Milí řešitelé a řešitelky!

Držíte v ruce pátý leták 24. ročníku KSP. Každá série letos obsahuje 8 úloh a z nich se 5 nejlépe vyřešených započítává do celkového bodového hodnocení.

Nově je možno být přijat na MFF UK za úspěšné řešení KSP. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok lze získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

V této sérii jsme k většině úloh zařadili jednodušší variantu – typicky jde o stejnou úlohu, kde máte navíc nějaké zjednodušující předpoklady. Pokud její řešení vymyslíte, neváhejte s odesláním a slíbené body vás neminou. Řešení jednodušší varianty by vás mělo navést správným směrem – zkuste se zamyslet, jestli by nešlo upravit, aby řešilo i variantu složitější.

Termín odevzdání páté série je stanoven na **pondělí 4. června** v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do čtvrtka 31. května, aby nám stihlo přijít. CodExová úloha má termín o den posunutý, protože nám ji opravuje automat – 5. června v 8:00.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

Pátá série čtyřiadvacátého ročníku KSP

„Dobrý den, pane, máte tu jedno rekomando. Prosil bych jeden podpis… výborně, pěkný den přeju!“

Zvláštní – doporučeně už mi delší dobu nikdo nic neposlal, vynechám-li soudní obsílky… Tohle ani nevypadá úředně.

Milý příteli,

už je tomu dlouho, kdy jsem se naposledy ozval. Nezapomněl jsem na Tebe – jen jsem měl poslední dobou hodně práce kvůli té naší chatě. Před časem jsi nám říkal, že se máme ozvat, až budeme potřebovat pomoc – tak Ti tedy píšu.

Chata už je skoro hotová, jen bychom potřebovali pomoc s jedním výkopem. Mohl by ses někdy stavit v jižních Čechách? Sešli bychom se na obvyklém místě, dopravu na chatu zajistím.

Měj se pěkně,

Edo

Hmm… Mohl by to být normální dopis. Nebýt toho, že žádného Edu neznám, a tím méně jeho chatu. Vyvolalo to ve mně značnou nostalgii. Je tomu už hodně dávno, co jsem dostal něco podobného – podobné šifry už dnes chodí zásadně oknem.

Ani nevím, kdo dostal ten báječný nápad používat ke komunikaci poštovní holuby. Klasickou poštu i telefony od nepaměti kontroluje StB. Veškeré zprávy jsme museli šifrovat velmi podivným způsobem, aby nebudily podezření. A dostat cokoliv za hranice bylo až donedávna prakticky nemožné.

To všechno se s příchodem poštovních holubů změnilo. Jsou podstatně rychlejší, než klasická pošta. Ale hlavně – StB není schopná je jakkoliv kontrolovat. Takže si můžeme dovolit zprávy posílat prakticky nešifrovaně. A od dob RFC 1149[1] *ani nemusíme řešit nejednoznačnosti datových paketů.*

I holubi však mají spoustu chyb – například jednosměrnost přenosu. Takový poštovní holub umí jen jednu věc – ať ho dovezete kamkoliv, vždycky trefí domů. Pokud potřebujete poslat zprávu někam jinam, máte prostě smůlu… anebo musíte použít prostředníka (nebo jiného holuba).

24-5-1 Holubí centrála — 9 bodů

Typickým problémem bývalo svolávání srazu. Sraz může vyhlásit libovolný člen organizace, jen musí zajistit, že se informace o času a místě dostane ke všem ostatním.

Vás by zajímalo, kteří členové mohou sraz vyhlásit. Dostanete seznam členů včetně poštovních holubů, které mají jednotliví členové u sebe. Každý poštovní holub má určeno, ke kterému členovi doletí. Máte vypsat ty členy organizace, od kterých vede spojení pomocí holubů ke všem ostatním. Takové spojení samozřejmě může vést přes prostředníky.

Pokud má například organizace 6 členů (s čísly 1 až 6), člen číslo 3 má holuby letící ke členům 1, 2 a 5, člen 5 holuby pro 3 a 6, člen 6 umí poslat zprávu členovi 4 a ostatní nemají žádného holuba, je správným řešením vypsat členy 3 a 5.

Naše ornitologické oddělení nedávno vymyslelo i efektivní broadcasting (všesměrové vysílání): stačí využít hejna labutí. Labutě jsou při přesunu dobře vidět. Navíc se vyskytují ve dvou barvách: černé a bílé.

24-5-2 Labutí broadcasting — 11 bodů

Zpráva pro broadcast se sestavuje následujícím způsobem: Nejprve ji převedete do posloupnosti nul a jedniček, poté seřadíte labutí hejno. Každá labuť odpovídá jednomu bitu. Pokud je bit nulový, zařadíte černou labuť; pokud je jedničkový, zařadíte bílou. Takto seřazené hejno poté vypustíte na oblohu a doufáte, že poletí správným směrem.

V labutím hejnu má první labuť nejtěžší úkol – rozráží vzduch. Proto se labutě postupně střídají. Vždy, když je první labuť unavená, zařadí se na konec hejna, přičemž vedoucí pozici převezme labuť za ní.

[1] `http://www.faqs.org/rfcs/rfc1149.html`

Ornitologické oddělení dosud nevymyslelo vhodný přenosový protokol; proto se obracíme na vás.

Máte vymyslet co nejefektivnější přenosový protokol – víte, že při poslání N bitů příjemci dorazí N stejných bitů, ale náhodně rotovaných. Když tedy odešlete 1101, tak může přijít 1101, 1011, 0111 a 1110.

Vymyslete, jak tímto způsobem odeslat zprávu o K bitech, aby na její zakódování bylo potřeba co nejméně reálně odeslaných bitů a stále byla jednoznačně dekódovatelná.

Příklad: Pro $K = 1$ je řešení triviální, vyšleme tu správnou jednu labuť. Pro $K = 2$ vyšleme bity tak, jak jsme je dostali, a druhý z nich zopakujeme. Tedy pokud chceme odeslat xy, tak odešleme xyy. Na zprávu délky 2 jsme tedy spotřebovali 3 bity. Pro $K = 3$ potřebujeme 5 labutí.

5 bodů dostanete, pokud vymyslíte efektivní protokol pro $K = 8$.

Nostalgie bylo dost. Asi bych nás měl trochu představit, když už jsem to nakousl... jsem členem jedné organizace, která má za svůj cíl postavit tajnou necenzurovanou telefonní linku z ČSSR do Rakouska – snažíme se vybudovat rozumné spojení se sítí EARN.[2] *Což se samozřejmě nelíbí vládě ani StB – vznikl by nekontrolovatelný komunikační kanál se zahraničním disentem, navíc podstatně rychlejší než holubi a labutě dohromady. Takže pracujeme tajně.*

Činnost organizace je pochopitelně časově i organizačně velmi náročná.

24-5-3 Struktura organizace — 11 bodů

Abychom minimalizovali riziko odhalení, rozhodli jsme se pro zvláštní organizační strukturu. Každý člen zná jen své podřízené a svého přímého nadřízeného, od kterého dostává rozkazy. Podřízených může být i víc, nadřízeného má každý jediného, s výjimkou právě jednoho velkého šéfa, jenž už nadřízeného nemá. Nikdo není nadřízeným sám sobě, a to ani nepřímo.

Do akce je posílána vždycky skupina členů. Ti mezi sebou potřebují komunikovat, proto skupina musí zůstat souvislá. To znamená, že po odeslání do akce musí každý člen být schopný odeslat zprávu všem ostatním. Zprávy se samozřejmě mohou předávat pouze mezi známými, tedy mezi podřízenými a nadřízenými.

Vás by zajímalo, kolika způsoby můžeme vytvořit libovolně velkou skupinu, kterou pošleme do akce.

Například pokud máme 3 zaměstnance, přičemž zaměstnanec číslo 3 je přímý nadřízený zaměstnanců 1 a 2, tak máme dohromady 6 možností, jak skupinku vytvořit: $\{1\}$, $\{2\}$, $\{3\}$, $\{1,3\}$, $\{2,3\}$, $\{1,2,3\}$. Zaměstnance 1 a 2 vyslat nemůžeme, protože pak by byli naprosto oddělení.

7 bodů dostanete, pokud úlohu vyřešíte pro strukturu tvořící úplný binární strom. Zde má každý dva nebo žádného podřízeného, navíc všichni bez podřízených „jsou si rovni“ – mají nad sebou stejný počet nadřízených.

Tentokrát to vyšlo na mě. Abych to nezakecal, to rekomando znamená zahájit stavbu, sraz ve městě, kde by chtěl žít každý. Zajištění dopravy znamená, že nemusím shánět bagr.

Tak už jen zabalit několik kilometrů kabelu a hurá na cestu!

Kdo jste někdy viděli sraz členů tajné organizace na veřejném místě, jistě dáte za pravdu, že to není nic jednoduchého. Nemůžete si prostě vzít transparent hlásající „Hledám své tajné kolegy!“ a stoupnout si doprostřed náměstí.

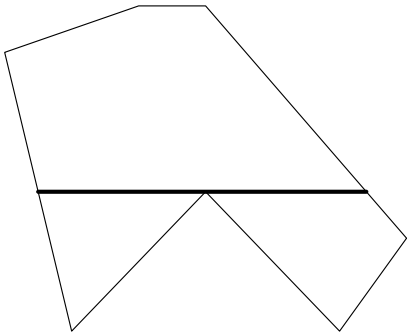
Místo toho je nutné mít předem domluvený způsob, jak se poznat. Samozřejmě dostatečně nenápadný. My většinou využíváme zeměpisných vlastností dané lokace.

Protentokrát jsme zvolili sraz na západním konci nejdelší úsečky vedoucí ve východozápadním směru, kterou je možné na náměstí najít. Mapu máme. Pomůžete nám s hledáním takové úsečky?

24-5-4 Sraz na náměstí — 11 bodů

Na vstupu dostanete (ne nutně konvexní) mnohoúhelník představující náměstí, zadaný například posloupností vrcholů. Máte vypsat nejdelší úsečku ve vodorovném směru, která je v mnohoúhelníku celá obsažena.

Příklad:



Tučně je vyznačena hledaná úsečka.

6 bodů dostanete, pokud vyřešíte úlohu pro konvexní náměstí.

Nakonec jsme se našli a snad nás přitom nikdo neviděl.

Na podobně dlouhých linkách se hodně projevuje rušení, zejména proto, že nemáme finance na dostatečně stíněné kabely – ty jsou moc drahé. Proto je občas nutné kabel přerušit a umístit stanici, která detekuje příchozí signál a předá ho dál.

Polohy těchto zesilovacích stanic jsou dány částečně technickými limity a rušením signálu, hlavně však tím, kde všude máme svoje lidi a elektřinu.

Řezání kabelů (a připojování koncovek) také není jednoduché. Pokud to jde, snažíme se kabely nařezat na příslušné délky pěkně v klidu někde v továrně.

24-5-5 Řezání kabelů — 9 bodů

Máte dlouhý kabel a chtěli byste ho co nejrychleji nařezat na kusy o délce $k_1, k_2, \ldots, k_n$. Kabel má celkovou délku $K = k_1 + k_2 + \ldots + k_n$. Je namotaný na cívce, před řezáním ho musíte celý odmotat a přeměřit. Při řezání rozdělíte jednu souvislou část kabelu na dvě menší o příslušných délkách. Odmotané kusy jsou dlouhé, takže je musíte ihned namotat na jinou cívku.

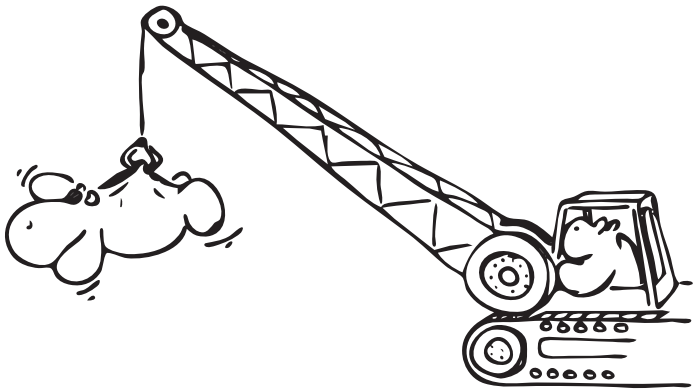
Nejvíce času zabere neustálé namotávání a smotávání, samotné řezání lze zanedbat. Každý řez tedy trvá tak dlouho, jaká je délka řezaného kusu kabelu.

Na vstupu dostanete počet úseků a jejich délky. Máte vypsat takové pořadí řezů, které zabere co nejméně času.

Příklad: Pro úseky délek 3 3 3 3 8 je optimálním řešením posloupnost řezů $20 \to 8 + 12$, $12 \to 6 + 6$, $6 \to 3 + 3$, $6 \to 3 + 3$.

[2] `http://en.wikipedia.org/wiki/European_Academic_Research_Network`

Po nařezání kabelů jsme se dali do stavby. Občas se nás místní ptali, co to vlastně děláme. Na podobné dotazy jsme připraveni – hlavně proto, že někdy provádíme neohlášené výkopy na cizích zahradách. Vždy se stačilo vymluvit na tajnou linku od Správy pošt a telekomunikací stavěnou pro armádu – tím jsme úspěšně odradili jak vojáky, tak „kolegy" od SPT. Majitele pozemků jsme typicky odbyli slovy „Když nesledujete vývěsní desku, tak se nedivte." Než si to stihli ověřit, už jsme byli pryč.



Brzy jsme dorazili k hraničnímu pásmu, tady si nemůžeme dovolit být tak drzí. Našli jsme jedno slabší místo, kudy se dostaneme zhruba kilometr od hranic bez jakéhokoliv rizika odhalení. Má to však jeden problém – po celé délce je minové pole.

24-5-6 Minové pole — 13 bodů

Taková typická hraniční mina má určenou oblast, kde detekuje pohyb – když se sem něco dostane, tak vybouchne a celou ji zničí. Miny byly pokládány do čtvercové sítě, navíc při výbuchu zničí pouze obdélníkovou oblast kolem sebe. Minové pole je obdélníkové.

Máte detektor kovů, víte tedy, kde se jaká mina nachází, a z jejich velikostí víte, jakou oblast daná mina kontroluje. Pro každé pole čtvercové sítě by vás zajímalo, kolik min vybouchne, když na něj šlápnete.

Na vstupu dostanete rozměry minového pole (počet řádků a počet sloupců čtvercové sítě: R a S) a seznam min spolu s oblastí, kterou daná mina kontroluje (zadanou levým horním a pravým spodním rohem).

Pokud obdélník začíná a končí na stejném řádku, resp. sloupci, tak je jedno políčko široký, resp. vysoký.

Vypište matici o rozměrech $R \times S$, kde je na pozici (i, j) uvedeno číslo udávající počet min, které vybouchnou při šlápnutí na toto pole.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[3] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

U prvních 5 vstupů bude zadané pole jednorozměrné – vyřešením získáte 7 bodů.

Explodující minové pole jsme úspěšně nechali za sebou. Vzniklé krátery se dají skvěle využít pro položení kabelu!

„Taky sis vzpomněl na hru Čtvercové bombardování?"

„Pst! Někoho slyším!"

Mezi námi a hranicí zbývá jen pohraniční stráž. Teď už se nevzdáme! Naštěstí máme na jejich velitelství své lidi a známe denní rozpisy hlídek – ukrýt se tak, aby nás nenašli, není těžké. Dokonce jsme zvládli i zamaskovat výkopy.

Kousek za hranicí nás už netrpělivě očekávali rakouští kolegové. Spojili jsme natažené kabely a pak nás kolegové odvezli do Linze na svou centrálu. Zároveň jsme morseovkou poslali prvních několik krátkých zpráv, abychom ověřili, že naše linka funguje.

Byla v pořádku! Rakouští kolegové okamžitě začali posílat informace, které se k nám jinak nedostanou.

Vypadá to, že fyzická část spojení je hotová. Ještě zbývá vyřešit softwarovou část, abychom mohli propojit počítače a zbavili se zdlouhavé práce telegrafistů. K tomu se nám bude hodit pomoc zkušeného odborníka.

24-5-7 Cesta přes hranice — 13 bodů

Odborník sídlí v německém Pasově. Potřebujete se k němu dostat a následně ho dopravit do Prahy. Cestou budete muset několikrát překročit hranice. To je menší problém, protože nemáte platný pas. Máte však několik výmluv, které můžete při průjezdu použít – abyste zabránili odhalení, můžete každou použít pouze jednou. Samozřejmě jich máte jen konečně mnoho a neměli byste jimi plýtvat, aby vám něco zbylo i na příště. Na druhou stranu si vás celníci zapamatují a při každém dalším průjezdu stejnou celnicí vás už kontrolovat nebudou.

Na vstupu dostanete mapu oblasti – seznam měst a cest mezi nimi, včetně vzdáleností. Dále u každého města víte, jestli je v něm celnice, nebo ne. Taky dostanete pozici Linze (zde začínáte), Pasova (tam se musíte zastavit) a Prahy (tam musíte skončit).

Nalezněte a vypište nejefektivnější cestu. Primárně se snažíte minimalizovat počet průjezdů celnicemi, sekundárně ujetou vzdálenost.

7 bodů získáte za vyřešení úlohy pro zapomnětlivé celníky. Ti si váš průjezd celnicí nezapamatují, takže při každém dalším průjezdu jejich celnicí musíte použít novou výmluvu.

Cestou do Prahy bylo jasně poznat, že se něco děje. Oblohu křižovala černobílá labutí hejna, noviny byly plné zahraničních informací a málem jsme srazili dva poštovní holuby.

Očividně si toho všimla i StB – tolik silničních kontrol jsme už hodně dlouho nepotkali. Ale je vidět, že absolutně netuší, co se stalo.

Povedlo se!

Radim „Rumcajz" Cajzl

24-5-8 Jak hraje deskovky počítač? — 15 bodů

Herní seriál se blíží ke svému konci a je třeba mu nasadit korunku. Po dvou dílech o matematických hrách a jejich řešení přinášíme díl o hrách mnohem složitějších, které jen tak na papíře vyřešit neumíme. Můžete si představovat například šachy, dámu, piškvorky pět v řadě nebo jinou deskovku pro dva hráče.

V první sérii[4] byl probrán algoritmus Minimax, v druhé[5] jeho vylepšení pomocí Alfa-beta ořezávání. Pak uběhla celá zima, během níž možná leckomu algoritmy v paměti roztály

[3] http://ksp.mff.cuni.cz/zaciname/codex.html
[4] http://ksp.mff.cuni.cz/viz/24-1-8
[5] http://ksp.mff.cuni.cz/viz/24-2-8

jako jarní sníh. Zopakovat oba by však bylo na dlouho, takže se budeme muset spokojit s Minimaxem. K pochopení dalšího textu a úkolu by nám měl stačit.

Strom hry a Minimax

Situace je následující: máme hru bez náhody a chceme najít z její určité pozice co nejlepší tah. Když se však podíváme na jednotlivé tahy, neumíme jednoduše určit, který povede k výhře a který ne. Proto budeme muset prozkoumat i pozice, do nichž vedou naše tahy, což provedeme rekurzivně (tím samým algoritmem).

V podstatě procházíme tzv. *herním stromem* – jeho kořenem je pozice, pro niž hledáme nejlepší tah, synové kořene jsou pozice vzniklé po jednom našem tahu, jejich synové jsou pozice po tahu soupeře atd. Listy stromu jsou buď pozice, kde jsme vyhráli, nebo pozice, v nichž vyhrál soupeř (na remízu na chvíli zapomeňme).

Nechť jsme prošli rekurzivně celý strom. Jak zjistit, který tah vede do pozice pro nás vyhrané? (To je taková pozice, v které při *dokonalé* strategii obou hráčů vyhrajeme my.) Pomůže nám k tomu *ohodnocování* uzlů stromu, čili pozic.

Listy ohodnotíme tak, že pro nás vyhrané pozice budou ∞ a pro soupeře $-\infty$. Ostatním vrcholům přiřadíme hodnotu, až když máme ohodnocené jejich syny. Pokud jsme na tahu my, vezmeme maximum z ohodnocení synů (tedy ∞ odpovídající naší výhře, pokud tam je), soupeř na tahu zase bere minimum.

V praxi nejsme většinou schopni propočítat celý herní strom (s výjimkou jednoduchých her nebo pozic v koncovce), proto je dobré prohledávání ukončit v určité hloubce (odpovídající počtu odehraných tahů z kořene).

Prohledáváním jen do určité hloubky však získáme listy, které pro nás nejsou vyhrané či prohrané. Ty musíme ohodnotit heuristickou funkcí, která bude pro danou pozici vracet, jak moc pravděpodobné je, že v ní vyhrajeme. Když je lepší pro nás, vrátí kladné číslo, když pro soupeře, vrátí záporné. Vyrovnaná nebo remízová pozice obdrží 0.

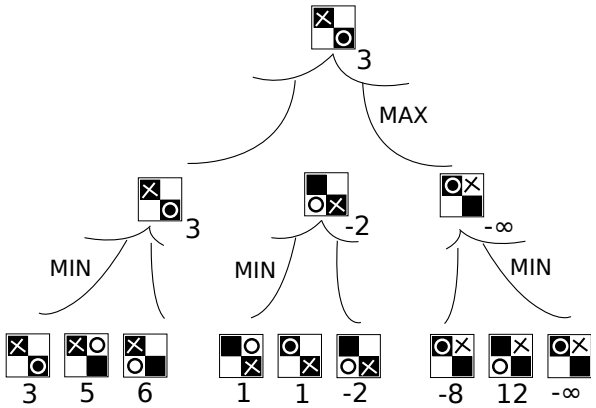
Pokud jsme na tahu, vybíráme maximum ze synů (hráči, který vybírá maximum, budeme říkat *Max*), soupeř vybírá minimum (a nechť se jmenuje *Min*), algoritmus se tedy nazývá *Minimax*. Zde je jeho pseudokód:

```
// funkce vrací hodnotu pozice a nejlepší tah
def minimax(pozice, hloubka, natahu):
  // jsme v listu
  if hloubka == 0 or konecHry(pozice)
    return (hodnota(pozice), prazdnyTah)

  if natahu == Max:
    nejHodnota = -nekonecno - 1
    nejTah = prazdnyTah
    // projdeme tahy hráče Max
    for p in mozneTahy(pozice, Max):
      (hodnota, tah) =
        minimax(provedTah(p), hloubka - 1, Min)
      if hodnota > nejHodnota:
        nejHodnota = hodnota
        nejTah = p
    return (nejHodnota, nejTah)

  if natahu == Min:
    nejHodnota = nekonecno + 1
    nejTah = prazdnyTah:
    // projdeme tahy hráče Min
    for p in mozneTahy(pozice, Min):
      (hodnota, tah) =
        minimax(provedTah(p), hloubka - 1, Max)
      if hodnota < nejHodnota:
        nejHodnota = hodnota
        nejTah = p
    return (nejHodnota, nejTah)
```

Pokud vám něco ohledně Minimaxu není jasné, nakoukněte do první série. Též přikládáme obrázek herního stromu prohledaného do hloubky 2.



Algoritmus lze zjednodušit tak, že pokaždé budeme vybírat maximum z hodnot synů, ale musíme pak mezi úrovněmi přenásobit hodnotu pozice číslem -1 a patřičně upravit hodnotící funkci. Zkuste si sami takto upravit pseudokód a ověřit, že dělá to samé. Zjednodušenému algoritmu se říká *Negamax* (násobení -1 je jakási negace a vždy vybíráme maximum).

Co se týče hodnotící funkce, měla by být velmi rychlá (rychlejší než prohledávání do hloubky o jedna větší s triviální ohodnocovací funkcí).

Minimax je sám o sobě dost neefektivní, protože zkouší všechny možné varianty, jak by hra dále mohla probíhat (i ty nesmyslné). Možným zrychlením výpočtu je proto negenerovat všechny tahy, což může být však mnohdy nebezpečné, protože lze přehlédnout dobrý tah… ale třeba u piškvorek přeskočení dobrého tahu zas tolik nehrozí, viz první sérii.

Algoritmus *Alfa-beta ořezávání* pak dostaneme z Minimaxu, když si všimneme, že některé uzly ve stromu mohou být pro jednoho z hráčů tak nevýhodné, že do nich určitě nebude hrát. Tyto části herního stromu tedy není potřeba prozkoumávat, mohou být tzv. *ořízntuty.*

Z časoprostorových důvodů odkážeme na podrobnější popis Alfa-beta ořezávání[6] do druhé série.

Transpoziční tabulky

Často se také stane, že k jedné pozici se lze dostat několika různými posloupnostmi tahů, je tedy v herním stromě víckrát. Aby se vždy nemusela znovu a znovu prozkoumávat, uloží se poprvé výsledek výpočtu do tzv. *transpoziční tabulky.*

Když tedy máme prozkoumat nějakou pozici, nejprve nahlédneme do transpoziční tabulky, není-li tam. Pokud ano a byla už prohledána do stejné hloubky, jako chceme, vrátíme uložený výsledek, jinak provedeme výpočet a pozici uložíme.

[6] http://ksp.mff.cuni.cz/viz/24-2-8

Transpoziční tabulka technicky není nic jiného než hešovací tabulka (o nich se více můžete dočíst v kuchařce o hešování).[7] Z pozice vytvoříme obrovské číslo (třeba 64-bitové), které by mělo být pokud možno unikátní – nazývá se heš pozice.

Heš modulo velikost tabulky udává, kam máme pozici uložit. Jelikož velikost transpoziční tabulky bývá o dost menší než rozsah hodnot heše a také než počet dosažitelných pozic, často se stane, že políčko v tabulce, kam chceme pozici uložit, je už obsazené.

Tento problém se může řešit různými způsoby, ale vždy se nějaká pozice z tabulky za určitých podmínek vyhazuje (jinak by program spotřeboval moc paměti). Nově ukládaná pozice bývá vždy uložena.

Nejčastěji se do každého políčka tabulky dávají dvě pozice, aby nedocházelo tak často k vyhazování. Když už jsou před ukládáním na políčku dvě pozice, vyhodí se z tabulky ta, jež byla prohledána do menší hloubky, což se musí ukládat v tabulce.

Toto samozřejmě není jediný způsob, jak se chovat, když je buňka v tabulce obsazena, ale bývá lepší než ukládání jedné pozice do jednoho políčka tabulky, jak ukázaly testy.[8]

Abychom ověřili, že máme na konkrétním políčku uloženu hledanou pozici, musíme v tabulce uchovávat i heše pozic. Takže celkově pro každou pozici budeme ukládat její heš, vypočtenou hodnotu, nejlepší tah a hloubku, do níž byla prohledávána.

Může se také stát, že dvě různé pozice dostanou stejnou heš. Aby se to stávalo co nejméně, musí být funkce počítající heš dostatečně náhodná a rozsah hodnot heše velký. Když však problém nastane, často nelze zahrát z pozice tah uložený v transpoziční tabulce. Jinak se tento problém většinou neřeší, jeho výskyt bývá řídký.

Zbývá jen povědět, jak počítat onu heš. Často se používá *Zobristovo hešování*. Před výpočtem si pro každou kombinaci herního políčka a herního kamene (figurky) vygenerujeme náhodnou hodnotu (v rozsahu heše). Heš konkrétní pozice je XOR hodnot kombinací políčka a kamene, jež se momentálně nacházejí na herní desce.

Tedy např. v šachách se heš může počítat takto: náhodné číslo pro bílou věž na A1 XOR číslo pro bílého jezdce na B1 XOR atd.

Význam transpoziční tabulky vzroste při použití *iterativního prohlubování*. Při něm prostě prohledávání pouštíme do hloubky 1, pak 2, 3, … , dokud nedojde čas nebo nezjistíme, že pozice je pro nás vyhraná či prohraná. Navíc při prohledávání upřednostňujeme nejlepší tahy z minulého prohledávání do menší hloubky (ty najdeme právě v transpoziční tabulce).

Dalších vylepšení Alfa-beta algoritmu je lidově řečeno hafo. Ostatní však ponecháme na dobrovolné samostudium, které se může hodit při řešení úkolu. Dobrým zdrojem může být Chess Programming Wiki.[9]

Úkol [14b]: Úkol spočívá ve zkoumání a analýze hry Dvonn, neboli jak by měl v takové hře počítač hledat z daného stavu nejlepší tah. Abyste se měli čeho chytit, dostanete návodné otázky. Odladěný program po vás chtít nebudeme, mohlo by vám to sebrat klidně celé jaro. :-)

Aby se vám hra dobře analyzovala, je možné ji hrát třeba na BoardSpace.net[10] (s lidmi i roboty). Pravidla najdete na internetu i v češtině[11] a soupeře si můžete domlouvat na našem fóru[12] (třeba autor seriálu si s vámi rád zahraje). Bohatě stačí, když se zamyslíte nad fází hry po rozmístění kamenů (tj. když už se kameny přemísťují).

Algoritmus na hledání nejlepšího tahu už znáte, pár triků také. Představte si, že chcete robota pro Dvonn implementovat ve svém oblíbeném jazyce, který by ovšem sám o sobě měl být rychlý (což třeba Python není, C# také moc ne). Jak efektivně reprezentovat pozici? Jak s pomocí té reprezentace rychle generovat a provádět tahy?

Zamyslete se rovněž nad ohodnocováním pozice. (Výhra bílého je nějaká velká konstanta H, výhra černého $-H$, remízová nebo vyrovnaná pozice má 0, vše ostatní je na vás.) I toto by mělo být pekelně rychlé. Namísto slovního popisu můžete dodat rozumně čitelný (pseudo)kód, což lze udělat i u jiných částí úkolu.

Dalším námětem může být řazení tahů dle výhodnosti pro hráče na tahu, které se hodí pro Alfa-beta ořezávání (lepší tahy spíše způsobí ořezání pozice). Jak lze v této hře řadit tahy? Dají se generovat rovnou v nějakém „dobrém“ pořadí?

Úkol je v podstatě dost kreativní a klidně napište i o něčem jiném, co vás při zkoumání hry a přemýšlení o algoritmech napadne, bude to náležitě oceněno.

Udělá nám radost (a vám bodově přilepší) samostudium algoritmů a jiných technik z této oblasti (např. těch, co vylepšují Alfa-beta ořezávání). Z toho pak sepište vlastní poznámky o té technice, případně i o jejím nasazení na Dvonn. Stačí i pár odstavců.

Asi vás zajímá bodování. Plným počtem ohodnotíme řešení obsahující:

- vhodnou reprezentaci pozice a krátký popis, jak implementovat generování tahů,
- způsob ohodnocení pozice, neboli jak a proč se různé vlastnosti stavu hry započítávají do hodnoty, rovněž s krátkým nastíněním efektivní implementace,
- alespoň krátké zamyšlení nad řazením tahů v Dvonnu,
- jak zhruba vypadá herní strom, tedy jak dlouhá je běžná hra (měřeno tahy) a kolik má hráč průměrně tahů v různých částech hry.

Jednotlivé části hodnocení lze nahradit i jiným souvisejícím nápadem, tématem apod. Velmi dobrá řešení (po kvalitativní i kvantitativní stránce) možná obdrží nějaký ten bonusový bod.

[7] http://ksp.mff.cuni.cz/viz/kucharky/hesovani
[8] http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html
[9] http://chessprogramming.wikispaces.com/
[10] http://www.boardspace.net/
[11] http://deskovehry.blogspot.com/2009/10/pravidla-dvonn.html
[12] http://ksp.mff.cuni.cz/forum/
[13] http://fragrieu.free.fr/SearchingForSolutions.pdf
[14] http://senseis.xmp.net/?MonteCarlo

Alfa-beta není zdaleka jediným používaným algoritmem v oblasti her, i pokud pomineme algoritmy vhodné jen pro konkrétní hry. V koncovkách se často hodí nasadit *Proof Number Search*,[13] bylo jím nedávno také zjištěno, že počáteční pozice v anglické dámě je remízová. Dalším zajímavým algoritmem je *Monte-Carlo Tree Search*,[14] používající pseudonáhodné simulace hry.

Oba tyto algoritmy sice nejsou jednoduché, ale jsou obecně použitelné pro velké množství her. Existují také algoritmy určené jen pro jednu hru založené na jejích specifických vlastnostech.

Tak a je po seriálu o hrách matematických i výpočetně složitějších. Věříme, že vás zaujal a třeba se vám budou nabyté znalosti ještě někdy hodit. Na vaše řešení se těší a hezké jaro přeje

Pavel „Paulie“ Veselý

Recepty z programátorské kuchařky

Geometrické algoritmy

V dnešním díle našeho kuchařkového speciálu se budeme učit vařit geometrické problémy. A co že si představujeme pod pojmem geometrický problém? Trochu analytické geometrie, například zjištění, na které straně orientované přímky bod leží, trocha plotů, neboli konvexních obalů, a obecně mnoho zametání.

V celé kuchařce se omezíme pouze na dvourozměrné problémy, tedy na algoritmy v rovině. Některé postupy se dají zobecnit pro trojrozměrné, a většinou i pro n-rozměrné problémy, ale to je již nad rámec této kuchařky.

Geometrické základy

Nejdříve trocha středoškolské analytické geometrie pro ty, kdo ji ještě neměli. Ostatní mohou tuto sekci přeskočit.

Každý bod v rovině můžeme určit jeho souřadnicemi vůči osám. Nejběžněji se používá takzvaný *kartézský souřadný systém,* tedy dvě na sebe kolmé osy označované jako x-ová osa (vodorovná) a y-ová osa (svislá). Obvykle se uvažuje, že hodnoty na osách rostou směrem doprava (osa x) a směrem nahoru (osa y), my se toho budeme v naší kuchařce držet.

Místo, kde se obě osy protínají, se označuje jako *počátek* soustavy souřadnic. Samotné *souřadnice* bodu zapisujeme jako dvojici čísel, která udávají, o kolik jednotek se musíme posunout ve směru které z os, abychom z počátku dorazili do bodu, kterému souřadnice patří. Počátek má souřadnice $[0, 0]$. Bod se souřadnicemi $[a, b]$ leží na pozici, kterou získáme tak, že se od počátku posuneme o a jednotek ve směru první osy (x-ové) a o b jednotek ve směru druhé osy (y-ové).

Vše ostatní funguje tak, jak jsme se učili při geometrii na základní škole, tedy úsečka je určena dvěma krajními body, obdélník čtyřmi a podobně. Ještě si ale řekneme, co je to vektor, a zavedeme některé další pojmy.

Často potřebujeme popsat vzájemnou polohu dvou bodů. Můžeme například udat jejich vzdálenost a směr (třeba jako úhel vzhledem k ose x). Praktičtější ale bývá říci, o kolik se liší jejich x-ové a y-ové souřadnice. To nám dá dvojici čísel, které říkáme *vektor.*

Pokud například k bodu $[1, 1]$ přičteme vektor $a = (2, -1)$, dostaneme se do bodu $[3, 0]$. Stejně tak, pokud odečteme například bod $[4, 2]$ od bodu $[1, 3]$, tak dostaneme vektor $b = (-3, 1)$ udávající jejich vzájemnou polohu.

Pomocí vektoru a bodu tedy lze určit přímku. Bod nám určí, kam umístit vektor, a vektor nám určí směr přímky z daného bodu. Tomuto vektoru se říká *směrový vektor,* nebo také někdy *směrnice,* dané přímky nebo úsečky.

Samotné vyjádření přímky nebo úsečky poté může být ve dvou tvarech. Prvním z nich je *parametrický tvar.* Základem je nějaký bod $A = [a_x, a_y]$. Od toho se ve směru směrového vektoru $u = (u_x, u_y)$ můžeme pohybovat libovolně a stále budeme na přímce. To nám vede na následující tvar, kde t je libovolný reálný parametr, neboli proměnná, za kterou si můžeme dosadit jakékoliv reálné číslo a vždy nám vyjde bod na přímce. Parametrický tvar vypadá:

$$x = a_x + tu_x$$

$$y = a_y + tu_y$$

To samé můžeme vyjádřit i vektorově, tedy $X = A + tu$.

Pro ilustrování funkce parametru, když bude $t = 0$, tak dostaneme výchozí bod přímky. Pokud poté budeme s parametrem hýbat od $-\infty$ do $+\infty$, dostaneme postupně všechny body na přímce.

Druhým způsobem zápisu je *obecný tvar přímky.* K jeho vyjádření budeme potřebovat kolmý vektor ke směrovému vektoru, tomu se také říká *normálový vektor.* V rovině ho získáme jednoduše. Pokud je $v = (v_x, v_y)$ směrnice přímky, tak vektor na něj kolmý má tvar $n = (v_y, -v_x)$. Jako poznámku pro zvídavé můžeme uvést, že *skalární součin* těchto vektorů, tedy součin po složkách ($v \cdot n = ab + b(-a)$), je roven 0, což je také jedna z definic kolmosti.

A jak tedy vypadá slibovaný obecný tvar přímky? Pokud je $n = (a, b)$ normálový vektor přímky, tak obecný tvar přímky je rovnice $ax + by + c = 0$. Dobře, a a b máme, jak ale zjistit c? Normálový vektor určuje směr, kterým přímka povede, ale stále ji můžeme libovolně posouvat. Potřebujeme ještě znát jeden bod, který na naší přímce leží, aby byla určená jednoznačně.

Když dosadíme souřadnice takového bodu do rovnice přímky s neznámou c, získáme tak rovnici pro c, kterou vyřešíme. A máme hotovo, známe hodnoty všech koeficientů v rovnici. Ještě si můžeme všimnout, že pro $c = 0$ prochází přímka počátkem.

Takovéto tvary se hodí jednak pro nějaké zapsání přímek, ale také pro *zjištění jejich průsečíku.* Když hledáme průsečík, hledáme vlastně místo, kde mají obě přímky navzájem stejné x-ové a y-ové souřadnice. A to vede na jednoduché soustavy lineárních rovnic, které jistě již vyřešit umíte.

Ještě si ale zdůrazněme rozdíl úseček oproti přímkám. V případě parametrického tvaru omezujeme velikost parametru t (například $t \in \langle 0, 1 \rangle$) a v případě obecného tvaru omezujeme rozsah jedné ze souřadnic (například $x \in \langle -2, 2 \rangle$). V případě, že bychom chtěli vyjádřit polopřímku, si parametr nebo souřadnici omezíme pouze z jedné strany.

Nakonec si ukážeme jednu základní aplikaci parametru a parametrického vyjádření úsečky. Jak snadno spočítat střed nějaké úsečky AB? V takovém případě není nic jednodušší, než si vzít vektor $B - A$, přenásobit ho parametrem $1/2$ (střed úsečky je v polovině její délky) a přičíst k bodu A. Triviální úpravou pak zjistíme, že střed úsečky můžeme spočítat jako aritmetický průměr jejích krajních bodů:

$$A + \frac{1}{2} \cdot (B - A) = \frac{A + B}{2}$$

Jako příklad na rozkoukání si ukážeme, jak zjistit, na které straně přímky leží bod.

Zjištění polohy bodu vůči přímce

Nejdříve si zavedeme pojem orientovaná přímka. Když budeme mít přímku určenou dvojicí bodů A a B, budeme se na ni dívat, jako kdybychom stáli v prvním bodě (bod A) a dívali se směrem ke druhému (bod B). Pak již máme jasně definovanou pravou a levou stranu a můžeme říci, kde vůči přímce bod leží.

Vezměme si tedy přímku určenou body A a B a bod X. Určíme si vektory $u = X - A$ a $v = B - A$ (s prvky u_x, u_y, respektive v_x, v_y) a porovnáme úhel mezi nimi.

Pokud jste už měli analytickou geometrii, určitě znáte vzoreček na výpočet úhlu mezi dvěma vektory. Vzoreček má tvar:

$$\cos \alpha = \frac{u \cdot v}{|u||v|}$$

Jeho nevýhodou je, že výpočet inverzní funkce $\cos^{-1}$ trvá dlouho. Je proto lepší použít jiný způsob výpočtu, kde si vystačíme pouze s násobením.

Tím jiným způsobem je výpočet determinantu matice určené těmito vektory. *Matice* je pouze tabulka, kde jsou vektory poskládány pod sebe (ta naše tedy bude velká 2 na 2 políčka).

Determinant matice této velikosti nám udává obsah rovnoběžníku určeného zadanými vektory. A navíc znaménko determinantu nám říká, jestli je úhel mezi vektory (měřený v kladném směru, tedy proti směru hodinových ručiček) menší než π, nebo větší než π.

Kdo se ještě s determinanty nesetkal, může brát následující vzorec pro výpočet determinantu matice dva krát dva jako kouzelnou formuli. Kdo přesto chce zdůvodnění, může si zkusit udělat rozbor všech vzájemných poloh dvou přímek (a jejich směrových vektorů), které mohou nastat. Po chvíli dojdete ke vztahu přesně odpovídajícímu následujícímu vzorečku:

$$d = u_x v_y - u_y v_x.$$

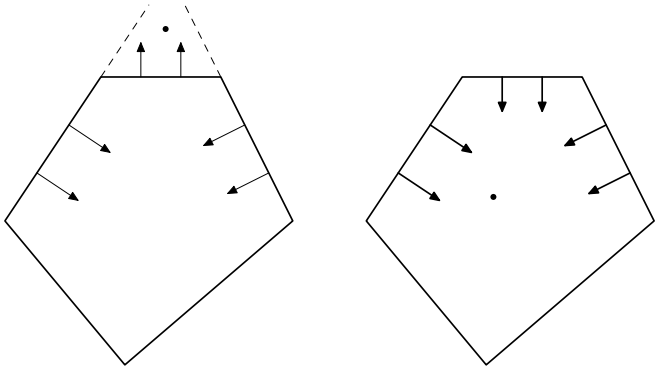
Pokud vyjde d kladné, je bod napravo od přímky, pokud vyjde d záporné, je bod nalevo od přímky, a konečně, pokud vyjde $d = 0$, tak bod leží na přímce.

Bod a konvexní mnohoúhelník

Konvexní mnohoúhelník je takový, který nemá žádný vnitřní úhel větší než 180°. Jinou definicí je, že pokud si zvolíme libovolné dva body v mnohoúhelníku a natáhneme úsečku mezi nimi, nikdy nám nevyleze z mnohoúhelníku ven.

Když už víme, co konvexní mnohoúhelník je, jak zjistíme, jestli nějaký bod leží v něm nebo ne? Využijeme vlastnosti konvexnosti. Stačí nám jít po hranách na obvodu a zjišťovat, jestli hledaný bod leží na stejné straně všech hran (tedy přímek určených koncovými body hran), nebo neleží.

Pokud bod leží na stejné straně všech hran, nachází se uvnitř mnohoúhelníku. Pokud se ale vůči jen jediné hraně nachází na jiné straně než vůči ostatním, leží bod vně mnohoúhelníku. Nejlépe to vysvětlí obrázek:



Tomuto postupu se také někdy říká test polorovinami. Každá kontrola nám zabere konstantně mnoho času. Časová složitost tohoto postupu je tedy lineární vzhledem k počtu hran, neboli $\mathcal{O}(N)$.

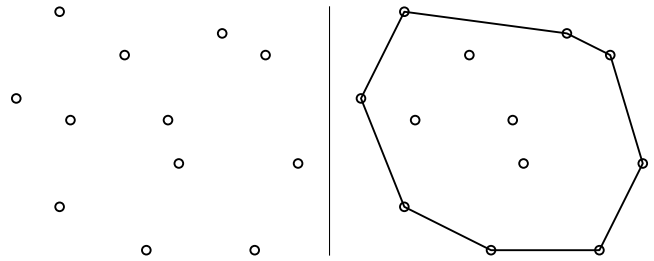
Pro nekonvexní útvary je již postup o něco těžší, jednoduše si můžeme všimnout, že postup s kontrolováním polohy bodu vůči všem hranám nebude fungovat. Zkuste si postup pro nekonvexní obrazce rozmyslet sami. Můžete buď využít testu polorovinami, jako v případě konvexního obrazce, nebo využít zajímavé vlastnosti průsečíků hran obrazce s náhodně vedenou polopřímkou.

Pokud se vám o tom nechce přemýšlet, můžete se podívat na vzorové řešení úlohy 24-4-2.[15]

Konvexní obal a zametání roviny

Podíváme se na jeden z nejznámějších geometrických problémů, totiž hledání konvexního obalu množiny bodů v rovině. *Konvexní obal* je nejmenší konvexní mnohoúhelník, který obsahuje všechny zadané body. Můžeme si všimnout, že všechny vrcholy výsledného mnohoúhelníka musí být nějaké body ze zadané množiny, jinak bychom mohli mnohoúhelník ještě zmenšit (a nebyl by to konvexní obal).

Jako motivaci si představte třeba situaci, že máte sad ovocných stromů a chcete je oplotit co nejkratším plotem. Jak takový plot, nebo obecně obal, nalézt?



Vlevo neobalené body, vpravo obalené.

Ukážeme si postup, kterému se říká *zametání roviny.* Je to trik, který najde uplatnění u mnoha různých geometrických problémů a vyplatí se ho umět.

Základní myšlenka spočívá v tom, že nějakou přímkou, říkejme jí *zametací přímka,* přejedeme přes celou rovinu (od minus nekonečna do plus nekonečna, zleva doprava nebo shora dolů) a vždy když zametací přímka protne nějaký pro nás zajímavý bod, zpracujeme příslušnou událost. *Událost* je něco významného, co souvisí s příslušným bodem (průsečík přímek, vrchol mnohoúhelníka apod.)

Ale jak jet přímkou postupně od minus nekonečna do plus nekonečna? To není vůbec nutné. Pohyb přímky můžeme

[15] `http://ksp.mff.cuni.cz/viz/24-4-2/reseni`

začít v nějakém startovním bodě (většinou první událost v setříděné posloupnosti událostí) a ukončit ho po zpracování všech událostí. Navíc nebudeme přímkou pohybovat plynule, ale budeme jí vždy skákat z události na událost (protože mezi událostmi se nic zajímavého neděje).

Vraťme se k našemu problému s konvexním obalem. Jako události budeme brát všechny body, které dostaneme na vstupu. V tomto případě nám žádné nové události v průběhu výpočtu vznikat nebudou, takže frontu událostí můžeme implementovat jako lineární spojový seznam.

Na začátku si body setřídíme podle jejich x-ové souřadnice (zatím budeme pro jednoduchost předpokládat, že žádné dva body nemají stejnou x-ovou souřadnici), začneme je zametací přímkou postupně procházet zleva doprava a budeme si udržovat konvexní obal bodů, které jsme už zpracovali.

V průběhu výpočtu si budeme konstruovat horní a dolní *obálku.* Obě obálky budou určitě začínat v nejlevějším a končit v nejpravějším bodě (jednoduchým pozorováním lze nahlédnout, že tyto body do obalu určitě patří). A jak už název napovídá, horní obálka půjde vrchem a bude se zatáčet stále doprava, a dolní obálka naopak půjde spodem a bude se stále zatáčet doleva.

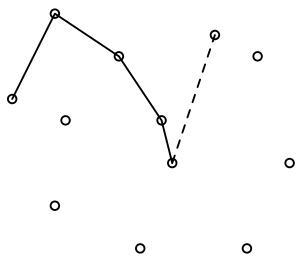
Můžeme se pro zjednodušení dohodnout, že nejlevější i nejpravější bod patří do obou obálek. Když pak horní a dolní obálku spojíme, dostaneme konvexní obal.

Horní (respektive dolní) obálku si budeme udržovat jako lineární seznam vrcholů.

Teď si ukážeme, jak bude probíhat jeden krok zpracování. Výpočet se bude provádět samostatně pro horní a dolní obálku, my si ho ukážeme jen pro horní (pro dolní je až na zrcadlení stejný).

Uvažujme, že už máme nějakou část horní obálky, skočili jsme zametací přímkou na další bod a ten teď chceme přidat. Podíváme se na poslední bod v horní obálce a zkontrolujeme úhel poslední hrany v obálce a úsečky mezi posledním bodem obalu a novým bodem.

K tomu můžeme využít například test polorovinami z úvodu kuchařky (pokud nový bod leží vůči poslední hraně horní obálky napravo, je vnitřní úhel konvexní, pokud nalevo, je úhel konkávní). Jestliže se horní obálka zatáčí doprava, máme vyhráno, přidáme nový bod do obálky a můžeme se posunout na další bod. Zajímavější je ale situace, kdy se nám obálka stočí doleva a vznikne konkávní úhel.



Pokud se podíváme na obrázek výše, jasně vidíme, že je potřeba dosavadní poslední bod obálky odebrat a zkusit spojit nově přidávaný bod s předposledním. Odstraníme tedy poslední bod obálky a budeme test opakovat s předposledním bodem.

Takto budeme pokračovat (a případně vyhazovat další body), než buď bude úhel hran konvexní, nebo dokud nám v obálce nezůstane pouze jeden bod (počáteční). Pak nový bod přidáme do obálky a pokračujeme s dalším.

Výše popsaný postup je nejvýhodnější provádět najednou pro obě dvě obálky. Tedy každý bod se pokusím připojit k horní i dolní obálce a podle toho obě obálky příslušně upravím.

Proč tento postup funguje? Postupně projdeme všechny body a každý z nich se alespoň na chvíli stane posledním bodem obálky. Při změně obálky se obsažená plocha v konvexním obalu vždy pouze zvětší a žádný bod tedy nám tedy nemůže zůstat mimo konvexní obal.

Ještě jsme zapomněli na případ, kdy úhel není ani konvexní, ani konkávní. V takovém případě se rozhodneme, jestli budeme vrchol tohoto úhlu započítávat mezi vrcholy konvexního obalu. Obvykle se takový vrchol z konvexního obalu vyhazuje, ale nakonec vždycky záleží, k čemu ten konvexní obal vlastně potřebujeme.

Skončíme, až zametací přímkou skočíme na poslední bod a zpracujeme ho. V tomto bodě se nám obálky spojí a dostaneme celý konvexní obal. Teď ale přichází otázka, kolik času nám tento postup zabere?

Může se zdát, že hodně, protože při vyhazování bodů z obálky můžeme postupně vyhodit skoro všechny body. Označme si velikost zadané množiny (počet bodů na vstupu programu) N. Musíme si uvědomit, že každý bod do obálky přidáme pouze jednou a vyhodíme ho také maximálně jednou, tedy časová složitost je lineární k velikosti množiny, tedy $\mathcal{O}(N)$, v případě, že již máme setříděný vstup. Pokud ne, musíme ještě přičíst čas potřebný k setřídění bodů, tedy $\mathcal{O}(N \log N)$ při použití nějakého rychlého třídicího algoritmu.[16]

Nakonec ještě zbývá dořešit více bodů se stejnou x-ovou souřadnicí. Pokud to nejsou krajní body, tak nám to v postupu nevadí. Menším problémem je, když to jsou počáteční, nebo koncové body. Problém ale snadno vyřešíme tím, když body seřadíme lexikograficky, tedy nejdříve podle x a pokud je stejné, pak podle y. To nám jednoznačně určí pořadí bodů a počáteční i koncový bod.

Také si to můžeme představit tak, že rovinu nepatrně natočíme. Tím se určitě konvexní obal (až na natočení) nezmění, nikde nebudou dva body nad sebou a z pohledu algoritmu je to vlastně totéž, jako bychom prošli body v lexikografickém pořadí.

Hledání průsečíků úseček

Nakonec si ukážeme ještě jeden typický zametací problém, který principu zametání využívá o trochu více než konvexní obal. Představte si, že máte v rovině N úseček a chcete najít všechny jejich průsečíky.

Hledáme samozřejmě co nejrychlejší algoritmus vzhledem k N a počtu průsečíků P.

Bystří si jistě již spočítali, že průsečíků může být v extrémním případě až N^2 a tedy nic rychlejšího než zkontrolovat každou úsečku se všemi dalšími v tomto případě není.

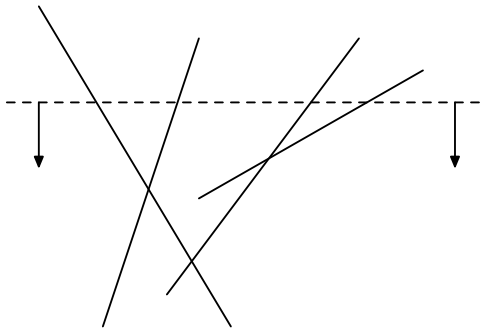
Ale takové případy se moc často nestávají, spíše naopak. Uvažujme tedy, že průsečíků je řádově tolik, kolik je úseček a v tom případě je výše popsaný algoritmus již pomalý.

Předpokládejme pro zjednodušení, že v žádném bodě se neprotínají tři a více úseček, žádné dvě úsečky nemají více než

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

jeden společný bod (neleží přes sebe) a žádná úsečka není ani přesně svislá, ani přesně vodorovná. Vyřešení takovýchto případů spočívá v jednoduchých úpravách uvedeného řešení.

Použijeme opět zametací přímku (pro lepší představu teď jdoucí shora dolů, obecně ale nemá směr zametání význam), kterou budeme skákat přes události, a na ní si budeme udržovat aktuální stav. Nazvěme ji třeba *průřezem.* Jak už název napovídá, bude udržovat pořadí úseček, které aktuálně protínají zametací přímku. Jelikož se průřez bude po každé události měnit, budeme pro něj potřebovat šikovnou datovou strukturu. Ale na to se podíváme až potom, co si rozebereme události, ať víme, co od průřezu budeme chtít.



Stejně jako v minulém případě budou mezi událostmi všechny body na vstupu (tedy počáteční i koncové body úseček), vyskytnou se tam ale i další. Pojďme si tedy trochu lépe rozebrat události a akce, které se při nich mají stát:

- *Začátek úsečky:* Přidáme úsečku na správné místo do průřezu, spočítáme případné průsečíky s okolními úsečkami a přidáme je do seznamu událostí.
- *Konec úsečky:* Smažeme úsečku z průřezu, a jelikož se nám dvě okolní úsečky dostanou smazáním této k sobě, musíme ještě spočítat jejich případný průsečík a přidat ho do seznamu událostí.
- *Průsečík:* Započítáme a zapíšeme si průsečík úseček, prohodíme pořadí těchto dvou úseček na průřezu, a jelikož se nám k sobě na průřezu dostaly nové úsečky, musíme spočítat, jestli se někde protínají, a případně průsečíky přidat do seznamu událostí.

Spočítání průsečíků úseček je jednoduchá analytická geometrie. Nejdříve porovnáme jejich směrnice. Pokud jdou od sebe, nemusíme se o nic starat, pokud jdou k sobě, spočítáme, ve kterém bodě se protnou. A když máme tento bod, jenom ověříme, jestli leží na obou úsečkách (neboli že úsečky nekončí ještě před spočítaným průsečíkem).

Když se podíváme na požadavky, hodilo by se nám umět v průřezu rychle vyhledávat, přidávat a mazat, k čemuž nám nejlépe poslouží vyhledávací strom. Ale co za informace si budeme o úsečkách ve vrcholech stromu pamatovat? Jejich aktuální x-ovou pozici (tedy přesněji x-ovou souřadnici bodu této úsečky na úrovni zametací přímky)? Tu bychom museli po každé události u všech úseček přepočítat, budeme na to tedy muset jít chytřeji.

Ve vrcholech stromu si budeme ukládat pouze nějaký rovnicový tvar úsečky (například její obecnou rovnici, nebo směrnici a bod) a vždy, když budeme vyhledávat ve stromu, tak si na základě aktuální y-ové pozice zametací přímky spočítáme v konstantním čase aktuální x-ovou pozici úsečky (jednoduchým doplněním do obecné rovnice) a podle toho se budeme ve vyhledávacím stromu pohybovat.

Máme tedy datovou strukturu pro průřez, ale jak dlouho budou trvat operace s ním? Jelikož v každou chvíli bude ve vyhledávacím stromu maximálně N vrcholů (tedy maximálně tolik, kolik je úseček), budou všechny operace se stromem trvat $\mathcal{O}(\log N)$.

Do seznamu událostí budeme potřebovat také přidávat prvky, takže tentokrát se nám mnohem více hodí použití nějaké haldy. Opět si můžeme uvědomit, že v haldě bude najednou pouze $\mathcal{O}(N)$ prvků (za každou úsečku její začátek a konec a průsečíky úseček vedle sebe na průřezu, tedy maximálně $N-1$ průsečíků) a tedy operace s haldou bude trvat také $\mathcal{O}(\log N)$.

Když už máme vybudované datové struktury, podívejme se na to, jak algoritmus poběží. Na začátku přidáme do průřezu první úsečku a do seznamu událostí všechny začátky i konce úseček. Pak již jen postupujeme po událostech, každou událost zpracujeme podle postupu výše a skončíme ve chvíli, kdy nám dojdou všechny události.

Algoritmus funguje správně, jelikož postupně projde přes všechny průsečíky (když jedna úsečka protíná více dalších, tak postupným prohazováním v průřezu se dostanou všechny tyto dvojice vedle sebe a všechny průsečíky přidáme do událostí) a žádný průsečík neprojdeme vícekrát.

Zpracování jakékoliv události nás stojí konstantní množství operací s datovými strukturami, a protože každá z těchto operací stojí maximálně $\mathcal{O}(\log N)$, tak nás zpracování jedné události stojí $\mathcal{O}(\log N)$. Počet událostí je $2N+P$ kde N je počet úseček a P počet průsečíků na výstupu, tedy celková časová složitost je $\mathcal{O}((N+P)\log N)$. Pro pořádek ještě uveďme paměťovou složitost, které je díky použitým datovým strukturám $\mathcal{O}(N)$.

Můžeme si všimnout, že pokud by průsečíků bylo řádově N^2, tak jsme si vlastně pohoršili. Předpokládali jsme ale situaci, kdy je průsečíků řádově stejně jako úseček. V tomto případě je náš algoritmus výrazně rychlejší.

Závěr

Prošli jsme si základní geometrické algoritmy pro rovinné problémy a ukázali jejich základní myšlenky. Různou aplikací a kombinací těchto postupů můžeme řešit většinu lehčích geometrických problémů v rovině, se kterými se setkáme.

Jen jako ochutnávku si ještě uvedeme například *Voroného diagramy,* což je rozklad roviny na oblasti, které jsou vždy nejblíž danému bodu (motivací může být například přiřazení obcí na mapě k nejbližšímu krajskému městu). Při jejich konstrukci se také uplatní zametání roviny, ale tentokrát již ne přímkou, ale pomocí zametacích parabol.

A jak jsme si uvedli na začátku, mnohé z uvedených postupů lze zobecnit z roviny i do prostoru a podobně. Ale o tom třeba někdy jindy. Pokud však máte zájem o další informace o geometrických algoritmech, tak vás mohu odkázat na studijní text o geometrických algoritmech[17] k přednášce ADS na stránkách Martina Mareše.

Pokud stále nemáte geometrie dost, můžete si ještě zkusit vyhledat pojmy *kombinatorická a výpočetní geometrie.* Dostanete se tak ke spoustě dalších zajímavých materiálů.

Dnešní kuchařkové menu vám servíroval

Jirka Setnička

[17] `http://mj.ucw.cz/vyuka/1112/ads2/6-geom.pdf`

24-4-1 Iniciály předků

Nejdříve si přeformulujeme zadání. Naším úkolem je pro daný řetězec zjistit jeho nejkratší periodu. Tedy takový podřetězec, jehož opakováním dostaneme celý řetězec.

Pro řešení využijeme algoritmus KMP, který je popsán v kuchařce ke čtvrté sérii. V zadaném řetězci si vytvoříme zpětné hrany, jako bychom jej chtěli vyhledávat v textu a všimneme si, že o něm platí následující tvrzení:

Buď s periodický řetězec délky n s periodou o velikosti $p < n$. Potom je p rovno délce nejdelší zpětné hrany řetězce s spočítané algoritmem KMP.

Stačí se tedy kouknout, jaká je nejdelší zpětná hrana a ověřit, zda takto dlouhý počáteční úsek nám složí celý řetězec. Pokud ano, tak máme délku periody a pokud ne, tak nejkratší periodou je celý řetězec.

Zbývá nám jen dokázat naše malé tvrzení. Zpětnou hranu delší než je perioda řetězce jednoduše mít nemůžeme, protože by pak řetězec nebyl periodický (celá perioda by zpětnou hranou byla přeskočena). Může se nám tedy stát, že by nejdelší zpětná hrana byla menší?

Nechť je délka nejdelší zpětné hrany d menší než délka periody p. O zpětných hranách víme, že každá další je buď stejně dlouhá jako předchozí, nebo delší. Z toho vyplývá, že od jistého místa řetězce budou všechny zpětné hrany stejně dlouhé.

My se podíváme na dva po sobě jdoucí úseky periody na místě, kde už se vyskytují jen nejdelší zpětné hrany. Pokud je takové místo moc na konci, tak pár period přidáme. Nyní ze zpětných hran, které jsou dlouhé d víme, že

$$s_p = s_{p-d}$$
$$s_{p+1} = s_{p-d+1}$$
$$s_{2p-1} = s_{2p-d-1}$$

Z toho dostaneme, že některé znaky v rámci periody musí být stejné. Například pro $p = 5$ a $d = 3$ dostaneme, že všechny znaky jsou stejné a tedy, že perioda je vlastně 1. Obecně pro dané p a d dostaneme z vynucených shodných znaků menší periodu, která bude rovna přesně $\mathrm{nsd}(p, d)$, což je spor s tím, že řetězec má periodu p.

Délka nejdelší zpětné hrany nemůže být ani větší, ani menší než délka periody. Tedy musí nastat rovnost.

Časová složitost algoritmu je lineární. Řetězec projdeme jen jednou při stavbě zpětných hran a jednou pro ověření, že řetězec má periodu rovnu délce nejdelší zpětné hrany. Paměťová složitost je taktéž lineární, uchováváme jen řetězec a jeho zpětné hrany.

Karel Tesař

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-4-1.cpp`

24-4-2 Sledování exponátů

Niektorí z vás sa pokúšali úlohy vyriešiť príliš mocnými nástrojmi. Zpravila týmto vzniklo správne, avšak pomalé riešenie.

Jednoduchý algoritmus je viesť polpriamku s počiatočným bodom X, kde X je bod, o ktorom chceme rozhodnúť, či je v mnohouholníku. Ak táto polpriamka pretne mnohouholník párny počet-krát, tak sme vonku, inak vnútri.

Pre každú hranu mnohouholníka zistíme, či ju náhodne zvolená polpriamka pretína. Priesečník polpriamky a úsečky nájdeme v čase $\mathcal{O}(1)$ – ak neviete ako, pozrite sa do geometrickej kuchárky.[18] Ak má mnohouholník N vrcholov, má taktiež N hrán; všetky priesečníky nájdeme v čase $\mathcal{O}(N)$.

Ak by sme náhodou polpriamkou trafili do nejakého vrcholu, zvolíme inú polpriamku. Môžeme očakávať, že mimo vrchol sa trafíme na $\mathcal{O}(1)$ pokusov, takže si časovú zložitosť nepokazíme.

Správnosť odargumentujeme tým, že ak sme vonku, tak za každé preťatie mnohouholníka, keď doň vchádzame, musíme mnohouholník preťať aj keď z neho vychádzame, teda preťatí musí byť párny počet. Ak sme naopak vnútri, tak situáciu prevedieme na prvý prípad tým, že z neho vyjdeme, čo nám dá jedno preťatie a teda celkovo máme nepárny počeť preťatí.

Poznamenám na záver zaujímavosť, že toto funguje vďaka tomu, že platí Jordanova veta o kružnici,[19] ktorá vlastne hovorí to, že každá spojitá, uzavretá krivka nepretínajúca samu seba rozdeľuje rovinu na dve disjunktné časti. Formálny dôkaz tohoto (zdanlivo) zrejmého tvrdenia dá v matematike prekvapivo veľa práce.

Peter „pizet“ Zeman

24-4-3 Cinkání skleničkami

Nejdříve ukažme dolní odhad počtu taktů a potom teprve hledejme kýžený způsob, jak to provést.

Snadno nahlédneme, že pokud si mají cinknout všichni se všemi, je počet cinknutí roven počtu hran úplného grafu o N vrcholech, tedy $\binom{N}{2} = \frac{N(N-1)}{2}$. Dále rozlišujme situaci dle parity N.

Je-li N liché, je maximální počet cinknutí v jednom taktu roven $\frac{N-1}{2}$. Tedy v každém taktu si jeden necinká, protože nemá nikoho do páru (proto $N-1$) a každé cinknutí počítáme jen jednou (proto dělíme dvojkou). Tedy minimální počet taktů je roven N.

Obdobně postupujeme pro sudá N. Dostáváme tak dolní odhad $N-1$. Ten ale můžeme vylepšit. Uvažme situaci, kdy si má v rámci jednoho z taktů cinknout např. první se třetím (účastníky číslujeme postupně po směru hodinových ručiček). V tu chvíli si druhý nemůže cinknout s nikým, neboť by musel zkřížit ruce s prvním a třetím. Pro sudá N teď máme odhad také N. Výjimku tvoří N rovno dvěma. V takovém případě lze cinknutí provést na jeden takt.

Víme tedy, že potřebujeme alespoň N taktů. Nyní již ukážeme, že dokážeme najít způsob, jak cinkání na N taktů provést. Představme si přípitek jako kruh, na jehož obvodu je rovnoměrně rozestavěno N účastníků přípitku. Dále všechny lidi očíslujme po směru hodinových ručiček 1 až N. Opět rozdělíme situaci dle parity N.

Nejdříve uvažme N liché. Člověk s číslem 1 si v tomto taktu necinkne. Pro j od 1 do $\frac{N-1}{2}$ si cinkne $(j+1)$-tý s $(N-j+1)$-

[18] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`
[19] `http://en.wikipedia.org/wiki/Jordan_curve_theorem`

tým. Každé cinknutí si představme jako úsečku mezi příslušnými lidmi. Snadno nahlédneme, že každá z úseček je rovnoběžná s ostatními, tedy podmínka nekřížení je splněna.

V dalším taktu pootočíme číslování o 1 proti směru hodinových ručiček a pokračujeme stejným způsobem. Pokud otočení opakujeme $(N-1)$-krát, dostali jsme i s počátečním rozložením N různých situací, v nichž každý z lidí necinkal právě jednou. Tedy si musel nutně cinknout se všemi ostatními.

Nyní pro sudé N. Pro prvních $\frac{N}{2}$ taktů použijme následující způsob. V prvním taktu si j-tý cinkne s $(N-j+1)$-tým pro j od 1 do $\frac{N}{2}$. Nyní i po každém z následujících $\frac{N}{2}-1$ taktů provedeme přečíslování stejným způsobem jako v případě lichého N. Úsečky reprezentující cinknutí jsou opět rovnoběžné. Vidíme, že takto si cinknou všechny páry ve tvaru lichý a sudý, resp. sudý a lichý. Situace totiž jsou opět různé a jejich počet je stejný jako počet lidí označených sudým, resp. lichým číslem.

Pro dalších $\frac{N}{2}$ taktů zvolíme cinkání následovně. V $(\frac{N}{2}+1)$-tém taktu první a $(\frac{N}{2}+1)$-tý stojí a pro j od 2 do $\frac{N}{2}$ si cinkne j-tý s $(N-j+2)$-tým. Pro každý další takt opět přečíslujeme. Jelikož je parita obou cinknuvších stejná, situace jsou opět různé; je jich $\frac{N}{2}$ a každý z účastníků stojí právě jednou. Dostáváme tady konečně způsob cinkání na N taktů i pro sudá N.

Jan Bok

24-4-4 Vozíky ve skladu

„Nebýt těch silnoproudých vodičů, šlo by to řešit jednodušeji.“ Takto si určitě povzdechla velká část z vás a měli jste částečně pravdu. Nebýt proměnlivé délky uliček, tak si celé skladiště můžeme představit jako graf a jednoduše na něj pustit Dijkstrův algoritmus.[20]

Ten nám vyhledá nejkratší cestu od jednoho vrcholu ke všem ostatním v grafu a to s časovou složitostí $\mathcal{O}((M+N)\log N)$, kde N je počet vrcholů (křižovatek) a M je počet hran (uliček mezi nimi).

Pracuje ve zkratce tak, že vždy vezme vrchol s nejmenší vzdáleností, který doposud není označený za finální, označí ho za finální a aktualizuje vzdálenosti ke všem jeho sousedům. Takto zpracuje všechny vrcholy grafu a postupně buduje cesty z nejkratších vzdáleností ke zpracovávaným vrcholům.

Podívejme se na zadání znovu. Vždyť se na grafu zase tolik nezměnilo, nestačilo by jen přidat nějaké hrany nebo upravit Dijkstrův algoritmus? Existují v podstatě dva možné přístupy.

Prvním z nich je si celý graf zdvojit pro lichou a sudou délku cesty. Vždy natáhneme hrany mezi odpovídajícími lichými a sudými vrcholy s tím, že lichým hranám se silnoproudými vodiči nastavíme dvojnásobnou délku. A pak na tento upravený graf pustíme klasický Dijkstrův algoritmus.

Tento postup je lehčí z hlediska toho, že si nemusíme upravovat samotný algoritmus, ale je těžší na přípravu grafu a na vypsání správného výstupu na konci (musíme si více hlídat, po kterých hranách jsme přišli).

Druhým postupem je neměnit si graf, ale upravit Dijkstrův algoritmus tak, aby zpracovával odděleně sudé a liché průchody. Každý vrchol tedy nebude mít jednu vlastnost finality, ale bude mít samostatnou finalitu pro lichý a sudý průchod.

V takovém případě si ale musíme zdvojit haldu vrcholů v algoritmu a při zpracování vlastně každý vrchol projdeme dvakrát. Zastavíme se ve chvíli, kdy dojdeme do cílového vrcholu po sudé i liché hraně. V ukázkovém programu použijeme tuto variantu.

Oběma postupy projdeme maximálně dvojnásobek hran, respektive dvojnásobek vrcholů, než v klasické implementaci Dijkstrova algoritmu, a paměti spotřebujeme také zhruba dvojnásobek. Tato konstanta se nám schová do $\mathcal{O}$, tedy časová složitost je stále $\mathcal{O}((M+N)\log N)$ a paměťová $\mathcal{O}(N)$.

Jirka Setnička

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-4-4.cpp`

24-4-5 Holografické projektory

Převedeme úlohu na obarvení vrcholů grafu… aha, ale to je poměrně známý NP-úplný problém, to by nám orgové neudělali, ne?

Neudělali, alespoň protentokrát ne. Zadání totiž není obecný graf, ale jen speciální druh, takže úloha není NP-úplná. Stačilo položit si oblíbenou otázku: „A nejde to hladově?“ Jde to hladově. Nuže, jak na to?

Na vstupu máme pořadí zobrazených obrazů. Příklad ze zadání `3 1 2 5 4` říká, že o první obraz se stará projektor číslo 3, o druhý obraz projektor číslo 1 atd.

Budeme si pro jednoduchost značit dvojici projektor-obraz (kterou můžeme chápat také jako paprsek) jako $x \to y$, kde x je pozice projektoru a y je pozice obrazu.

Algoritmus bude jednoduchý – prvnímu obrazu přiřadíme frekvenci 1. Nejbližšímu dalšímu obrazu, jehož paprsek se nekříží s předchozím, přiřadíme také frekvenci 1. Takto pokračujeme, dokud neprojdeme všechny obrazy.

Pak projdeme znova obrazy, jimž jsme nepřiřadili žádnou frekvenci. Prvnímu z nich dáme frekvenci 2, nejbližšímu dalšímu, jehož paprsek se nekříží s předchozím, dáme také 2, … a takto procházíme obrazy, dokud máme nějaké bezfrekvenční.

Proč to funguje? Všimneme si, že pokud má nějaký paprsek $x \to y$ frekvenci $f > 1$, tak určitě existuje paprsek $x' \to y'$ s frekvencí $f-1$, pro který platí, že $y' < y$ (obraz je víc vlevo) a $x' > x$ (projektor je víc vpravo), takže se kříží.

Totéž ale platí pro nový paprsek. Opakováním až do $f = 1$ zjistíme, že pokud existuje paprsek $x_f \to y_f$ s frekvencí $f > 1$, tak existují paprsky $x_1 \to y_1$, $x_2 \to y_2$, …, $x_f \to y_f$, pro které platí $x_1 > x_2 > x_3 > \cdots > x_f$ a zároveň $y_1 < y_2 < y_3 < \cdots < y_f$, neboli které se všechny navzájem protínají. A na takový chrchel potřebujeme jistě f barev.

Naše metoda přiřazování frekvencí tedy jistě nepřidělí zbytečně mnoho frekvencí… a je zjevné, že se žádné paprsky se stejnou frekvencí neprotínají. Tedy je náš algoritmus správně.

Jaká je jeho složitost? Označme si počet obrazů N. Času na každý průchod spotřebujeme $\mathcal{O}(N)$, průchodů bude $\mathcal{O}(N)$ (všechny paprsky se vzájemně protínají), takže celkem $\mathcal{O}(N^2)$. Paměti potřebujeme $\mathcal{O}(N)$ na uložení vstupu a výstupu.

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

Tady bychom mohli skončit s argumentem, že průsečíků zadaných paprsků může být také $\mathcal{O}(N^2)$ a všechny je musíme probrat. Ale chyba lávky, jde to zrychlit. Při jednotlivých průchodech se totiž dost flákáme a určitě se nemusíme podívat na každý průsečík zvlášť.

Během prvního průchodu se podíváme na všechny projektory, ale určíme frekvenci jen u některých. Při dalším průchodu musíme zase projít všechny zbylé ve stejném pořadí se stejnou činností. Zkusíme tedy vyřídit všechno potřebné jedním průchodem.

Projdeme obrazy od začátku do konce jen jednou. Budeme si pro každou frekvenci udržovat, na jaké pozici je poslední známý projektor, který tuto frekvenci má. Tento seznam bude jistě setříděný sestupně, rozmyslete si, proč. Díky tomu v něm můžeme vyhledávat půlením intervalu.

Vždy, když budeme zpracovávat další obraz $x \to y$, vyhledáme nejmenší takovou frekvenci f, jejíž poslední projektor $p(f)$ je víc vlevo než x ($p(f) < x$). Tuto frekvenci přiřadíme, upravíme seznam a jdeme na další obraz. Pokud zjistíme, že taková frekvence zatím není přiřazena (vytekli jsme ze seznamu), tak ji přiřadíme a zvětšíme seznam.

Proč to je ekvivalentní algoritmus? Jednoduše provádíme všechny fáze najednou podle původního plánu. Když zrovna přidělujeme frekvenci f, tak si můžeme představit, že jsme skočili zrovna do f-té fáze...

Časová složitost je nyní výrazně lepší. Půlení intervalu nám zabere nejhůř $\mathcal{O}(\log N)$ a provádíme jej N-krát, takže celkem máme $\mathcal{O}(N \log N)$. Paměťově jsme pořád na $\mathcal{O}(N)$ (musíme si uložit celé původní pole). A pak že to nejde rychleji.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-4-5.c`

Jan „Moskyto“ Matějka

24-4-6 Starý kód

Složitě (respektive spíše ošklivě) zapsaný kód je jen hledáním mostů v grafu (viz grafovou kuchařku).[21] Jeho srozumitelnost značně stoupne, pokud zjistíme, co znamená která proměnná.

`MAX_H`	Maximální počet hran grafu.
`MAX_V`	Maximální počet vrcholů grafu.
`N`	Počet vrcholů grafu.
`M`	Počet hran grafu.
`h`	Hrany grafu (s konci `.x` a `.y`).
`v`	Seznam hran vedoucích z daného vrcholu.
`p`	Počet hran vedoucích z daného vrcholu.
`f`	Fronta vrcholů, které jsme navštívili.
`b`	„Byli jsme tu“ – vrcholy, kam jsme se již dostali.

A podrobněji jaká je funkce programu? Na začátku načte hranu grafu do polí `h` a `v`. Pak prochází všechny hrany grafu `for (int k= ...` a u každé otestuje, jestli je sama mostem (tj. jestli se po zbylých hranách dá dojít z vrcholu `h[k].x` do `h[k].y`). To dělá procházením do šířky z vrcholu `h[k].x`. Do fronty zařazuje jen vrcholy, kam jsme se ještě nepodívali. Pokud jsme po vyprázdnění fronty (tj. po průchodu všech vrcholů, kam se z počátečního vrcholu lze dostat po hranách různých od `h[k]`) nenavštívili `h[k].y`, tj. druhý konec zkoumané hrany, je daná hrana mostem a tedy jí vypíšeme.

Paměťová složitost tohoto kódu je zřejmě $\mathcal{O}(M+N)$, časová $\mathcal{O}(M(M+N))$ (v každé iteraci for-cyklu můžeme projít až všechny hrany).

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_H 1000000
#define MAX_V 1001

typedef struct {int x, y;} H;
int N, M;
H h[MAX_H];
int v[MAX_V][MAX_V];
int p[MAX_V];
int f[2*MAX_V];
short b[MAX_V];

int main() {
  // načteme počet vrcholů a hran
  scanf("%d%d", &N, &M);
  if (N>MAX_V || M>MAX_H) {
    printf("Chybny vstup.\n");
    return 1;
  }
  // a následně i jednotlivé hrany
  for (int i=0; i<M; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    if (x>N || x<1 || y>N || y<1) {
      printf("Chybny vstup.\n");
      return 1;
    }
    h[i] = (H){x, y};
    v[x][p[x]++] = y;
    v[y][p[y]++] = x;
  }
  printf("Vysledny seznam:\n");
  // budeme postupně testovat všechny hrany
  for (int k=0; k<M; k++) {
    int a = 0;
    int z = 0;
    for (int i=1; i<=N; i++)
      b[i] = 0;
    // zatím jsme navštívili jen
    // počáteční vrchol
    b[h[k].x] = 1;
    f[z++] = h[k].x;
    // dokud není prázdná fronta, tj. je ještě
    // nezpracovaný vrchol...
    while (a<z && b[h[k].y]==0) {
      int q = f[a++];
      // projdeme hrany vedoucí
      // z tohoto vrcholu
      for (int i=0; i<p[q]; i++) {
        if (b[v[q][i]]==0 && !(q==h[k].x
            && v[q][i]==h[k].y)) {
          // a pokud to není testovaná hrana
          // a končí někde, kde jsme nebyli
          f[z++] = v[q][i];
          // přidáme koncový vrchol do fronty
          // na zpracování a označíme si ho
          b[v[q][i]] = 1;
        }
      }
```

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

```
    }
    // pokud jsme na konec této hrany nedošli
    // vypíšeme jí, neboť je to most
    if (b[h[k].y]==0)
      printf("%d %d\n", h[k].x, h[k].y);
  }
  return 0;
}
```

A jak program zrychlit? Vcelku jednoduše. Stačí si uvědomit, že most není v grafu součástí žádného cyklu. Takže budeme procházet graf do hloubky a pokud narazíme na hranu, která vede do vrcholu, který už jsme navštívili, jsme našli cyklus. Budeme si pamatovat, jak hluboko sahal (tj. kam až se můžeme dostat cyklem) a při návratu z rekurze víme, že všechny hrany až do této hloubky nejsou mosty. Zbytek vypíšeme. Každou hranu projdeme maximálně dvakrát takže časová i paměťová náročnost tohoto řešení je $\mathcal{O}(M+N)$.

Tímto samozřejmě dostaneme jiné pořadí výstupních hran jež z původního starého kódu. Tato nedokonalost se dá poměrně snadno napravit (aniž by to asymptoticky zpomalovalo program), zkuste si rozmyslet jak.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-4-6.c`

Pavel Čížek

24-4-7 Čtvercové bombardování

Zopár poznámok na úvod

Aj napriek tomu, že úloha je označená ako ťažká, niektorí z vás poslali riešenie, ktoré bolo očividne príliš pomalé, malo extrémne nároky na pamäť alebo dokonca oboje. Úplne najjednoduchšie riešenie úlohy má časovú zložitosť $\mathcal{O}(n)$, kde n je počet budov. Stačí si uložiť všetky body do poľa a vždy, keď príde dotaz, pole prejsť a o každom bode rozhodnúť, či do štvorca spadá. Za toto riešenie ste mohli získať jeden bod.

S jedným bodom sa neuspokojíme

Prvé, čo si možno všimnúť je, že sa vlastne pýtame na niečo, čomu by sa dalo hovoriť *dvojrozmerné intervaly*. Poďme si úlohu kvapku zjednodušiť a pozrieť sa na to, ako by sme riešili jednorozmernú verziu. Dostaneme teda (celočíselné) body $x_1, \ldots, x_n$ na x-ovej osi a chceme vedieť, že koľko ich patrí do nejakého intervalu $[x, x']$.

V tejto chvíli si spomenieme, že sme nedávno (v minulej sérii) čítali kuchárku o intervalových stromoch, a že asi bude stáť za to, pokúsiť sa tieto pozoruhodné štruktúry využiť.

Predtým, než sa pustíme do samotného rozprávania o intervalových stromoch, treba ošetriť ešte jednu nepríjemnosť. Hodilo by sa nám, aby sme nemali dva body, ktoré by mali rovnakú x-ovú alebo y-ovú súradnicu (neskôr si budete môcť rozmyslieť prečo). Označme si body zadané na vstupe $b_1, \ldots, b_n$, kde $b_i = (x_i, y_i)$. Položme $b'_i := nb_i + i$ (zmeníme obe zložky). Je zrejmé, že ak mali dva body b_i a b_j rovnakú x-ovú alebo y-ovú súradnicu, tak potom body b'_i a b'_j ju majú rôznu. Ešte nahliadnime, že ju nebudú mať rovnakú, ak ju mali rôznu. Nech $x_i > x_j$. Potom je určite $nx_i > nx_j$ a keďže $|i-j| < n$, tak aj $nx_i + i > nx_j + j$. Analogicky pre y-ovú súradnicu. Dotaz $[x, x'] \times [y, y']$ musíme však upraviť na $[nx, nx' + n - 1] \times [ny, ny' + n - 1]$. V ďalšom texte predpokladáme body s rôznymi súradnicami.

Vráťme sa teraz k jednorozmernej verzii problému. Na tu nám v skutočnosti bude stačiť obyčajné pole, v ktorom sú body zoradené vzostupne. Pri dotaze typu $[x, x']$ stačí v poli dvakrát binárne vyhľadať. Najprv hodnotu x a potom x'. Potom je už jednoduché zistiť počet bodov, ktoré vyhovujú dotazu. Odpoveď zvádneme v čase $\mathcal{O}(\log n)$ a pole pripravíme na dotazy v čase $\mathcal{O}(n \log n)$.

Ďalšou možnosťou je použiť istý druh intervalových stromov. Intervalový strom pre body $x_i, \ldots, x_j$ (nech sú zoradené vzostupne) definujeme rekurzívne:

- Koreň bude uchovávať informáciu o bodoch $x_i, \ldots, x_j$.
- Ak $i < j$, tak položme $mid = \lfloor (i+j)/2 \rfloor$. Ľavý syn uchová informáciu o bodoch $x_i, \ldots, x_{mid}$ a potom pravý o bodoch $x_{mid+1}, \ldots, x_j$.

Môžete si všimnúť, že každý vrchol v intervalového stromu S vybudovaného nad bodmi $x_1, \ldots, x_n$ reprezentuje nejaký úsek $[x_i, x_j]$ na x-ovej osi, jeho ľavý syn $\mathrm{l}(v)$ reprezentuje interval $[x_i, x_{mid}]$ a pravý syn $\mathrm{p}(v)$ reprezentuje interval $[x_{mid+1}, x_j]$, kde $mid = (i+j)/2$.

Takýto strom bude mať hĺbku $\mathcal{O}(\log n)$ a jeho definícia nám vlastne hovorí, ako ho budeme budovať. Budovanie má časovú zložitosť $\mathcal{O}(n \log n)$, keďže si body potrebujeme vzostupne zoradiť. Pamäťová zložitosť je $\mathcal{O}(n)$.

Ako odpovedať na dotaz $[x, x']$? Chceme vlastne vybrať také vrcholy stromu S, aby spolu reprezentovali interval $[x, x']$ a zároveň chceme, aby týchto vrcholov bolo čo najmenej. Vyhľadáme si v strome x a x'. Budeme vyhľadávať ako v binárnom vyhľadávacom strome až na to, že z vrcholu v sa do $\mathrm{l}(v)$ presunieme ak vyhľadávaná hodnota je menšia alebo rovná x_{mid} a v opačnom prípade do $\mathrm{p}(v)$). Vyhľadávanie skončíme v liste.

Označme v_x a $v_{x'}$ listy, v ktorých skončí vyhľadávanie x a x' a v_p ich najbližšieho spoločného predka. Skontrolujeme, či do hľadaných bodov máme započítať aj bod v v_x a $v_{x'}$. Teraz budeme postupovať z v_x do v_p a vždy, keď do nejakého vrcholu prídeme z ľavého syna, tak do výsledku pridáme interval z pravého syna. Rovnako budeme postupovať z $v_{x'}$ do v_p a ak prídeme do vrcholu z pravého syna, tak pridáme interval z ľavého syna.

Na dotaz vieme teda odpovedať v čase $\mathcal{O}(\log n)$. Neskôr sa presvedčíme, že rovnako rýchlo sme schopný odpovedať aj na dvojrozmerný dotaz.

Dvojrozmerné intervalové stromy

Skúsme teraz zostrojiť štruktúru, v ktorej budeme schopní odpovedať na dvojrozmerný dotaz.

Použijeme intervalový strom z predchádzajúcej časti, aby sme rozdelili jeden dvojrozmerný dotaz na niekoľko jednorozmerných poddotazov. Vybudujeme intervalový strom S, ktorý ignoruje y-ové súradnice bodov. V každom vrchole v intervalového stromu, ktorý reprezentuje interval $[a_v, b_v]$ na x-ovej, vybudujeme druhoúrovňovú štruktúru $S_y(v)$, ktorá obsahuje všetky body v intervale $[a_v, b_v]$. Každý vrchol v stromu S nám teda reprezentuje nejaký vertikálny pásik v rovine a $S_y(v)$ uchováva body v ňom. Štruktúra $S_y(v)$ môže byť opäť intervalový strom, ktorý tentoraz ignoruje x-ové súradnice. Môže to byť ale aj pole bodov v príslušnom vertikálnom pásiku, zoradených vzostupne podľa y-ovej súradnice. Prvá varianta sa ľahšie zovšeobecní do viacerých dimenzií, my ale pre jednoduchosť budeme uvažovať, že $S_y(v)$ je pole.

Môžeme si všimnúť, že pre každý vrchol v stromu S platí, že body uložené v $S_y(v)$ sú presne body uložené v $S_y(\mathrm{l}(v))$ a $S_y(\mathrm{p}(v))$. Preto ak $S_y(\mathrm{l}(v))$ a $S_y(\mathrm{p}(v))$ poznáme, tak $S_y(v)$ vybudujeme jednoducho zliatím $S_y(\mathrm{l}(v))$ a $S_y(\mathrm{p}(v))$. Celé budovanie si môžeme predstaviť ako merge sort, s rozdielom, že doposiaľ zoradené polia si ukladáme v jednotlivých vrcholoch S a nakoniec v koreni k dostaneme výsledné vzostupne zoradené pole. A síce, pole $S_y(k)$. Vďaka tomu, že máme rôzne súradnice, môžeme body do druhoúrovňovej štruktúry rozmiestniť rovnomerne.

Je vidieť, že budovanie má časovú zložitosť $\mathcal{O}(n \log n)$, rovnako ako merge sort. Hĺbka stromu je $\mathcal{O}(\log n)$. Na každej hladine si v druhoúrovňových štruktúrach pamätáme spolu $\mathcal{O}(n)$ bodov. Pamäťová zložitosť je teda $\mathcal{O}(n \log n)$.

Na dotaz typu $[x, x'] \times [y, y']$ môžeme odpovedať tak, že sa najprv stromu S spýtame na $[x, x']$, čím vymedzíme $\mathcal{O}(\log n)$ vrcholov S, ktoré spolu tvoria horizontálny interval $[x, x']$. Pre každý takýto vrchol v dvakrát vyhľadáme v poli $S_y(v)$. Najprv hodnotu y, potom y'. Jednoducho spočítame, koľko bodov sa nachádza v $[a_v, b_v] \times [y, y']$, a keďže to spočítame pre každý vrchol v, ktorý sme vymedzili, tak dostaneme počet bodov v $[x, x'] \times [y, y']$.

Pri dotaze $\mathcal{O}(\log n)$-krát binárne vyhľadáme. Časová zložitosť je teda $\mathcal{O}(\log^2 n)$. Za riešenie, ktoré malo rovnakú časovú zložitosť, ste mohli získať plný počet bodov.

Na záver

Počas výkladu riešenia sme nikde nevyužili toho, že dotaz, ktorý príde na vstupe je štvorcový. Sme teda schopní odpovedať aj na ľubovoľný obdĺžnikový dotaz v rovine.

Peter „pizet“ Zeman

Fractional cascading

Existuje moc pěkný trik (říká se mu *fractional cascading*), kterým se dá časová složitost dotazu v dvojrozměrném intervalovém stromu snížit na $\mathcal{O}(\log n)$.

Zopakujme si, jak se vyhodnocuje obdélníkový dotaz: postupujeme stromem shora dolů a podle x-ových souřadnic se rozhodujeme, zda máme jít doleva nebo doprava. V každém vrcholu, který navštívíme, je přitom uložen seznam, v němž potřebujeme vyhledat minimální a maximální y-ovou souřadnici našeho obdélníku. Jelikož seznam je setříděný, můžeme hledat binárně v čase $\mathcal{O}(\log n)$. To provedeme $\mathcal{O}(\log n)$-krát.

Hledání v setříděném seznamu obecně zrychlit nemůžeme, ale pomůže nám, když si uvědomíme, že seznamy, v nichž hledáme, nejsou nezávislé. Pokaždé, když se přesuneme do nějakého syna, najdeme v něm totiž podseznam seznamu uloženého v otci.

Podívejme se na to tedy obecněji: Máme nějaký seznam $A = a_1, \ldots, a_n$ a víme, kde se v něm nachází číslo x – buďto je rovno nějakému a_i, nebo leží mezi a_i a a_{i+1}. Nyní chceme totéž x najít v jeho podseznamu $B = b_1, \ldots, b_m$.

K tomu nám pomůže, když si pro každé a_i předpočítáme, kde se nachází v seznamu B. A pokud se tam nenachází, zapamatujeme si nejbližší větší prvek seznamu B. Kdyby neexistoval (a_i by bylo větší než všechna b_j), ukážeme za konec seznamu B. Řečeno formálně: $f_i := \min\{j \mid b_j \geq a_i\}$, přičemž dodefinujeme $b_{m+1} := +\infty$, aby minimum vždy existovalo.

Pokud tedy pro hledané x platí $a_i \leq x < a_{i+1}$, najdeme ho v seznamu B mezi pozicemi $f_{i+1} - 1$ a f_{i+1}.

Vraťme se k intervalovému stromu. Do každého jeho vrcholu přidáme pomocné ukazatele, které seznam v tomto vrcholu propojí se seznamy v obou synech. V kořeni tedy budeme stále muset použít binární vyhledávání, ale pokaždé, když se přesuneme do syna, přepočítáme pouze začátek a konec intervalu podle uložených ukazatelů, což stihneme v konstantním čase. Celkem nás tedy celé hledání stojí $\mathcal{O}(\log n)$ v kořeni a $\mathcal{O}(1)$ v $\mathcal{O}(\log n)$ dalších vrcholech, což dohromady dává $\mathcal{O}(\log n)$.

Martin „Medvěd“ Mareš

Program (C):
`http://ksp.mff.cuni.cz/viz/24-4-7.c`

24-4-8 O hrách a číslech

Úkol 1: Hromádka n sirek

Úkol vyřešíme takto: nejdříve přiřadíme číslo hrám pro malá n, poté si ukážeme, že pro každé n je hra číslo, a nakonec odvodíme vzorec, jak určit toto číslo.

- $n = 0$ – nikdo nemá tah, hra je tedy 0,
- $n = 1$ – levý má tah do 0, pravý táhnout nemůže, což se dá zapsat $\{0 \mid \} = 1$,
- $n = 2$ – levý táhne do 0, pravý do 1, tedy $\{0 \mid 1\} = 1/2$,
- $n = 3$ – $\{1/2 \mid 1\} = 3/4$,
- $n = 4$ – $\{1/2 \mid 3/4\} = 5/8$,
- $n = 4$ – $\{5/8 \mid 3/4\} = 11/16$,

Dále chceme ověřit, že každá hra je číslo. Nejdříve si všimneme, že hry pro sudá n mají menší čísla než hry s $n + 1$ sirkami a naopak hry pro lichá n mají větší čísla než hry pro $n + 1$ sirek.

Důkaz provedeme indukcí podle n. Pro prvních pár her platí, že sudé hry mají menší číslo než liché (viz výše). Podívejme se na hru s n sirkami, přičemž předpokládáme, že to máme dokázané pro všechny menší hry.

Je-li n sudé, hraje levý do hry s $n - 2$ sirkami a pravý do $n - 1$. Z indukčního předpokladu víme, že číslo hry $n - 2$ je menší než číslo hry $n - 1$, tedy hra s n sirkami je číslo, navíc menší, než má hra s $n - 1$ sirkami. Analogicky pro liché n je hra číslo větší než hra pro předchozí sudé $n - 1$.

Dalším pozorováním je, že hra s n sirkami má po rozšíření zlomku dvěma stejný jmenovatel jako hra s $n + 1$ sirkami. Jelikož jejich čitatel se liší jen o 1, hra s $n + 2$ sirkami tak má ve jmenovateli koeficient o jedna větší a je rovna jejich průměru.

Na čísla her se můžeme dívat jako na posloupnost a_n, kde a_n udává číslo hry s n sirkami. Počátek posloupnosti je $a_0 = 0$, $a_1 = 1$. Platí následující vzorec, který stačil k plnému počtu bodů:

$$a_n = (a_{n-1} + a_{n-2})/2.$$

Nyní můžeme chtít explicitní vzorec pro n-tý člen posloupnosti. Ten vyřešíme pomocí kuchařky o lineárních rekurencích.[22]

Charakteristický polynom posloupnosti $x^2 - x/2 - 1/2 = 0$ má řešení $-1/2$ a 0. Vzorec tedy bude tvaru $a_n = A \cdot 1^n + B \cdot (-1/2)^n$ pro nějaké konstanty A a B. Ty zjistíme dosazením za prvních pár členů posloupnosti, čili vyřešením soustavy rovnic $0 = A + B$ a $1 = A - B/2$.

[22] `http://mj.ucw.cz/papers/linrec.pdf`

Z první rovnice vyjde, že $A = -B$, druhou upravíme na $1 = A + A/2$. Dostáváme $A = 2/3$ a $B = -2/3$, vzorec pro n-tý člen tudíž je:

$$a_n = 2/3 - 2/3 \cdot (-1/2)^n$$

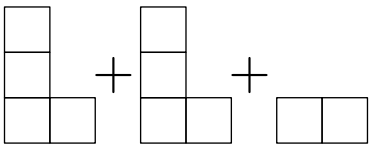
Kdo se setkal s konvergencí posloupností, asi již vidí, že limitou posloupnosti jsou 2/3.

[Poznámka M.M.: Mimochodem, jde to i bez explicitního vzorce. Zkusme členy posloupnosti zapisovat ve dvojkové soustavě: $a_0 = 0$, $a_1 = 1$, $a_2 = 0.01$, $a_3 = 0.11$, $a_4 = 0.101$, $a_5 = 0.1011$, $a_6 = 0.10101$, ... a není těžké ověřit (třeba indukcí), že i další členy mají tento pravidelný tvar. Blíží se proto k $0.\overline{10}$, což jsou desítkově 2/3.]

Úkol 2: Dominování

Pro pozici v dominování s hodnotou 1/2, kterou budeme označovat G, chceme dokázat, že $G + G = 1$. Nejsnažším řešení bylo použít poznámku na začátku seriálu: pokud $G - H$ je prohraná hra, pak $G = H$.

Budeme tedy jednoduše zkoumat hru $G + G - 1$, která vypadá takto:



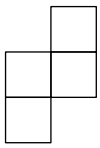
Začne-li levý, položí do jedné z her G své svislé domino buď tak, aby sebral soupeři tah, nebo o políčko výše. Při první možnosti zahraje pravý do druhé hry G a vznikne součet $1 - 1 = 0$. Na tahu je levý, tedy prohrává.

Druhá možnost (levý položí domino o políčko výše) vede opět na výhru pravého: pravý položí domino do stejné hry jako levý. Ten má v druhé hře G jeden tah, ale pravý má ještě další tah ve hře -1.

Pokud začne pravý, může začít hrát do G nebo -1. Zahraje-li do G, levý potáhne do druhé hry G tak, aby tam pravý neměl tah. Opět dostáváme součet $1 - 1$, na tahu je však pravý, kvůli čemuž prohraje.

Jestliže pravý položí první domino do hry -1, levý zahraje do G tak, aby tam už nemohl táhnout pravý, jemuž zbude jedna možnost v druhé hře G. Potom však bude mít levý ještě jeden tah, ale pravý ne, takže prohraje.

Druhá část úkolu byla celkem o nápadu, proto jsme nakonec její absenci hodnotili mírně. Řešením je třeba tato pozice H:



Dokážeme jen, že H není číslo, a ověření rovnosti $H + H = 1$ necháme jako cvičení (velmi se podobá první části úkolu).

Ve hře H má levý dva tahy, oba vedou do hry 1, pravý může táhnout jen do hry 0. Platí tedy $H = \{1 \mid 0\}$, čili H není číslo. (Lze rovněž vyvrátit, že $H = G$, konkrétně přes rozbor hry $H - G$.)

Úkol 3: Padající domino

Úkol byl cvičením na vyškrtáváním tahů, bylo však třeba dávat pozor a ověřovat nerovnosti: např. mezi možnostmi levého můžeme vyškrtnout pozici A jen, pokud může levý zahrát do B a $A \leq B$. Jak ověřovat nerovností je popsáno na začátku seriálu.

Nejprve přiřadíme čísla několika jednodušším pozicím:

- všechna domina popadala – číslo 0 (nikdo nemá tah),
- v pozici B má levý dvě možností a pravý žádnou, takže je to 1, pozice BB je 2, BBB je 3... Podobně například $CCCC$ je -4,
- v BC mají oba hráči tah do 0, takže je to *,
- v BBC má levý tah do 0 a 1 (0 je pro levého horší než 1, můžeme ji vyškrtnout), a pravý do 0, jde tedy o $\{1 \mid 0\}$. $BBBC$ je z podobného důvodu $\{2 \mid 0\}$ a platí $BBBC > BBC$, což se ověří prozkoumáním hry $BBBC - (BBC) = BBBC + CCB$ (je-li to hra levého, nerovnost platí).

Také si všimneme, že otočené pozice dostanou stejná čísla ($BBBBC$ i $CBBBB$ jsou $\{3 \mid 0\}$).

Nyní se podívejme na pozici $BCBBBBC$. Levý má možnost hrát do následujících pozic:

- 0 (všechna domina shozena, nikdo nemá tah),
- $BBBC = \{2 \mid 0\}$,
- BBC, BC, C jsou všechny horší než $BBBC$ (lze dokázat, že $BBC < BBBC$), takže je můžeme zapomenout,
- $CBBBBC$ – v této pozici má levý tah do $BBBC$ (ostatní možnosti jsou pro něj horší) a pravý do 0 a $BBBBC$ (0 a $BBBBC$ jsou neporovnatelné, neboť $BBBBC + 0$ je hra vyhraná pro začínajícího). Platí tedy: $CBBBBC = \{\{2 \mid 0\} \mid 0, \{3 \mid 0\}\}$. Opět lze dokázat, že tato pozice je horší než $BBBC$,
- BCB, $BCBB$, $BCBBB$ – z těchto her má cenu uvažovat jen $BCBBB$ (ostatní jsou menší, důkaz ponecháme jako cvičení). $BCBBB = \{2 \mid 1\}$, což je větší než $\{2 \mid 0\}$ a 0, takže $BBBC$ a 0 můžeme vyškrtnout.

Pravý může táhnout do pozic:

- všechna domina shozena, tedy 0,
- $B = 1$, ale $1 > 0$, takže tuhle možnost můžeme zapomenout,
- $BBBBC = \{3 \mid 0\}$,
- $BCBBBB = \{3 \mid 1\}$, ale to je větší než $BBBBC$.

Celkově tedy $BCBBBBC = \{\{2 \mid 1\} \mid 0, \{3 \mid 0\}\}$. (To odpovídá intuitivnímu odhadu, že levý zahraje do $BCBBB$.) Možnost $\{3 \mid 0\}$ pro pravého můžeme sice intuitivně vyškrtnout, ale formálně na to nemáme nástroj (hry 0 a $\{3 \mid 0\}$ jsou neporovnatelné).

Podobně, ale stručněji rozebereme hru $BBCCBC$. Levý má tahy do:

- $B = 1$ (což je větší než 0 či $C = -1$, které můžeme vyškrtnout),
- $BBCC = \pm 1$ (což je neporovnatelné s 1)
- $CCBC$ je zjevně horší než 0,
- $BCCBC$ – jelikož hra $BCCBC - B$ je vyhraná pro pravého, platí $B > BCCBC$ a možnost $BCCBC$ není třeba uvádět.

Pravý má možnost táhnout do následujících pozic:

- $CBC = -1/2$
- všechna domina shozena, tedy 0, ale $0 > -1/2$,
- $BC = * > -1/2$,
- $BBC = \{1 \mid 0\} > *$,
- $BB = 2 > -1/2$,
- $BBCCB > -1/2$.

Tedy platí, že $BBCCBC = \{1, \pm 1 \mid -1/2\}$. Možnost ± 1 můžeme intuitivně vyškrtnout, i když je neporovnatelná s 1.

Pavel „Paulie“ Veselý

Výsledková listina čtvrté série dvacátého čtvrtého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2441*	*2442*	*2443*	*2444*	*2445*	*2446*	*2447*	*2448*	*série*	*celkem*
0.					10	8	8	10	12	9	13	15	60,0	237,0
1.	Vojtěch Hlávka	GŠlapanice	3	14	10	8	8	10	6	9	5	5	45,0	208,8
2.	Martin Raszyk	G_Karvina	2	9	10		8	10	12	9	1		49,0	195,7
3.	Lukáš Ondráček	GVolgogrOS	3	4	10			10	12	9	3		46,4	176,2
4.	Michal Pokorný	SŠkybernHK	4	9		8		8	8	8			33,3	161,0
5.	Jiří Eichler	SlovanGOL	4	11	10								10,0	138,0
6.	Alexander Mansurov	GNVPláníPH	3	8	10	4	8	9,5		2	8		41,7	137,4
7.	Dominik Macháček	GLanškroun	3	4	2	8	7	4	8		0		35,7	135,9
8.	Mark Karpilovsky	GJarošeBO	3	4	6	8	2,5			9			29,2	132,9
9.	Jerguš Greššák	ŠPMNDaGB	3	8									0,0	110,9
10.	Vojtěch Vašek	GHli	3	3	10	8	2	7				5	39,3	104,6
11.	Vojtěch Sejkora	SPSE_Pard	3	9	10	3	5		2			6	28,5	100,9
12.	Michal Punčochář	GJírovcČB	2	5	9								9,6	99,1
13.	Rastislav Rabatin	GJHroncaBA	3	3	10	8		6			13		39,2	90,8
14.	Martin Španěl	ArcibisGPH	3	2	10	1	8		4	6	1	15	48,2	85,3
15.	Štěpán Trčka	GSlavičín	1	3	9				8	2	0		24,0	85,1
16.	Jan Knížek	G_Strakon	1	4									0,0	84,2
17.	Martin Mirbauer	PORGPha	4	3									0,0	77,1
18.	Ondřej Mička	GJírovcČB	3	12	10					9			19,0	77,0
19.	Matej Lieskovský	GOmskPha	2	4		6				1			9,1	75,2
20.	Jan-Sebastian Fabík	GJarošeBO	2	4	10								10,0	74,0
21.	Aneta Šťastná	GOmskPha	2	3									0,0	70,5
22.	Dalimil Hájek	GKepleraPH	1	4			4,5		7	1			17,7	67,3
23.	Lukáš Folwarczný	GKomHavíř	4	9	10								10,0	66,6
24.	Ondřej Cífka	GNAlejíPH	3	8	10								10,0	43,4
25.	Ondřej Hübsch	GArabskáPH	2	14	10								10,0	43,0
26.	Joel Jančařík	MensaG	4	1									0,0	38,8
27.	David Bernhauer	GZborovPH	4	4									0,0	38,5
28.	Jonatan Matějka	GJírovcČB	2	9									0,0	34,3
29.	Jan Hadrava	GZborovPH	4	5									0,0	32,6
30.	Jindřich Pilař	GBroumov	4	8									0,0	30,2
31.	Tereza Hulcová	GKlatovy	3	7									0,0	28,5
32.	Pavel Kratochvíl	VOŠGSvětlá	4	15	8								7,1	24,5
33.	Josefina Mádrová	GDobruška	4	3			8						8,0	21,4
34.	Bohumil Mravenec	GArabskáPH	3	1	10	8	1						20,5	20,5
35.	Pavel Salva	VOŠŠumperk	2	3									0,0	19,4
36.	Štěpán Šimsa	GJungmanLT	3	14	8								7,3	16,3
37.	Jitka Fürbacherová	GKlatovy	3	5									0,0	15,7
38.	Jan Žárský	VSSKopř	1	1									0,0	14,1
39.	Zuzana Vozárová	GJHroncaBA	4	1									0,0	13,8
40.	Václav Volhejn	GKepleraPH	-1	1	9								9,8	9,8
41.	Vojtěch Polívka	GMikulášPL	4	1									0,0	9,5
42.	František Zajíc	G_Nymburk	-1	1									0,0	9,2
43.	Michal Hruška	GJirsíkaČB	4	1									0,0	7,6
44.	Matěj Židek	GBroumov	4	7									0,0	6,2
45.	Vladan Glončák	GĽŠtúraTN	3	1	3								6,0	6,0
46.	Břetislav Hájek	GČesBrod	-2	3									0,0	5,9
47.	Juda Kaleta	GKlatovy	3	8									0,0	5,2
48.	Jan Pavlík	VOŠŠumperk	4	1									0,0	1,3