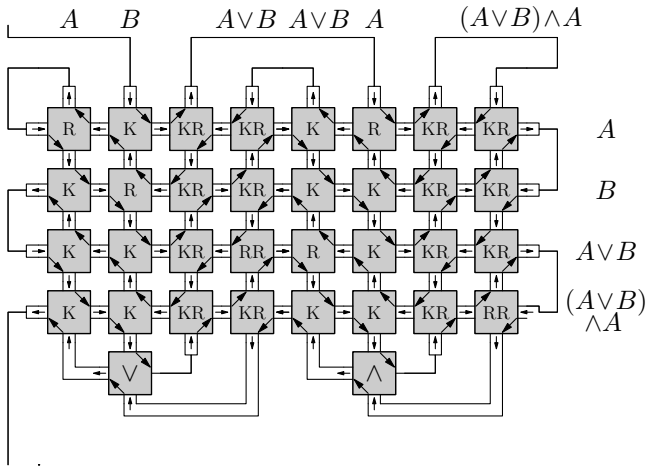
Hradla pak budeme přidávat po sloupcích: Pro každé hradlo nejprve pomocí jednoho rovnítka a několika křížení přes každý signál dovedeme vstupy daného hradla do spodní části konstrukce. Přes všechny řádky tedy povedeme signální sloupec, u kterého pomocí rovnítka místo jednoho křížení vynutíme rovnost s požadovaným řádkem. Ve spodní části obvodu pod těmito sloupci je umístěné samotné hradlo. Napájení vedoucí z hradla společně s jeho výstupem pomocí opačně otočených křížení signálů dostaneme zase zpět nahoru (a pomocí jednoho rovnítka, které je také otočené kolem své osy, stejným způsobem připojíme výstup).

Nad tyto řádky pak přidáme dva řádky napájení, které vždy povedou do sloupců vedoucí k hradlu a zase se vrátí pomocí dvojice opačných křížení.

Jednotlivé řádky pak napojíme do podoby hada. Tedy první řádek napojíme z horní hrany strany pájivého pole na jeho

levou stranu. První řádek spojíme s druhým pravou stranou a pak zase další dvojici levou stranou. Poslední řádek pak napojíme na spodní stranu pájivého pole.

Funkčnost celého propojení se dá nahlédnout z následujícího obrázku. V něm je vyznačeno převedení výroku $(A \vee B) \wedge A$. Čtverečky s písmenem K, respektive KR značí křížení signálů respektive jeho převrácenou verzi, podobně písmena R a RR pro rovnítko. $\wedge$ a $\vee$ jsou hradla.



U každého přepínače ponecháme možné obě dvě varianty. Průchod sběrnice šířky 3 polem pak odpovídá nějaké konkrétní volbě proměnných, příslušným výstupům hradel a vstupu LED. Pokud ovšem u LED zablokujeme nulový koridor, bude průchod sběrnice vždy popisovat volbu proměnných, u které obvod odpoví jedničkou. Sběrnici je tedy možné polem protáhnout právě tehdy, když původní logický obvod je splnitelný.

Tím jsme úspěšně převedli SAT na náš problém, takže i náš problém je NP-úplný. Zbývá ověřit, že převod zvládne běžet v polynomiálním čase. Vskutku: naše konstrukce běží v čase $\mathcal{O}((p+h) \cdot h)$, kde p je počet proměnných a h počet hradel, a vygeneruje stejně velký výstup.

Převod jsme si dokonce naprogramovali a otestovali na výroku $A \wedge B$. Vygenerované pájivé pole se nám ale do letáčku nevešlo, tak se prosím podívejte do webové verze řešení.

Program (C++):
`http://ksp.mff.cuni.cz/viz/32-4-3NP.cpp`

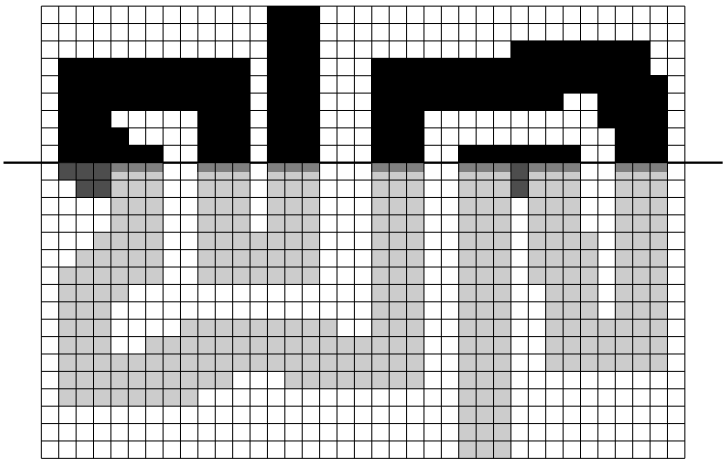
Jak řešit těžký problém koštětem

Dobrá, tak jsme dokázali, že naše úloha je těžká. Ale i přesto bychom ji chtěli vyřešit – ostatně, spousta úloh, které v životě potkáme, je těžká. Vzhledem k okolnostem se spokojíme s exponenciálním algoritmem. Budeme předpokládat pevnou šířku sběrnice, nalezení maximální šířky dořešíme binárním vyhledáváním.

Jistě bychom mohli pro každý možný začátek sběrnice na horní straně pole vyzkoušet všechny možnosti, jak sběrnici mezi součástkami proplést. Prohledávali bychom do hloubky a značili si ty části pole, které už jsou sběrnicí obsazené. Kdykoliv bychom se dostali do slepé uličky, tak bychom se z rekurze vrátili a použitá políčka zase prohlásili za volná.

Jak rychlé by to bylo? Pro zjednodušení předpokládejme, že pole je čtverec z $N \times N$ políček. Délka sběrnice jistě nepřekročí počet políček, tedy N^2. Na každém políčku se můžeme rozhodovat mezi až třemi variantami: pokračovat rovně, zabočit doleva, zabočit doprava. Takže všech možných tras sběrnice, které prozkoumáme, ne nejvýše 3^{N^2}.

To už je i na poměry exponenciálních algoritmů velmi pomalé. Proto zvolíme lepší strategii. Budeme pole zametat koštětem tvaru vodorovné přímky a udržovat si všechny možnosti, jak může vypadat sběrnice nad koštětem.



Uvažujme, jak je dolní část (pod koštětem) ovlivněna tou horní:

- Na některých místech z horní části vystupuje sběrnice. Těmto částem říkejme *konektory.*
- Občas se stane, že z horní části vyčuhuje kus sběrnice, který neodpovídá žádnému konektoru – na tato políčka se nejde napojit, ale jsou *blokovaná.*

Kombinaci těchto vlastností budeme říkat *druh.* A všimneme si, že pokud v korektní sběrnici vyměníme horní část za jinou téhož druhu, dostaneme opět korektní sběrnici. Přitom nezáleží na tom, který konektor je propojený s kterým, protože v každém propojení konektorů povede nějaká cesta z horního okraje k dolnímu; nějaké další kusy sběrnice vytvoří nedosažitelné cykly, ale ty můžeme smazat. (Kdybychom to chtěli zdůvodnit pořádně: každý graf, jehož 2 vrcholy mají stupeň 1 a ostatní stupeň 2, vždy vypadá jako disjunktní sjednocení jedné cesty s libovolným počtem kružnic.)

Z toho plyne, že si místo všech možných horních částí stačí zapamatovat jen všechny možné druhy a pro každý druh jednu variantu horní části.

Algoritmus začne s koštětem na horní hraně pole. Tam jsou možné přesně ty druhy, ve kterých je právě jeden konektor a žádná blokovaná políčka.

Nyní uvažujme, co se stane při posunutí koštěte o řádek níže. U každého konektoru si můžeme vybrat, jak z něj bude sběrnice pokračovat:

- Můžeme ji prodloužit o jeden řádek, takže získáme opět konektor.
- Můžeme ji prodloužit vodorovně doleva nebo doprava. Pak ji necháme pokračovat dolů (čímž vznikne konektor), nebo nahoru (takže se napojí na jiný konektor z horní části). Přitom:
  - Nesmíme narazit na žádné blokované políčko.
  - Nesmíme zakrýt konektor (tehdy by sběrnice, která z něj vede, neměla kudy pokračovat).
  - Tam, kde pod vodorovnou částí nevznikne konektor, musíme zablokovat políčka.
- Konečně můžeme vyrobit novou vodorovnou část sběrnice se dvěma konektory na okrajích. Platí tatáž omezení jako výše.

Samozřejmě nedovolíme sběrnici pokračovat na políčko, kde už leží nějaká součástka.

Jeden krok algoritmu tedy uváží množinu všech možných druhů části nad koštětem, pro každý z nich vygeneruje všechna možná pokračování, a tak získá novou množinu druhů pro koště posunuté o řádek níž.

Nakonec se koště zastaví na dolní hranici pole. Ve výsledné množině druhů zkusíme najít takový, v němž je jediný konektor (výstup sběrnice z pole) a žádná blokovaná políčka (sběrnice nesmí zasahovat pod dolní hranici pole). Ten odpovídá korektnímu průchodu sběrnice od horní hranice ke spodní.

Analýza složitosti

Jakou složitost má řešení s koštětem? Zatím ji odhadneme jen velice zhruba vzhledem k parametrům N (délka hrany pole) a k (šířka sběrnice). Složitost jistě nebude větší než součin následujících hodnot:

- počet možných poloh koštěte (to je $N+1$, čili $\mathcal{O}(N)$)
- počet možných druhů horní části (označme D)
- počet možných pokračování jednoho druhu (označme V)
- čas na zpracování jednoho pokračování (Sestrojíme odpovídající druh a zařadíme ho do nějaké hešovací tabulky, což stihneme v $\mathcal{O}(N)$. Ještě si musíme zapamatovat, jak vypadá sběrnice nad koštětem, ale na to nám stačí uložit nově přidaný řádek a odkaz na předchozí stav sběrnice nad koštětem, což je též $\mathcal{O}(N)$ a na závěr z toho jistě dokážeme celou sběrnici rekonstruovat.)

Složitost tedy shora odhadneme $\mathcal{O}(NDVN) = \mathcal{O}(N^2DV)$. Zbývá odhadnout D a V.

Nejprve V. Každé pokračování je jednoznačně určeno tím, kde začínají a končí vodorovné části sběrnice (konektory, které se nenapojí na vodorovné částí, přirozeně pokračují; nové konektory jsou zdola na těch koncích vodorovných částí, kde se nenapojil konektor shora). Pokračování tedy můžeme popsat posloupností N nul a jedniček, kde jedničky budou na pozicích začátků a konců vodorovných částí. Počet pokračování je tedy menší nebo roven počtu nula-jedničkových posloupností délky N, takže $V \leq 2^N$.

Komplikovanější je to s D. Polohy konektorů a blokovaná políčka můžeme popsat tak, že projdeme koště zleva doprava a pro každou vodorovnou souřadnici zapíšeme jeden symbol:

- `K` – zde leží levý okraj nějakého konektoru;
- 1 až $k-1$ – příslušný počet políček pod koštětem je blokovaný (to se vylučuje s konektorem);
- `0` – sběrnice tudy neprochází.

Sestrojili jsme tedy posloupnost N symbolů z $(k+2)$-prvkové abecedy. Všech takových posloupností je $(k+2)^N$.

Dosazením do výrazu pro složitost celého algoritmu dostáváme:

$$\mathcal{O}(N^2 \cdot (k+2)^N \cdot 2^N) = \mathcal{O}(N^2 \cdot (2k+4)^N),$$

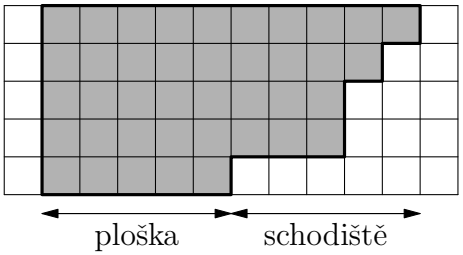
což leží v $\mathcal{O}((2k+5)^N)$. To je určitě mnohem lepší než $\mathcal{O}(3^{N^2})$ z řešení hrubou silou, ale pořád není hezké, že základ exponenciály závisí na šířce sběrnice k.

Ta část výpočtu, která závisela na k, počítala možnosti blokování políček. Ukážeme, jak totéž odhadnout těsněji, byť to dá víc práce. Opět budeme vytvářet nějakou posloupnost N symbolů popisujících zleva doprava situaci pod koštětem.

Blokovaná políčka rozdělíme na *úseky*. Každý úsek bude souvislý, ale ne nutně maximální souvislý – dva úseky se mohou dotýkat. Přitom konektory budeme považovat za políčka blokovaná do hloubky $k-1$. Každý úsek zakódujeme zvlášť, prostor mezi úseky vyplníme nějakými „mezerami“.

Blokovaná políčka vznikají pod novými vodorovnými kusy sběrnice, případně je zdědíme z předchozí polohy koštěte (přičemž se hloubka blokování sníží o 1).

Dokážeme, že úseky můžeme zvolit tak, aby každý vypadal následovně: Jeho základem je *ploška* šířky aspoň k (což je několik políček vedle sebe blokovaných do stejné hloubky, přičemž konektor považujeme za blokaci hloubky $k-1$). Na plošku buď vlevo, anebo vpravo navazuje *schodiště* políček, jejichž hloubka blokování neostře klesá.



Konstrukci úseků provedeme indukcí podle pohybu koštěte. Samotné posunutí koštěte a prodloužení konektorů sníží všechny hloubky blokování o 1, takže platnost tvrzení zachováme. Může ji změnit vznik vodorovného kusu sběrnice. Všimneme si, že dříve blokovaná políčka mohou k vodorovnému kusu přiléhat pouze zvenku (jinak by kus procházel blokovanými políčky). Rozebereme případy podle toho, jak je vodorovný kus napojen na konektory:

a) Oběma konektory nahoru. Tehdy vznikne alespoň $2k$ políček blokovaných do hloubky $k-1$, zleva a zprava k nim možná přiléhají nějaká dřívější schodiště. Nová políčka můžeme rozdělit na dvě plošky, každá si vezme schodiště na své straně.

b) Jedním konektorem nahoru a druhým dolů. Na straně konektoru nahoru můžeme zdědit schodiště, u konektoru dolů nemohlo žádné být. Proto můžeme konektor dolů prohlásit za novou plošku a zbytek za schodiště.

c) Oběma konektory dolů: v tom případě nedědíme žádné schodiště. Každý konektor se stane novou ploškou, políčka mezi konektory rozdělíme na dvě schodiště (poněkud rovná, připouštíme) u těchto plošek.

Zbývá dokázat, že úsek délky $\ell \geq k$ dokážeme zakódovat do ℓ symbolů. Nejprve řekneme, zda schodiště pokračuje doleva nebo doprava. Pak pro každou hloubku h od 1 do $k-1$ určíme počet p_h políček s touto hloubkou (vzhledem k monotonii schodiště je tím jednoznačně určeno, která políčka to jsou).

Kód bude vypadat takto:

- jeden bit, který říká, zda je schodiště vlevo nebo vpravo;
- p_{k-1} nul a jedna jednička;
- p_{k-2} nul a jedna jednička;
- $\ldots$
- p_1 nul a jedna jednička.

Kód je dlouhý $k + p_1 + p_2 + \ldots + p_{k-1}$. Ovšem $k \leq \ell$ a součet všech p_i je roven ℓ, takže délka kódu činí nejvýše 2ℓ. Doplníme ho nulami na délku přesně 2ℓ. To je dvakrát tolik, než potřebujeme, takže si pořídíme 4 symboly a každý bude kódovat 2 bity. Pak bude mít celý kód délku přesně ℓ.

Všechny úseky tedy dohromady zakódujeme do řetězce délky N nad 5-prvkovou abecedou (pátý symbol je mezera mezi kódy úseků). Takže všech možností blokování políček je maximálně 5^N.

Druh je ovšem určený ještě polohami konektorů. Ty zapíšeme nezávislým N-bitovým řetězcem, který nám bude říkat, kde jsou začátky konektorů. To dává 2^N možností. Celkem tedy víme, že $D \leq 5^N \cdot 2^N = 10^N$.

Časová složitost celého algoritmu pak vyjde $\mathcal{O}(N^2 \cdot 10^N \cdot 2^N) = \mathcal{O}(N^2 \cdot 20^N) \subseteq \mathcal{O}(21^N)$. To už je poctivý exponenciální algoritmus.

Dodejme ještě, že i tento horní odhad má stále obrovskou rezervu a možných druhů (tím spíš těch, které se doopravdy vyskytnou v konkrétním výpočtu) je mnohem méně. Proto by algoritmus byl pro rozumně malé vstupy docela dobře použitelný.

Jiří Kalvoda, Martin Mareš, Zuzka Urbanová

32-4-4 Zpětný signál

Tato grafová úloha je téměř hledání nejkratší cesty, jen ta cesta musí začínat a končit stejným vrcholem a musí vést přes nejméně jeden další vrchol. Hledáme tedy nejkratší „okruh“ s počátkem a koncem v daném vrcholu. Nabízí se tedy použít nějaký algoritmus na hledání nejkratších cest a upravit ho pro tento případ. My budeme pracovat s Dijkstrovým algoritmem, jehož popis najdete v kuchařce o haldě a nejkratších cestách.[5]

Kdybychom hledali cestu z počátečního vrcholu (řídícího panelu), kterému budeme dále říkat u, do sebe sama, algoritmus by ve své základní podobě našel jen onu skutečnou nejkratší cestu o délce nula hran. My ale chceme, aby cesta vedla přes aspoň jeden jiný vrchol. Víme tedy, že jistě povede přes nějakého souseda u. Jako sousedy u budeme nazývat takové vrcholy, do který z u vede hrana. Pokud z nějakého vrcholu vede hrana do u, ale neexistuje hrana z u do tohoto vrcholu, tak ho za souseda nepovažujeme (pamatujte, že graf je orientovaný).

Provedeme následující trik: Dijkstra nebude hledání nejkratších cest začínat z vrcholu u, nýbrž ze všech jeho sousedů. Tím se dostáváme k prvnímu řešení: pro každého takového souseda zvlášť spustíme Dijkstru, k nalezené nejkratší cestě ze souseda do u přidáme na počátek hranu z u do souseda (a tím vytvoříme hledanou okružní cestu) a ze všech takto nalezených cest vybereme tu nejkratší. Můžeme ovšem provést ještě jeden hezčí trik.

Vytvoříme v grafu nový vrchol v. Pro každého souseda u vytvoříme hranu z v do tohoto vrcholu, která bude mít stejnou délku jako obdobná hrana z u. Poté najdeme klasickým Dijkstrou nejkratší cestu z v do u a tato cesta bude naším hledaným okruhem, jen její počáteční vrchol v nahradíme za u. Jelikož sousedé u jsou stejní jako sousedé v, jistě se bude jednat o validní cestu. Protože jsme vynutili, aby tato cesta vedla přes nějakého souseda u, jedná se o validní okruh a ze správnosti Dijkstrova algoritmu víme, že bude nejkratší možný.

Zbývá rozmyslet časovou složitost. Jelikož graf se přidáním vrcholu v nezvětší víc než konstanta-krát, vyjde asymptotická složitost stejná jako u Dijkstrova algoritmu. Ve verzi s haldou tedy $\mathcal{O}((n + m) \cdot \log n)$ pro graf s n vrcholy a m hranami.

Kuba Pelc

32-4-5 Svázání bouře

I když úloha nebyla označená jako kuchařková, dal se pro její řešení použít jen mírně modifikovaný algoritmus na hledání konvexního obalu z geometrické kuchařky.[6] Ale souvislost s konvexním obalem možná není na první pohled vidět, tak si řešení pojďme ukázat pěkně od začátku.

Zadání po nás chce, aby nebylo možné již žádný paprsek přidat. Může nás tedy napadnout na celou úlohu jít inkrementálně. Začneme s jedním emitorem a žádným paprskem. V každém dalším kroku si přidáme jeden nový emitor a napojíme jej na všechny emitory, na které to půjde.

Správnost takového inkrementálního řešení se dokazuje velice snadno. Pokud by šel na konci přidat paprsek mezi dva emitory, pak šel určitě přidat i v okamžik přidání druhého z těchto emitorů. Nicméně pokud bychom to celé dělali v pořadí, ve kterém jsou emitory na vstupu, museli bychom rozebírat několik různých případů.

Jednak se může stát, že nový bod leží již uvnitř trojúhelníku ohraničeného třemi paprsky (v každém vrcholu trojúhelníku je jeden emitor). V takovém případě bychom ten nový potřebovali napojit na všechny tři vrcholy a trojúhelník tím rozdělit na tři menší. Dále teoreticky může emitor ležet přímo na nějakém již existujícím paprsku, nebo zcela mimo zatím nakreslený útvar.

Poznávat rychle ve kterém trojúhelníku nový bod leží je pracné – obzvlášť, když tuto strukturu neustále měníme přidáváním nových trojúhelníků. Tak se zkusíme obejít bez toho. Emitory nejprve seřadíme třeba podle x-ové souřadnice. Budeme chvíli předpokládat, že žádné dva emitory tuto souřadnici nemají stejnou. Přidáváním v tomto pořadí zařídíme, že každý nový bod bude ležet vždy mimo mnohoúhelník ohraničující dosud zpracované emitory. Vystačíme si pak s tím, že nový emitor jen propojíme se všemi emitory na současném obvodu útvaru, na které jej napojit lze, a obvod následně aktualizujeme.

A jak bude vypadat celý algoritmus? Nejprve všechny emitory seřadíme a v tomto pořadí – říkejme tomu třeba zleva doprava – je budeme následně procházet. V průběhu si budeme udržovat dva zásobníky, které budou obsahovat všechny vrcholy na horní respektive dolní části současného (konvexního) obalu. Inicializaci provedeme pomocí prvních dvou emitorů – propojíme je a oba přidáme do obou zásobníků.

Každý nový emitor vždy nejprve napojíme na nejpravější z dosud zpracovaných emitorů – ten najdeme na vrcholu obou zásobníků. A dokud je možné emitor napojit i na emitor těsně pod vrcholem zásobníku (aniž by propoj křížil současný konvexní obal), napojíme je a následně ze zásobníku nejvrchnější emitor odebereme. Jakmile už nový emitor

[5] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

[6] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

nemůžeme na žádný další napojit, přidáme jej na oba zásobníky a pokračujeme s další iterací.

Jak bude takové řešení rychlé? Kvůli třídění určitě nebude rychlejší než $\mathcal{O}(n \log n)$. Nebude však ani pomalejší. Sice provedeme n iterací a některé iterace mohou vyžadovat až n kroků – to v případě, že budeme napojovat emitor na spoustu emitorů na zásobníku. Pokud však budeme emitor napojovat na více emitorů ze zásobníku, budeme přitom zásobník zmenšovat. V součtu přes všechny iterace těchto napojení může být jen lineárně mnoho. Paměťová složitost celého algoritmu je lineární.

Na závěr jen doplníme vynechaný detail: co dělat, když mají dva emitory stejnou x-ovou souřadnici? Myšlenkově je asi nejsnazší celou rovinou bodů otočit o malinkatý nenulový úhel, který zařídí, že body již nebudou přesně nad sebou. Úhel ale zvolíme tak malý, aby se žádné dva body nad sebe naopak nedostaly (nebo si dokonce neprohodily pořadí). Možná to na první pohled zní celé složitě na implementaci. V praxi je to ale velice jednoduché. Efektu pootočení o malinkatý úhel dosáhneme přesně tím, že budeme body řadit primárně podle x a sekundárně podle y.

Jenda Hadrava

32-4-6 Sudoku s koněm

Na tuto soutěžní úlohu šlo jít mnoha způsoby. Například vzorový příklad v zadání byl vytvořen ručně umístěním sedmi devítek po kterých se dalo skákat jezdcem, do prázdného sudoku. Toto sudoku jsme doplnili prvním solverem, co byl po ruce, a pak našli procházku, která během prvních 9 kroků prioritizovala vysoká čísla a pak už jen dotvořila nějakou validní cestu. To zřejmě nebylo příliš efektivní, jelikož se dá najít lepší skóre s vyšší hodnotou už na desáté pozici.

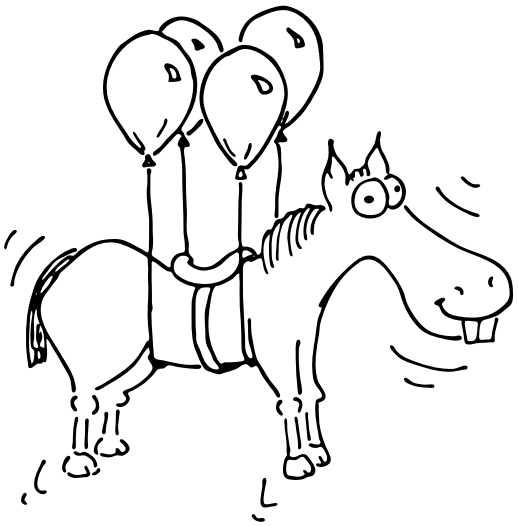
Pro řešení se nabízí dva postupy, které ale nejsou příliš efektivní. Prvním je zafixovat vyřešené sudoku a v něm hledat co nejlepší procházky. Druhým, opačným postupem je zafixovat si nějakou procházku a hledat pro ni co nejlepší sudoku. Toto je vhodné opakovat, jelikož v obou případech můžeme zjistit, že naše zafixovaná část neumožňuje získat velké skóre.

V prvním případě zřejmě budeme chtít začínat cesty v políčky s devítkami a držet se co nejdéle vysokých čísel. Pak ještě musíme procházku dotvořit, což nemusí být snadné a pokud jsme si počátek cesty zvolili nešikovně, tak to ani nepůjde. V případě se zafixovanou procházkou zase naopak hledáme sudoku, co mají na začátku procházky políčka s vysokou hodnotou.

Vytváření sudoku i procházky najednou

Efektivnějším přístupem může být spojit toto dohromady a hledat zároveň procházku i sudoku. Vybereme si nějaké počáteční políčko, o tom, která jsou pro to vhodná, si ještě povíme o pár odstavců dále. Pak hledáme cestu jezdcem, při které můžeme pokládat políčka s co nejvyšší hodnotou na sudoku bez toho, abychom porušili omezení na unikátnost hodnot v jeho řádcích, sloupcích a čtvercích. Tím jsme vlastně vytvořili docela složitý graf, jehož vrcholy obsahují jako stav pozici, už navštívená pole a umístěné hodnoty na sudoku.

V tomto grafu můžeme hledat cestu, která hladově maximalizuje skóre. Bohužel si ale docela často musíme vybírat z různých políček, kam se posuneme při zachování stejného skóre. Později pak budeme muset spoustu rozpracovaných cest zařízout a vrátit se zpět, když zjistíme, že už se cesta nedá dotvořit. Tento graf je tak velký, že se nedá v rozumném čase projít prozkoumat celý. Můžeme jej ale prozkoumávat s využitím náhody. Například si můžeme zvolit pravděpodobnosti pro zařízutí cesty, když nám dlouho trvá najít její rozšíření. Takto většinou získáme několik parametrů, které se také snažíme optimalizovat podle toho, jak dobrá řešení nám s jejich pomocí vychází.



Počáteční políčka

Pojďme se podívat na hledání procházky pro jezdce. Pokud jste neměli štěstí, tak je možné, že jste nemohli z nějakého začátečního políčka nemohli najít cestu, která by navštívila všech 81 políček, všechny cesty měly délku jen 80. Pro důvod se podívejme na jezdcův graf, kde vrcholy jsou políčka a hrany jsou skoky, po kterých se jezdec může vydat. Tento graf je bipartitní, a jelikož má 81 vrcholů, tak jedna partita musí být větší než ta druhá. Když začneme naši cestu v té menší partitě s 40 políčky, tak se nedokážeme dostat do jednoho z políček ve větší partitě, jelikož si dříve zablokujeme vrcholy přes které se tam dá dostat. Do té větší partity patří políčka, jejichž součet souřadnic $x + y$ je sudý. To platí nezávisle na tom, jestli indexujeme políčka od 0 nebo 1 (pokud to děláme pro oba rozměry stejně) i na tom, z kterého rohu sudoku začínáme počítat – rozmyslete si to.

Hledání jezdcovy procházky

Už samotné nalezení procházky může dát docela zabrat. Za zmínku nám přijde pravidlo, které vymyslel H. C. von Warnsdorff už v roce 1823: vždy se posunout na políčko, ze kterého má jezdec nejméně dalších možných pohybů. Tato heuristika umožňuje zásadně zmenšit prohledávaný prostor, využít ji můžeme například když máme na výběr z více políček, kam můžeme umístit stejnou hodnotu. V našem případě ale vůbec nemáme zajištěno, že se tímto způsobem dá najít dobrá řešení. Mohlo by se stát, že je nejlepší volbou vybrat pole, ze kterého je možností, kam se dál vydat, nejvíce. Opět se nabízí aplikovat heuristiku jen s nějakou pravděpodobností.

Optimální řešení

Všechna řešení, co nám přišla, nalezla různými metodami stejné skóre, sudoku a procházku, což nám zásadně zjednodušilo bodování. Teď zbývá jen otázka, zda je toto řešení nejlepší možné. Jde o následující skóre:

```
999999988988889778777667567565466544334623554151322511481156324132322341613748 22
```

V další části si ukážeme, že toto je optimální řešení. Lepší skóre získat nejde. Následující sudoku a procházka jsou až na symetrii jediným řešením s tímto skóre.

```
8 2 7 4 1 9 6 3 5
5 9 4 2 6 3 1 8 7
3 1 6 5 8 7 4 2 9
2 6 9 1 3 5 8 7 4
1 4 3 8 7 6 5 9 2
7 8 5 9 4 2 3 1 6
6 7 2 3 5 8 9 4 1
4 5 8 7 9 1 2 6 3
9 3 1 6 2 4 7 5 8
```

```
56 69 24 39 54  9 22 37 52
25  0 55 68 23 38 53  8 21
70 57 40 43 10 19 36 51  6
41 26  1 72 67 44  7 20 35
58 71 42 11 18 73 34  5 50
27 12 59  2 45 66 49 74 33
60 15 62 65 30 17  4 77 48
63 28 13 16  3 46 79 32 75
14 61 64 29 80 31 76 47 78
```

Jak jsme na to přišli a jak takovou věc ověřit? Pokud bychom aplikovali výše zmíněný algoritmus s tvořením procházky i sudoku najednou, bez heuristik, bez pravděpodobností, a našli první řešení, tak víme, že je optimální. Už jsme ale zjistili, že graf, který bychom museli prozkoumat, je obrovský. Museli bychom prozkoumat všechny větve, když si můžeme vybrat z více políček se stejnou hodnotou. Popřípadě nějak dokázat, že některé větve nejsou potřeba.

To jsme neudělali a nevíme, jak dlouho by to trvalo upočítat. Ukážeme si místo toho obecnější přístup, který u této úlohy fungoval krásně.

SAT

*Pokud jste nikdy neviděli zapsané logické spojky $\wedge$, $\vee$ a $\neg$ (*AND, OR *a* NOT*), doporučujeme si o nich rychle skočit něco přečíst. Víc jich potřebovat nebudeme.*

Nejdříve si musíme představit problém splnitelnosti booleovské formule často zvaný zkratkou SAT, z anglického satisfiability. Pro logickou formuli chceme zjistit, jestli se dají proměnné nastavit na takové hodnoty TRUE nebo FALSE, popřípadě 1 a 0, aby byla formule splněna. Najdete o něm zmínku v naší kuchařce o těžkých problémech, jelikož je to problém, co neumíme řešit v polynomiálním čase. Jde o takzvaný NP-úplný problém. (Také si zahrál důležitou roli v řešení úlohy 32-4-3. Tamtéž najdete příklad SATu.)

To ale neznamená, že se nedá napsat program, který tento problém řeší, i když nepoběží v polynomiálním čase. Takové programy se nazývají SAT solvery. Pro předhozenou úlohu SATu vám typicky solver umí odpovědět, jestli se dá formule splnit, nebo ne. K tomu umí najít **jedno** řešení, popřípadě certifikát nesplnitelnosti – posloupnost kroků, kterými lze logicky odvodit, že to nejde.

SAT solverů existuje velké množství a jde o aktivní oblast výzkumu. Jedním z motivátorů je soutěž SAT Competition,[7] kde jsou každoročně porovnávány SAT solvery hned v několika kategoriích. Kromě jedné kategorie je zároveň požadováno, aby byly jejich zdrojové kódy dostupné pro výzkum. Další výhodou je, že všechny podporují jednoduchý vstupní a výstupní formát DIMACS, a velká část se dá také použít jako knihovna. Díky tomu máme na výběr z efektivních solverů, implementovat SAT solver si tedy sami nebudeme.

Na SAT dá převést prakticky jakákoliv úloha, na kterou máme odpověď ANO/NE. To ale naše úloha není, hledáme maximální skóre. Budeme si muset úlohu rozdělit na podúlohy. Budeme opakovaně pokládat otázku, jestli existuje řešení, které má skóre začínající na nějaký prefix.

Začneme s prázdným skórem a budeme postupně zkoušet přidat na konec číslice, od největší možné. Když zjistíme, že takové řešení existuje, tak máme nový maximální prefix. Když zjistíme, že neexistuje, zkusíme číslici o jednu nižší. Jakmile rozšíříme prefix na všech 81 znaků, víme, že máme maximální splnitelný prefix. Jediné, co potřebujeme, je ta magická krabička, která nám říká, jestli existuje řešení začínající prefixem. A tu nám tady poskytne SAT solver. Zbývá tedy zakódovat úlohu do SATu.

Převod na SAT

Použijeme vcelku jednoduché zakódování do formule ve tvaru CNF, což je tvar, který podporuje většina solverů. Budeme muset do logických formulí zakódovat podmínky na číslice sudoku, na tvar procházky jezdce a na prefix skóre.

CNF nás omezuje na libovolný počet klauzulí sestavených z disjunkcí literálů (proměnných či jejich negací). Tyto klauzule jsou pak spojeny konjunkcemi. Příklad výroku v CNF s dvěma klauzulemi:

$$(i \vee \neg j \vee \cdots \vee k) \wedge (\neg i \vee j).$$

Pořídíme si hned devět logických proměnných $h^v_{(x,y)}$, kde $v \in \{1, \ldots, 9\}$, pro každé políčko (x, y) sudoku. Proměnná bude mít hodnotu TRUE, pokud na políčku bude ležet daná číslice. Protože zřejmě chceme, aby každé políčko mělo právě jednu číslici, musíme si na to přidat podmínky. Podmínky budeme definovat jako disjunkce proměnných či jejich negací a budeme chtít, aby všechny platily – tak získáme nakonec výrok v CNF.

To, že je alespoň jedna číslice zvolena, se zapíše jednoduše:

$$h^1_{(x,y)} \vee h^2_{(x,y)} \vee \cdots \vee h^9_{(x,y)}.$$

Zapsat, že je zvolena maximálně jedna číslice, jen pomocí disjunkcí je složitější. Pomůžeme si trikem: pro každou dvojici hodnot může být nastavená jen jedna z nich:

$$\forall a \neq b \in \{1, \ldots, 9\} : \neg h^a_{(x,y)} \vee \neg h^b_{(x,y)}.$$

Toto je obecná konstrukce, kterou použijeme ještě několikrát. Umožňuje nám zajistit, že je nastavená maximálně jedna z několika proměnných. Už se nebudeme vracet k tomu, jak to přesně zapsat.

Nyní tu samou konstrukci použijeme na zajištění, že v každém řádku, sloupci a čtverci sudoku je nastavena maximálně jedna jednička, jedna dvojka atd. Jelikož je políček právě devět, tak to stačí zapsat takto. Pokud teď pustíme SAT solver na to, co už máme zapsané, tak nám velmi rychle odpoví, že to je řešitelné, a dá nám ohodnocení odpovídající jednomu vyplněnému validnímu sudoku. Doporučujeme si při psaní podobných převodů průběžně zkoušet, zda fungují. Je celkem jednoduché tady udělat chybu a zkoušet to průběžně je snazší.

[7] `http://www.satcompetition.org/`

Teď je čas se věnovat pohybu jezdce. Použijeme megalomanský přístup, a pro každé políčko sudoku si vyrobíme 81 proměnných $j^p_{(x,y)}$, kde $p \in \{1, \ldots, 81\}$, které budou značit, kdy tam jezdec dorazil. Počítáte správně, to je 6561 proměnných, spolu s těmi 729, co už máme definované, to tvoří celkem 7290 proměnných. To vypadá jako děsivý počet, ale SAT solvery si s tím ještě poradí. To, že se proměnné navzájem vylučují, solverům dost pomáhá.

Opět přidáme podmínky pro omezení počtu hodnot, stejným způsobem jako výše. Pro každé políčko (x, y) bude TRUE právě jedna hodnota $j^p_{(x,y)}$, což nám implicitně vyřeší, že se na žádné políčko nedá vracet. Taktéž přidáme podmínky specifikující, že každá pozice p je právě na jednom políčku. Pokud bychom to neudělali, tak je validní řešení na každé políčko dát stejné pořadí návštěvy.

Chybí už jen přidat podmínku na omezení tvaru pohybu: skoky musí být pohyby jezdce. Pro každé políčko (x, y), pozici p, která není poslední, a jezdcovým skokem dostupná políčka $(x'_1, y'_1), \ldots, (x'_n, y'_n)$ přidáme následující podmínku:

$$\neg j^p_{(x,y)} \vee j^{p+1}_{(x'_1,y'_1)} \vee \cdots \vee j^{p+1}_{(x'_n,y'_n)}.$$

Tato podmínka je přímým přepisem implikace, že z políčka můžeme v příštím tahu jen na ta dosažitelná jezdcovým skokem.

Když vyřešíme problém SAT v současném stavu, dostaneme kladnou odpověď na splnitelnost a jedno validní řešení odpovídající vyřešenému sudoku a jezdcově cestě po něm. Můžeme si z něj spočítat skóre, ale téměř určitě nebude nejlepší možné. Chybí nám ještě poslední krok, totiž přidat podmínky na prefix skóre.

Vynucený prefix se dá zapsat velmi jednoduše. Pokud chceme omezit na pozici p hodnotu skóre pouze na číslici v, přidáme pro každé políčko (x, y) následující podmínku:

$$\neg j^p_{(x,y)} \vee h^v_{(x,y)}.$$

Tato podmínka vynucuje, že na pozici je buď vynucená hodnota skóre nebo že tam jezdec není.

Tím jsme hotovi. Tuto úlohu můžeme postupně předkládat s rozšiřovaným prefixem SAT solveru a on nám nejen najde jedno optimální řešení, ale dokonce to dokáže. Pro všechny prefixy, kterými by mohlo začínat vyšší skóre, máme totiž důkaz, že takové řešení neexistuje.

Optimalizace

Můžeme ještě provést několik optimalizací pro zrychlení hledání řešení. První je využití výše zmíněné vlastnosti s paritou políčka, kde se začíná. Nikdy totiž nenajdeme optimální řešení na políčku (x, y), kde $x+y$ je liché. Stačí přidat pro každé takové pole jednoprvkovou klauzuli $(\neg j^1_{(x,y)})$. I jednoprvková klauzule je validní v CNF, umožňuje nám nastavit proměnnou explicitně.

Další drobnou optimalizací je omezit symetrie, tím, že vynutíme nějak kanonicitu řešení. Sudoku s procházkou se dá symetricky otočit hned podle několik os, které zachovají skóre. Tím, že omezíme začátek na jednu diagonální polovinu (například podmínkou $x \leq y$), odstraníme některé. Pak můžeme přidat podmínky $y < 5$ a $x < 5$ (indexováno od nuly), tím omezíme vodorovnou i svislou symetrii. To ještě pořád neodstraní všechny symetrie, ještě zbývá případ, kdy je počáteční políčko na diagonále. V tom případě bychom museli ještě vynutit směr počátečního pohybu. Toto nejspíš už nejspíš moc nepomůže rychlosti solveru, ale hodí se to, když chceme zjistit, jestli jsou řešení jen symetrická, nebo nějak unikátní.

Unikátnost řešení

Ještě nám zbývá drobná otázka unikátnosti řešení. Už jsme si jedno ukázali, mohla by ale existovat další sudoku s procházkami, co mají stejné skóre. SAT solver sice neumí rovnou vydat všechna řešení, ale můžeme si pomoci drobným trikem. Vezmeme poslední problém, ten s prefixem obsahujícím už celé optimální skóre. Když nám SAT solver vydá nějaké řešení, tak k tomuto problému přidáme novou klauzuli, která je konjunkcí všech proměnných z řešení, s negací u proměnné, pokud je v řešení nastavená na TRUE.

Toto je podmínka, která vynutí, že se alespoň jedna proměnná z nalezeného řešení musí lišit od řešení, co jsme našli dříve. Toto můžeme opakovat s každým řešením, co nalezneme. Je ovšem možné, že touto úpravou získáme problém, který poběží výrazně déle. V případě našeho sudoku s koněm však vcelku rychle zjistíme, že optimální řešení existuje pouze jedno, až na symetrie.

Závěr

Naše vzorové řešení je programem, který implementuje právě to, co je popsáno výše. Zde je jeho celý výstup, který popisuje postupné rozšiřování prefixu skóre. Naše řešení využívá SAT solveru glucose[8] verze 4.1 zkompilovaného s podporou po paralelní řešení, abychom mohli využít vícejádrového procesoru. Vyměnit jej za jiný by mělo být jednoduché, zvláště díky tomu, že většina SAT solverů podporuje stejný vstupní a výstupní formát. Pro představu: celkový běh programu trval deset minut na normálním středně výkonném stolním počítači.

```
99999999 ...Not satisfiable [792.04ms]
99999998 ...OK [1.68s]
999999989 ...Not satisfiable [812.99ms]
999999988 ...OK [7.55s]
9999999889 ...OK [7.20s]
99999998899 ...Not satisfiable [779.70ms]
99999998898 ...OK [5.97s]
999999988989 ...Not satisfiable [800.42ms]
999999988988 ...OK [4.88s]
9999999889889 ...Not satisfiable [787.07ms]
9999999889888 ...OK [5.13s]
99999998898889 ...Not satisfiable [796.74ms]
99999998898888 ...OK [5.43s]
999999988988889 ...OK [7.43s]
9999999889888898 ...Not satisfiable [786.78ms]
9999999889888897 ...OK [6.87s]
99999998898888978 ...Not satisfiable [801.50ms]
99999998898888977 ...OK [3.49s]
999999988988889778 ...OK [8.06s]
...
99999998898888897787777667567565466544334623554151322511481156324132322341613748221
                                                    ...OK [851.94ms]
```

Celý výstup programu:
`http://ksp.mff.cuni.cz/viz/32-4-6-dukaz.txt`

Program (Rust):
`http://ksp.mff.cuni.cz/viz/32-4-6.rs`

Jirka Sejkora

[8] `https://www.labri.fr/perso/lsimon/glucose/`

Výsledková listina čtvrté série třicátého druhého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*4-1*	*4-2*	*4-3*	*4-4*	*4-5*	*4-6*	*série*	*celkem*
0.					9	10	12	10	13	15	60,0	240,0
1.	Jiří Kalvoda	GJarošeBO	3	9	9	8	10	10	13	15	57,4	241,8
2.	Kristýna Petrlíková	VOŠJičín	2	9	9	8	8	10		15	51,1	185,1
3.	Jan Adámek	GKepleraPH	3	4	9	10	9	10	12		52,3	180,6
4.	Ondřej Sladký	GMikulášPL	3	6	5	10	8	10	13		49,0	174,5
5.	Vladimír Chudý	G Chrudim	3	14	7	10	10	10	4		38,7	169,8
6.	Petr Borňás	GRoudnice	4	4	9		8	10	8		39,6	148,0
7.	Karel Chwistek	MendelGOP	3	5	3			10			14,7	131,7
8.	Václav Janáček	GJarošeBO	3	3							0,0	124,5
9.	Michal Bravanský	GBílovec	2	4				3			5,1	115,5
10.	Kateřina Vokálová	G Kolín	4	5		7	4	10			24,8	102,7
11.	Matej Štencel	GPošKošice	3	3							0,0	95,5
12.	Jiří Kvapil	GTomkovaOL	2	13	9	8		10			26,6	95,3
13.	Daniel Skýpala	GTomkovaOL	2	15							0,0	77,7
14.	Martin Hubata	GMikulášPL	4	5		5					6,9	77,4
15.	Ondřej Hráček	GOlgHavl	3	3							0,0	76,7
16.	Dominik Farhan	GMikulášPL	3	3	2	10					13,9	59,2
17.	Sebastián Svoboda	MendelGOP	3	2							0,0	53,9
18.–19.	Janek Hlavatý	GJirsíkaČB	1	4							0,0	50,9
	Michal Kodad	SPŠSmíchov	4	17	5	8		10		15	34,9	50,9
20.	Albert Kučera	GNadŠtolPH	3	3		8		7			18,1	47,8
21.	Jan Provazník	GVoděraPH	4	9		8		7			16,0	46,2
22.	Vít Skalický	GPísnickáPH	2	13				7			6,4	45,4
23.	Jan Piroutek	GŠpitálsPH	4	9				7			7,6	43,7
24.	Marie Kalousková	GNAlejíPH	4	5							0,0	35,7
25.	Lucie Vomelová	GŠpitálsPH	4	10				2			2,2	31,0
26.	Prokop Randáček	GFXŠaldyLI	1	2							0,0	29,7
27.	Ondřej Gonzor	G Brandýs	3	14							0,0	29,0
28.	Jiří Gallo	SPŠERožnov	3	1							0,0	28,1
29.	Denis Hromada	SŠIEŘ Rožnov	3	1							0,0	25,4
30.	Robert Jaworski	GÚstavníPH	2	2							0,0	23,4
31.	Martin Havelka	Gym Třeboň	2	2							0,0	23,3
32.	Darian Poljak	GJŠkodyPŘ	3	2							0,0	22,2
33.	Petr Šícho	GKepleraPH	2	1							0,0	20,2
34.	David Klement	GNAlejíPH	4	8							0,0	20,0
35.	Vojtěch Žák	GŠpitálsPH	4	8							0,0	18,9
36.	Vít Ulehla	GJŠkodyPŘ	3	1							0,0	18,8
37.	Oliver Tušla	GArabskáPH	3	2							0,0	16,2
38.	Kristýna Prokopová	GJosBožČT	4	3	2						3,9	13,9
39.	Lucie Kunčarová	GVolgogrOS	4	2							0,0	12,7
40.–41.	Jan Kaifer	GKepleraPH	4	14							0,0	12,0
	Adam Šlegl	GJosefskPH	3	1							0,0	12,0
42.	Antonín Otmar	GNadKavaPH	3	1							0,0	11,4
43.	Petr Hladík	GMikulášPL	2	1							0,0	11,2
44.	Klára Hloušková	G Kolín	4	3							0,0	10,1
45.–52.	Jan Klivan	GDačice	3	1							0,0	9,0
	František Kmječ	G Brandýs	4	11							0,0	9,0
	Stanislav Kozák	G Holice	2	1							0,0	9,0
	Jan Kučera	SOŠ Březová	3	2							0,0	9,0
	Klára Pernicová	GJarošeBO	3	1							0,0	9,0
	Timotej Toepfer	ArcibisGPH	3	1							0,0	9,0
	Kristýna Umlaufová	SPŠOstrov	3	1							0,0	9,0
	Vojtěch Zabořil	GTurnov	3	2							0,0	9,0
53.	Šimon Genčur	GBBr	0	1	1	3					8,5	8,5
54.	Jáchym Němeček	SPŠERožnov	3	1							0,0	8,1
55.–56.	Ondra Müller	GTurnov	3	3							0,0	7,8
	Tomáš Vesecký	SSŠVTPraha	3	3							0,0	7,8
57.	Robert Gemrot	GKomHavíř	3	3	5						7,1	7,1
58.	Šimon Andrš	GKepleraPH	1	1							0,0	6,7

	řešitel	*škola*	*ročník*	*sérií*	*4-1*	*4-2*	*4-3*	*4-4*	*4-5*	*4-6*	*série*	*celkem*
59.–63.	Jiří Bartošík	SUHr	3	1							0,0	4,4
	Jan Černohorský	G Brandýs	2	1							0,0	4,4
	Maria Filtsova	SŠInformFM	3	1							0,0	4,4
	David Maňásek	SUHr	3	1							0,0	4,4
	Daniel Šoltýs	GTřeKošice	2	1							0,0	4,4
64.–65.	Karel Bartůněk	GMilevsko	2	1							0,0	2,5
	Patrik Herman	GTomkovaOL	1	1							0,0	2,5
66.	Martin Klimeš	GZábřeh	4	2							0,0	2,4