8. Geometrie vrací úder *(sepsal Pavel Klavík)*

Když s geometrickými problémy pořádně nezametete, ony vám to vrátí! Ale když už zametat, tak určitě ne pod koberec a místo smetáku použijte přímku. V této přednášce nás spolu s dvěma geometrickými problémy samozřejmě čeká pokračování pohádky o ledních medvědech.

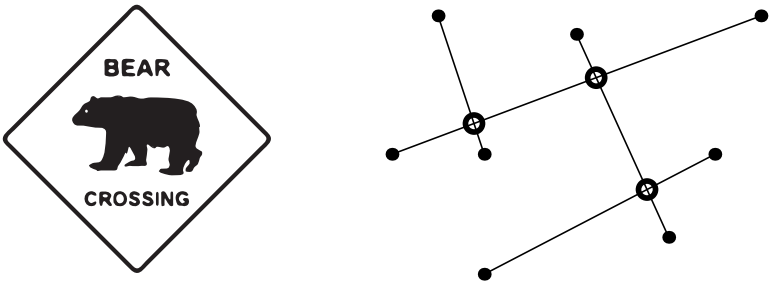
Medvědi vyřešili rybí problém a hlad je již netrápí. Avšak na severu nežijí sami, za sousedy mají Eskymáky. Protože je rozhodně lepší se sousedy dobře vycházet, jsou medvědi a Eskymáci velcí přátelé. Skoro každý se se svými přáteli rád schází. Avšak to je musí nejprve nalézt ...

Hledání průsečíků úseček

Zkusíme nejprve Eskymákům vyřešit lokalizaci ledních medvědů.

Když takový medvěd nemá co na práci, rád se prochází. Na místech, kde se trasy protínají, je zvýšená šance, že se dva medvědi potkají a zapovídají – ostatně co byste čekali od medvědů. To jsou ta správná místa pro Eskymáka, který chce potkat medvěda. Jenomže jak tato křížení najít?

Pro zjednodušení předpokládejme, že medvědi chodí po úsečkách tam a zpět. Budeme tedy chtít nalézt všechny průsečíky úseček v rovině.



Problém Eskymáků: Kde všude se kříží medvědí trasy?

Pro n úseček může existovat až $\Omega(n^2)$ průsečíků.[1] Tedy optimální složitosti by dosáhl i algoritmus, který by pro každou dvojici úseček testoval, zda se protínají. Časovou složitost algoritmu však posuzujeme i vzhledem k velikosti výstupu p. Typické rozmístění úseček mívá totiž průsečíků spíše pomálu. Pro tento případ si ukážeme podstatně rychlejší algoritmus.

Pro jednodušší popis předpokládejme, že úsečky leží v obecné poloze. To znamená, že žádné tři úsečky se neprotínají v jednom bodě a průnikem každých dvou úseček je nejvýše jeden bod. Navíc předpokládejme, že krajní bod žádné úsečky neleží na jiné úsečce a také neexistují vodorovné úsečky. Na závěr si ukážeme, jak se s těmito případy vypořádat.

[1] Zkuste takový příklad zkonstruovat.

Algoritmus funguje na principu zametání roviny, popsaném v minulé přednášce. Budeme posouvat vodorovnou přímku odshora dolů. Vždy, když narazíme na nový průnik, ohlásíme jeho výskyt. Samozřejmě spojité posouvání nahradíme diskrétním a přímku vždy posuneme do dalšího zajímavého bodu.

Zajímavé události jsou *začátky úseček*, *konce úseček* a *průsečíky úseček*. Po utřídění známe pro první dva typy událostí pořadí, v jakém se objeví. Výskyty průsečíků budeme počítat průběžně, jinak bychom celý problém nemuseli řešit.

V každém kroku si pamatujeme *průřez* P – posloupnost úseček aktuálně protnutých zametací přímkou. Tyto úsečky máme utříděné zleva doprava. Navíc si udržujeme kalendář K budoucích událostí. Z hlediska průsečíků budeme na úsečky nahlížet jako na polopřímky. Pro sousední dvojice úseček si udržujeme, zda se jejich směry někde protnou. Algoritmus pro hledání průniků úseček funguje následovně:

Algoritmus:

1. $P \leftarrow \emptyset$.
2. Do K vložíme začátky a konce všech úseček.
3. Dokud $K \neq \emptyset$:
4. Odebereme nejvyšší událost.
5. Pokud je to začátek úsečky, zatřídíme novou úsečku do P.
6. Pokud je to konec úsečky, odebereme úsečku z P.
7. Pokud je to průsečík, nahlásíme ho a prohodíme úsečky v P.
8. Navíc vždy přepočítáme průsečíkové události, vždy maximálně dvě odebereme a dvě nové přidáme.

Zbývá rozmyslet si, jaké datové struktury použijeme, abychom průsečíky nalezli dostatečně rychle. Pro kalendář použijeme například haldu. Průřez si budeme udržovat ve vyhledávacím stromě. Poznamenejme, že nemusíme znát souřadnice úseček, stačí znát jejich pořadí, které se mezi jednotlivými událostmi nemění. Při přidávání úseček procházíme stromem a porovnáváme souřadnice v průřezu, které průběžně dopočítáváme.

Kalendář obsahuje vždy nejvýše $\mathcal{O}(n)$ událostí. Podobně průřez obsahuje v každém okamžiku nejvýše $\mathcal{O}(n)$ úseček. Jednu událost kalendáře dokážeme ošetřit v čase $\mathcal{O}(\log n)$. Všech událostí je $\mathcal{O}(n+p)$, a tedy celková složitost algoritmu je $\mathcal{O}((n+p)\log n)$.

Slíbili jsme, že popíšeme, jak se vypořádat s výše uvedenými podmínkami na vstup. Události kalendáře se stejnou y-ovou souřadnicí budeme třídit v pořadí začátky, průsečíky a konce úseček. Tím nahlásíme i průsečíky krajů úseček a ani vodorovné úsečky nebudou vadit. Podobně se není třeba obávat průsečíků více úseček v jednom bodě. Úsečky jdoucí stejným směrem, jejichž průnik je úsečka, jsou komplikovanější, ale lze jejich průsečíky ošetřit a vypsat třeba souřadnice úsečky tvořící jejich průnik.

Na závěr poznamenejme, že Balaban vymyslel efektivnější algoritmus, který funguje v čase $\mathcal{O}(n \log n + p)$, ale je podstatně komplikovanější.

Hledání nejbližších bodů a Voroného diagramy

Nyní se pokusíme vyřešit i problém druhé strany – pomůžeme medvědům nalézt Eskymáky.

Eskymáci tráví většinu času doma, ve svém iglů. Takový medvěd je na své toulce zasněženou krajinou, když tu se najednou rozhodne navštívit nějakého Eskymáka. Proto se podívá do své medvědí mapy a nalezne nejbližší iglů. Má to ale jeden háček, iglů jsou spousty a medvěd by dávno usnul, než by nejbližší objevil.⟨2⟩

Popíšeme si nejprve, jak vypadá medvědí mapa. Medvědí mapa obsahuje celou Arktidu a jsou v ní vyznačena všechna iglů. Navíc obsahuje vyznačené oblasti tvořené body, které jsou nejblíže k jednomu danému iglů. Takovému schématu se říká *Voroného diagram*. Ten pro zadané body $x_1, \ldots, x_n$ obsahuje rozdělení roviny na oblasti $B_1, \ldots, B_n$, kde B_i je množina bodů, které jsou blíže k x_i než k ostatním bodům x_j. Formálně jsou tyto oblasti definovány následovně:

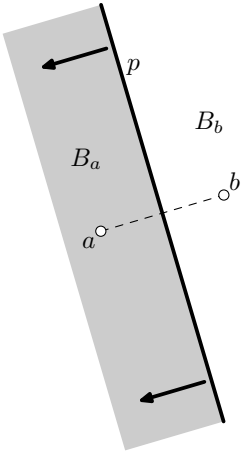
$$B_i = \left\{ y \in \mathbb{R}^2 \mid \forall j : \rho(x_i, y) \leq \rho(x_j, y) \right\},$$

kde $\rho(x, y)$ značí vzdálenost bodů x a y.

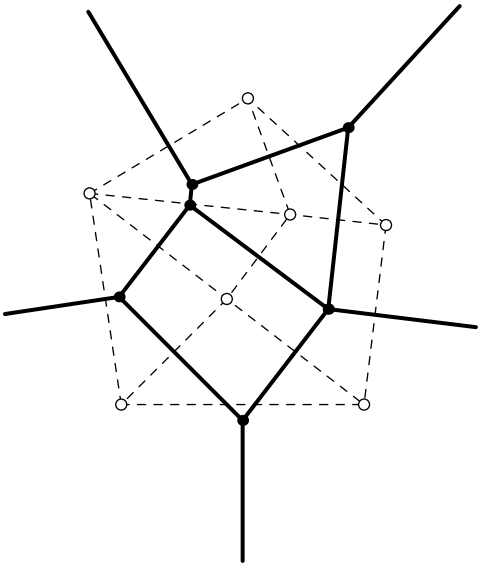
Ukážeme si, že Voroného diagram má překvapivě jednoduchou strukturu. Nejprve uvažme, jak budou vypadat oblasti B_a a B_b pouze pro dva body a a b, jak je naznačeno na obrázku. Všechny body stejně vzdálené od a i b leží na přímce p – ose úsečky ab. Oblasti B_a a B_b jsou tedy tvořeny polorovinami ohraničenými osou p. Tedy obecně tvoří množina všech bodů bližších k x_i než k x_j nějakou polorovinu. Oblast B_i obsahuje všechny body, které jsou současně bližší k x_i než ke všem ostatním bodům x_j – tedy leží ve všech polorovinách současně. Každá z oblastí B_i je tvořena průnikem $n - 1$ polorovin, tedy je to (možná neomezený) mnohoúhelník.⟨3⟩ Příklad Voroného diagramu je naznačen na obrázku. Zadané body jsou označeny prázdnými kroužky a hranice oblastí B_i jsou vyznačené černými čárami.

⟨2⟩ Zlí jazykové by řekli, že medvědi jsou moc líní a nebo v mapách ani číst neumí!

⟨3⟩ Slyšeli jste už o lineárním programování? Jak název vůbec nenapoví, *lineární programování* je teorii zabývající se řešením a vlastnostmi soustav lineárních nerovnic. Lineární program je popsaný lineární funkcí, kterou chceme maximalizovat za podmínek popsaných soustavou lineárních nerovnic. Každá nerovnice určuje poloprostor, ve kterém se přípustná řešení nachází. Protože přípustné řešení splňuje všechny nerovnice zároveň, je množina všech přípustných řešení (možná neomezený) mnohostěn, obecně ve veliké dimenzi $\mathbb{R}^d$, kde d je počet proměnných. Množiny B_i lze snadno popsat jako množiny všech přípustných řešení lineárních programů pomocí výše ukázaných polorovin. Na závěr poznamenejme, že dlouho otevřená otázka, zda lze nalézt optimální řešení lineárního programu v polynomiálním čase, byla pozitivně vyřešena – je znám polynomiální algoritmus, kterému se říká *metoda vnitřního bodu*. Na druhou stranu, pokud chceme najít přípustné celočíselné řešení, je úloha NP-úplná a je jednoduché na ni převést spoustu optimalizačních problémů. Dokázat NP-těžkost není příliš těžké. Na druhou stranu ukázat, že tento problém leží v NP, není vůbec jednoduché.



Body bližší k a než b.



Voroného diagram.

Není náhoda, pokud vám hranice oblastí připomíná rovinný graf. Jeho vrcholy jsou body, které jsou stejně vzdálené od alespoň tří zadaných bodů. Jeho stěny jsou oblasti B_i. Jeho hrany jsou tvořeny částí hranice mezi dvěma oblastmi – body, které mají dvě oblasti společné. Obecně průnik dvou oblastí může být, v závislosti na jejich sousedění, prázdný, bod, úsečka, polopřímka nebo dokonce celá přímka. V dalším textu si představme, že celý Voroného diagram uzavřeme do dostatečně velkého obdélníka,⟨4⟩ čímž dostaneme omezený rovinný graf.

Poznamenejme, že přerušované čáry tvoří hrany duálního rovinného grafu s vrcholy v zadaných bodech. Hrany spojují sousední body na kružnicích, které obsahují alespoň tři ze zadaných bodů. Například na obrázku dostáváme skoro samé trojúhelníky, protože většina kružnic obsahuje přesně tři zadané body. Avšak nalezneme i jeden čtyřúhelník, jehož vrcholy leží na jedné kružnici.

Zkusíme nyní odhadnout, jak velký je rovinný graf popisující Voroného diagram. Podle slavné Eulerovy formule má každý rovinný graf nejvýše lineárně mnoho vrcholů, hran a stěn – pro v vrcholů, e hran a f stěn je $e \leq 3v-6$ a navíc $v+f = e+2$. Tedy složitost diagramu je lineární vzhledem k počtu zadaných bodů $n = f$, $\mathcal{O}(n)$. Navíc Voroného diagram lze zkonstruovat v čase $\mathcal{O}(n \log n)$, například pomocí zametání roviny nebo metodou rozděl a panuj. Tím se však zabývat nebudeme,⟨5⟩ místo toho si ukážeme, jak v již spočteném Voroného diagramu rychle hledat nejbližší body.

⟨4⟩ Přeci jenom i celá Arktida je omezeně velká.

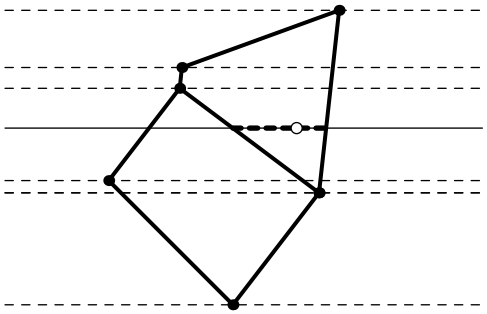
⟨5⟩ Pro zvídavé, kteří nemají zkoušku druhý den ráno: Detaily naleznete v zápiscích z předloňského ADSka.

Lokalizace bodu uvnitř mnohoúhelníkové sítě

Problém medvědů je najít v medvědí mapě co nejrychleji nejbližší iglů. Máme v rovině síť tvořenou mnohoúhelníky. Chceme pro jednotlivé body rychle rozhodovat, do kterého mnohoúhelníku patří. Naše řešení budeme optimalizovat pro jeden pevný rozklad a obrovské množství různých dotazů, které chceme co nejrychleji zodpovědět.⟨6⟩ Nejprve předzpracujeme zadané mnohoúhelníky a vytvoříme strukturu, která nám umožní rychlé dotazy na jednotlivé body.

Ukažme si pro začátek řešení bez předzpracování. Rovinu budeme zametat přímkou shora dolů. Podobně jako při hledání průsečíků úseček, udržujeme si průřez přímkou. Všimněte si, že tento průřez se mění jenom ve vrcholech mnohoúhelníků. Ve chvíli, kdy narazíme na hledaný bod, podíváme se, do kterého intervalu v průřezu patří. To nám dá mnohoúhelník, který nahlásíme. Průřez budeme uchovávat ve vyhledávacím stromě. Takové řešení má složitost $\mathcal{O}(n \log n)$ na dotaz, což je hrozně pomalé.

Předzpracování bude fungovat následovně. Jak je naznačeno na obrázku přerušovanými čárami, rozřežeme si celou rovinu na pásy, během kterých se průřez přímkou nemění. Pro každý z nich si pamatujeme stav stromu popisující, jak vypadal průřez při procházení tímto pásem. Když chceme lokalizovat nějaký bod, nejprve půlením nalezneme pás, ve kterém se nachází. Poté položíme dotaz na příslušný strom. Strom procházíme a po cestě si dopočítáme souřadnice průřezu, až lokalizujeme správný interval v průřezu. Dotaz dokážeme zodpovědět v čase $\mathcal{O}(\log n)$. Hledaný bod je na obrázku naznačen prázdným kolečkem a nalezený interval v průřezu je vytažený tučně.



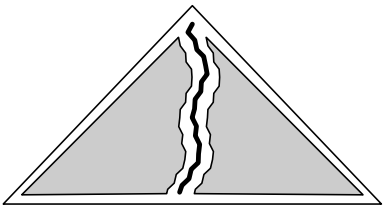
Mnohoúhelníky rozřezané na pásy.

Jenomže naše řešení má jeden háček: Jak zkonstruovat jednotlivé verze stromu dostatečně rychle? K tomu napomohou *částečně perzistentní* datové struktury.

⟨6⟩ Představujme si to třeba tak, že medvědům zprovozníme server. Ten jednou schroustá celou mapu a potom co nejrychleji odpovídá na jejich dotazy. Medvědi tak nemusí v mapách nic hledat, stačí se připojit na server a počkat na odpověď.

Pod perzistencí se myslí, že struktura umožňuje uchovávat svoji historii. Částečně perzistentní struktury nemohou svoji historii modifikovat.

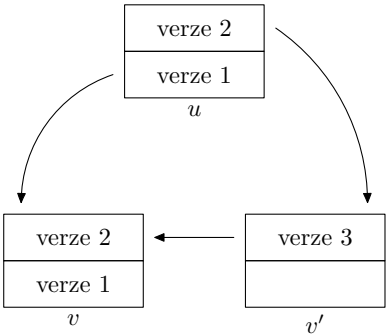
Popíšeme si, jak vytvořit perzistentní strom s pamětí $\mathcal{O}(\log n)$ na změnu. Pokud provádíme operaci na stromě, mění se jenom malá část stromu. Například při vkládání do stromu se mění jenom prvky na jedné cestičce z kořene do listu (a případně rotací i na jejím nejbližším okolí). Proto si uložíme upravenou cestičku a zbytek stromu budeme sdílet s předchozí verzí. Na obrázku je vyznačena cesta, jejíž vrcholy jsou upravovány. Šedě označené podstromy navěšené na tuto cestu se nemění, a proto na ně stačí zkopírovat ukazatele. Mimochodem změny každé operace se složitostí $\mathcal{O}(k)$ lze zapsat v paměti $\mathcal{O}(k)$, prostě operace nemá tolik času, aby mohla pozměnit příliš velikou část stromu.



Jedna operace mění pouze okolí cesty – navěšené podstromy se nemění.

Celková časová složitost je tedy $\mathcal{O}(n \log n)$ na předzpracování Voroného diagramu a vytvoření persistentního stromu. Kvůli persistenci potřebuje toto předzpracování paměť $\mathcal{O}(n \log n)$. Na dotaz spotřebujeme čas $\mathcal{O}(\log n)$, neboť nejprve vyhledáme půlením příslušný pás a poté položíme dotaz na příslušnou verzi stromu. Rychleji to ani provést nepůjde, neboť potřebujeme utřídit souřadnice bodů.

Lze to lépe? Na závěr poznamenejme, že se umí provést výše popsaná persistence vyhledávacího stromu v amortizované paměti $\mathcal{O}(1)$ na změnu. Ve stručnosti naznačíme myšlenku. Použijeme stromy, které při insertu a deletu provádí amortizovaně jenom konstantně mnoho úprav své struktury. To nám například zaručí 2-4 stromy z přednášky a podobnou vlastnost lze dokázat i o červeno-černých stromech. Při změně potom nebudeme upravovat celou cestu, ale upravíme jenom jednotlivé vrcholy, kterých se změna týká. Každý vrchol stromu si v sobě bude pamatovat až dvě své verze. Pokud chceme vytvořit třetí verzi, vrchol zkopírujeme stranou. To však může vyvolat změny v jeho rodičích až do kořene. Situace je naznačena na obrázku. Při vytvoření nové verze 3 pro vrcholu v vytvoříme jeho kopii v', do které uložíme tuto verzi. Avšak musíme také změnit rodiče u, kterému vytvoříme novou verzi ukazující na v'. Abychom dosáhli kýžené konstantní paměťové složitosti, pomůže potenciálový argument – změn se provádí amortizovaně jenom konstantně mnoho. Navíc si pro každou verzi pamatujeme její kořen, ze kterého máme dotaz spustit.



Vytvoření nové verze vrcholu.