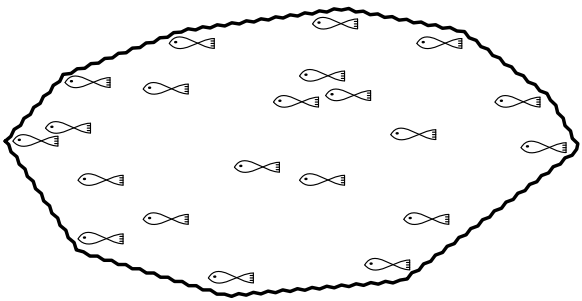
1. Geometrické algoritmy

Ukážeme si několik základních algoritmů na řešení geometrických problémů v rovině. Proč zrovna v rovině? Inu, jednorozměrné problémy bývají triviální a naopak pro vyšší dimenze jsou velice komplikované. Rovina je proto rozumným kompromisem mezi obtížností a zajímavostí.

Celou kapitolou nás bude provázet pohádka ze života ledních medvědů. Pokusíme se vyřešit jejich „každodenní“ problémy ...

1.1. Hledání konvexního obalu

Daleko na severu žili lední medvědi. Ve vodách tamního moře byla hojnost ryb a jak je známo, ryby jsou oblíbenou pochoutkou ledních medvědů. Protože medvědi z naší pohádky rozhodně nejsou ledajací a ani chytrost jim neschází, rozhodli se všechny ryby pochytat. Znají přesná místa výskytu ryb a rádi by vyrobili obrovskou síť, do které by je všechny chytili. Pomozte medvědům zjistit, jaký nejmenší obvod taková síť může mít.



Obr. 1.1: Problém ledních mědvědů: Jaký je nejmenší obvod sítě?

Neboli v řeči matematické, chceme pro zadanou množinu bodů v rovině nalézt její konvexní obal.

Definice: O množině bodů řekneme, že je *konvexní*, pokud pro každé dva body obsahuje i celou úsečku mezi nimi. *Konvexní obal* dané množiny bodů M je nejmenší konvexní množina, která obsahuje všechny body množiny M.[1]

[1] Nejmenší myslíme vzhledem k inkluzi, tedy je to průnik všech konvexních množin obsahujících všechny body z M.

Pamatujete si na lineární obaly ve vektorových prostorech? Lineární obal množiny vektorů je nejmenší vektorový podprostor, který tyto vektory obsahuje. Není náhoda, že tato definice připomíná definici konvexního obalu. Na druhou stranu každý vektor z lineárního obalu lze vyjádřit jako lineární kombinaci daných vektorů.

$n = 1$ $n = 2$ $n = 3$ $n = 4$

Obr. 1.2: Konvexní obaly malých množin

Snadno si všimneme, že konvexní obal konečné množiny bodů je vždy nějaký konvexní mnohoúhelník. Navíc víme, že vrcholy leží v některých ze zadaných bodů. Pro malé počty bodů to bude vypadat následovně:

Naším úkolem tedy bude najít tento mnohoúhelník a vypsat na výstup jeho vrcholy v pořadí, ve kterém na mnohoúhelníku leží (buď po směru hodinových ručiček nebo proti němu). Pro jednoduchost budeme konvexní obal říkat přímo tomuto seznamu vrcholů.

Navíc budeme předpokládat, že všechny body mají různé x-ové souřadnice.⟨2⟩ Tím máme zajištěné, že existují dva body, nejlevější a nejpravější, a ty určitě leží na konvexním obalu.

Algoritmus na nalezení konvexního obalu funguje na principu, kterému se obvykle říká *zametání roviny*. Procházíme rovinu zleva doprava („zametáme ji přímkou“) a udržujeme si konvexní obal těch bodů, které jsme už prošli.

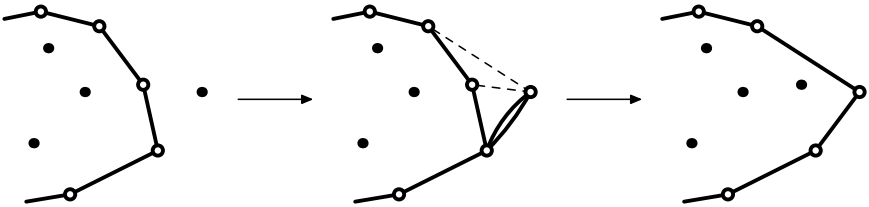
Na počátku máme konvexní obal jednobodové množiny, což je samotný bod. Nechť tedy už známe konvexní obal prvních $k-1$ bodů a chceme přidat k-tý bod. Ten určitě na novém konvexním obalu bude ležet (je nejpravější), ale jeho přidání k minulému obalu může způsobit, že hranice přestane být konvexní. To ale lze snadno napravit – stačí z hranice odebírat body po směru a proti směru hodinových ručiček tak dlouho, než opět bude konvexní.

Například na následujícím obrázku nemusíme po směru hodinových ručiček odebrat ani jeden bod, obal je v pořádku. Naopak proti směru ručiček musíme odebrat dokonce dva body.

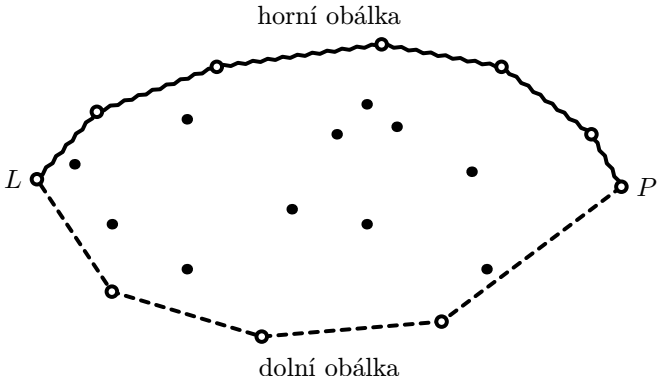
Podle tohoto principu už snadno vytvoříme algoritmus. Aby se lépe popisoval, rozdělíme konvexní obal na *horní obálku* a *dolní obálku* – to jsou části, které vedou od nejlevějšího bodu k nejpravějšímu „horem“ a „spodem“.

Podobně platí i pro konvexní obaly, že každý bod z obalu je konvexní kombinací daných bodů. Ta se liší od lineární v tom, že všechny koeficienty jsou v intervalu $[0, 1]$ a navíc součet všech koeficientů je 1. Tento algebraický pohled může mnohé věci zjednodušit. Zkuste dokázat, že obě definice konvexního obalu jsou ekvivalentní.

⟨2⟩ To si můžeme dovolit předpokládat, neboť se všemi body stačí nepatrně pootočit. Tím konvexní obal určitě nezměníme a x-ové souřadnice se již budou lišit. Pořadí otočených bodů podle x-ové souřadnice přitom odpovídá lexikografickému pořadí (druhotně podle souřadnice y) původních bodů. Takže stačí v našem algoritmu vyměnit třídění za lexikografické a bude fungovat obecně.



Obr. 1.3: Přidání bodu do konvexního obalu



Obr. 1.4: Horní a dolní obálka konvexního obalu

Obě obálky jsou lomené čáry, navíc horní obálka pořád zatáčí doprava a dolní naopak doleva. Pro udržování bodů v obálkách stačí dva zásobníky. V k-tém kroku algoritmu přidáme k-tý bod zvlášť do horní i dolní obálky. Přidáním k-tého bodu se však může porušit směr, ve kterém obálka zatáčí. Proto budeme nejprve body z obálky odebírat a k-tý bod přidáme až ve chvíli, kdy jeho přidání směr zatáčení neporuší.

Algoritmus KonvexníObal

1. Setřídíme body podle x-ové souřadnice, označme body $b_1, \ldots, b_n$.
2. Vložíme do horní a dolní obálky bod b_1: $H = D = (b_1)$.
3. Pro každý další bod $b = b_2, \ldots, b_n$:
4. Přepočítáme horní obálku:
5. Dokud $|H| \geq 2$, $H = (\ldots, h_{k-1}, h_k)$ a úhel $h_{k-1}h_kb$ je orientovaný doleva:
6. Odebereme poslední bod h_k z obálky H.
7. Přidáme bod b do obálky H.
8. Symetricky přepočteme dolní obálku (s orientací doprava).
9. Výsledný obal je tvořen body v obálkách H a D.

Rozebereme si časovou složitost algoritmu. Setřídit body podle x-ové souřadni-

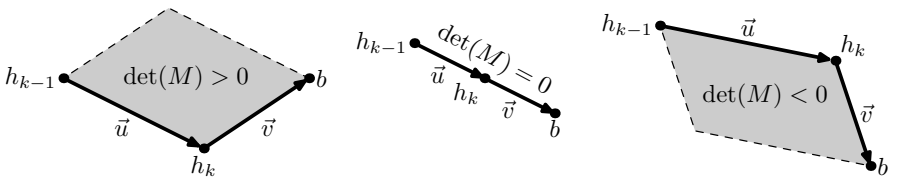
ce dokážeme v čase $\mathcal{O}(n \log n)$. Přidání dalšího bodu do obálek trvá lineárně vzhledem k počtu odebraných bodů. Zde využijeme obvyklý postup: Každý bod je odebrán nejvýše jednou, a tedy všechna odebrání trvají dohromady $\mathcal{O}(n)$. Konvexní obal dokážeme sestrojit v čase $\mathcal{O}(n \log n)$, a pokud bychom měli seznam bodů již utříděný, dokážeme to dokonce v $\mathcal{O}(n)$.

Algebraický dodatek: Jak zjistit orientaci úhlu? Ukážeme si jednoduchý způsob založený na lineární algebře. Budou se k tomu hodit vlastnosti determinantu. Absolutní hodnota determinantu je objem rovnoběžnostěnu určeného řádkovými vektory matice. Důležitější však je, že znaménko determinantu určuje „orientaci“ vektorů – zda je levotočivá či pravotočivá. Protože náš problém je rovinný, budeme uvažovat determinanty matic 2×2.

Uvažme souřadnicový systém v rovině, jehož x-ová souřadnice roste směrem doprava a y-ová směrem nahoru. Chceme zjistit orientaci úhlu $h_{k-1}h_kb$. Označme $\vec{u} = (x_1, y_1)$ rozdíl souřadnic bodů h_k a h_{k-1} a podobně $\vec{v} = (x_2, y_2)$ rozdíl souřadnic bodů b a h_k. Matici M definujeme následovně:

$$M = \begin{pmatrix} \vec{u} \\ \vec{v} \end{pmatrix} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}.$$

Úhel $h_{k-1}h_kb$ je orientován doleva, právě když $\det M = x_1y_2 - x_2y_1$ je nezáporný, a spočítat hodnotu determinantu je jednoduché. Možné situace jsou nakresleny na obrázku. Poznamenejme, že k podobnému vzorci se lze také dostat přes vektorový součin vektorů $\vec{u}$ a $\vec{v}$.



Obr. 1.5: Jak vypadají determinanty různých znamének v rovině

1.2. Rychlejší algoritmus

Také vám vrtá hlavou, zda jde konvexní obal sestrojit rychleji? Obecně ne, alespoň pokud chceme body na konvexním obalu vydat v pořadí, v jakém se na hranici nacházejí.⟨3⟩ Pokud ovšem na konvexním obalu většina bodů neleží, jde to rychleji.

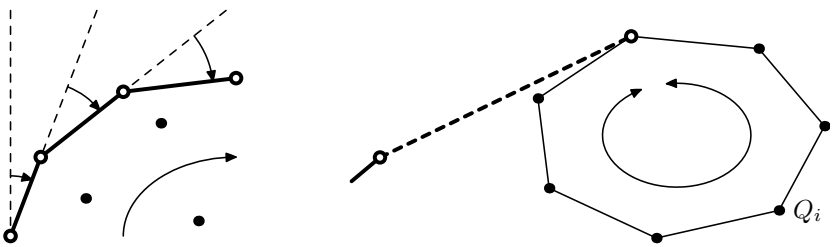
⟨3⟩ Pak bychom totiž uměli třídit reálná čísla v čase lepším než $\Omega(n \log n)$, což se ví, že nejde (náš dolní odhad složitosti třídění z minulého semestru na to ovšem nestačí). Zkuste převod z třídění na hledání konvexního obalu vymyslet sami.

Na přednášce to sice nebylo (a u zkoušky nebude), ale zde si ukážeme nejrychlejší známý algoritmus, jehož autorem je T. Chan a který funguje v čase $\mathcal{O}(n \log h)$, kde h značí počet bodů ležících na konvexním obalu. Překvapivě bude docela snadný.

Předpokládejme, že bychom znali velikost konvexního obalu h. Rozdělíme body libovolně do $\lceil n/h \rceil$ množin $Q_1, \ldots, Q_k$ tak, aby v každé množině bylo nejvýše h bodů. Pro každou z těchto množin nalezneme konvexní obal pomocí výše popsaného algoritmu. To dokážeme pro jednu v čase $\mathcal{O}(h \log h)$ a pro všechny v čase $\mathcal{O}(n \log h)$. V druhé fázi spustíme tyto předpočítané obaly slepíme do jednoho pomocí tak zvaného *provázkového algoritmu.* Ten se opírá o následující pozorování:

Pozorování: Úsečka spojující dva body a a b leží na konvexním obalu, právě když všechny ostatní body leží na téže straně přímky proložené touto úsečkou.

Algoritmu se říká provázkový, protože svojí činností připomíná namotávání provázku podél konvexního obalu. Začneme s bodem, který na konvexním obalu určitě leží, to je třeba ten nejlevější. V každém kroku nalezneme následující bod po obvodu konvexního obalu. To uděláme například tak, že projdeme všechny body a vybereme ten, který svírá nejmenší úhel s poslední stranou konvexního obalu. Nově přidaná úsečka vyhovuje pozorování a proto do konvexního obalu patří. Po h krocích se dostaneme zpět k nejlevějšímu bodu a výpočet ukončíme. V každém kroku potřebujeme projít všechny body a vybrat následníka, což dokážeme v čase $\mathcal{O}(n)$. Celková složitost algoritmu je tedy $\mathcal{O}(n \cdot h)$.



Obr. 1.6: Provázkový algoritmus a jeho použití v předpočítaném obalu

Provázkový algoritmus funguje, ale má jednu obrovskou nevýhodu – je totiž ukrutně pomalý. Kýženého zrychlení dosáhneme, pokud použijeme předpočítané konvexní obaly. Ty umožní rychleji hledat následníka. Pro každou z množin Q_i najdeme zvlášť kandidáta a poté z nich vybereme toho nejlepšího. Možný kandidát vždy leží na konvexním obalu množiny Q_i. Využijeme toho, že body obalu jsou „uspořádané“, i když trochu netypicky do kruhu. Kandidáta můžeme hledat metodou půlení intervalu, jen detaily jsou maličko složitější, než je obvyklé. Jak půlit, zjistíme podle směru zatáčení konvexního obalu. Detaily si rozmyslí čtenář sám.

Časová složitost půlení je $\mathcal{O}(\log h)$ pro jednu množinu. Množin je nejvýše $\mathcal{O}(\frac{n}{h})$, tedy následující bod konvexního obalu nalezneme v čase $\mathcal{O}(\frac{n}{h} \log h)$. Celý obal nalezneme ve slibovaném čase $\mathcal{O}(n \log h)$.

Popsanému algoritmu schází jedna důležitá věc: Ve skutečnosti většinou neznáme velikost h. Budeme proto algoritmus iterovat s rostoucí hodnotou h, dokud konvexní obal nesestrojíme. Pokud při slepování konvexních obalů zjistíme, že konvexní obal je větší než h, výpočet ukončíme. Zbývá ještě zvolit, jak rychle má h růst. Pokud by rostlo moc pomalu, budeme počítat zbytečně mnoho fází, naopak při rychlém růstu by nás poslední fáze mohla stát příliš mnoho.

V k-té iteraci položíme $h = 2^{2^k}$. Dostáváme celkovou složitost algoritmu:

$$\sum_{m=0}^{\mathcal{O}(\log\log h)} \mathcal{O}(n \log 2^{2^m}) = \sum_{m=0}^{\mathcal{O}(\log\log h)} \mathcal{O}(n \cdot 2^m) = \mathcal{O}(n \log h),$$

kde poslední rovnost dostaneme jako součet prvních $\mathcal{O}(\log\log h)$ členů geometrické řady $\sum 2^m$.

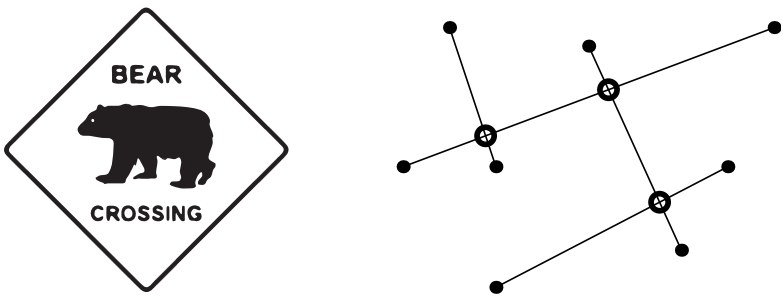
1.3. Hledání průsečíků úseček

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 ...

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.



Obr. 1.7: Problém Eskymáků: Kde všude se medvědí trasy kříží?

Pro n úseček může existovat až $\Omega(n^2)$ průsečíků.⟨4⟩ 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.

⟨4⟩ Zkuste takový příklad zkonstruovat.

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.

Algoritmus funguje na principu zametání roviny, podobně jako hledání konvexního obalu. 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 bodu, kde se něco zajímavého děje.

Tím zajímavým jsou *začátky úseček*, *konce úseček* a *průsečíky úseček*. Dohromady jim budeme říkat *události.* Po utřídění známe pro první dva typy událostí pořadí, v jakém se objeví. Průsečíkové události budeme objevovat průběžně.

V každém kroku si pamatujeme *průřez* P – posloupnost úseček zrovna protnutých zametací přímkou. Tyto úsečky máme utříděné zleva doprava. Navíc si udržujeme kalendář K budoucích událostí. V něm budou naplánovány všechny začátky a konce ležící pod zametací přímkou. Navíc se pro každou dvojici úseček podíváme, zda se pod zametací přímkou protnou, a pokud ano, tak takový průsečík také naplánujeme. (Mohli bychom přitom úsečky považovat za polopřímky, falešné průsečíky totiž beztak smažeme dříve, než na ně dojde řada.)

Algoritmus pro hledání průniků úseček pak funguje následovně:

Algoritmus Průsečíky

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 naplánované průsečíkové události (nejvýše dvě odebereme a dvě nové přidáme).

Zbývá rozmyslet, jaké datové struktury použijeme pro reprezentaci průřezu a kalendáře. S kalendářem je to snadné, ten můžeme uložit například do haldy. Co potřebujeme dělat s průřezem? Vkládat a odebírat úsečky, ale také k dané úsečce nalézt jejího předchůdce a následníka (to je potřeba při plánování průsečíkových událostí). Nabízí se využít vyhledávací strom, ovšem jako klíče v něm nemohou vystupovat x-ové souřadnice úseček (respektive jejich průsečíků se zametací přímkou). Ty se totiž při každém posunutí našeho „koštěte“ mohou všechny změnit.

Uložíme tedy do vrcholů místo souřadnic jen odkazy na úsečky. Ty se nemění (a mezi událostmi se nemění ani jejich pořadí, což je důležité) a kdykoliv nějaká

operace se stromem navštíví jeho vrchol, dopočítáme aktuální souřadnici úsečky a podle toho se rozhodneme, zda se vydat doleva nebo doprava.

Průřez obsahuje vždy nejvýše n úseček, takže operace se stromem budou trvat $\mathcal{O}(\log n)$. V kalendáři se nachází nejvýše n začátků a konců a nejvýše n průsečíkových událostí (ty plánujeme pro dvojice úseček sousedících v průřezu, a těch je vždy nejvýše n). Operace s kalendářem proto trvají také $\mathcal{O}(\log n)$.

Při vyhodnocování každé události provedeme $\mathcal{O}(1)$ operací s datovými strukturami, takže událost zpracujeme v čase $\mathcal{O}(\log n)$. Všech událostí dohromady je přitom $\mathcal{O}(n+p)$, a proto celý algoritmus doběhne v čase $\mathcal{O}((n+p)\log n)$.

Cvičení: Popište, jak algoritmus upravit, aby nepotřeboval předpoklad obecné polohy úseček. Podobně jako u konvexního obalu, i zde stačí jednoduché úpravy.

Na závěr poznamenejme, že existuje efektivnější, byť daleko komplikovanější, algoritmus od Chazella dosahující časové složitosti $\mathcal{O}(n \log n + p)$.

1.4. Hledání nejbližších bodů a Voroného diagramy

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

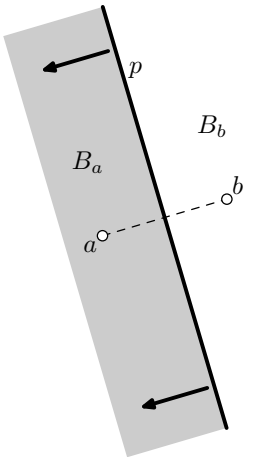
Eskymáci tráví většinu času doma, ve svém iglů. Takový medvěd je na své toulce zasněženou krajinou, když tu se najednou rozhodne navštívit nějakého Eskymáka. Proto se podívá do své medvědí mapy a nalezne nejbližší iglů. Má to ale jeden háček, iglů jsou spousty a medvěd by dávno usnul, než by nejbližší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

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:

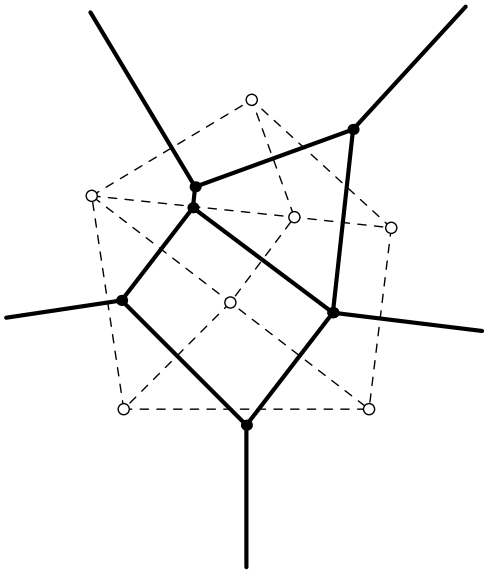
$$B_i = \left\{ y \in \mathbb{R}^2 \mid \forall j : \rho(x_i, y) \leq \rho(x_j, y) \right\},$$

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. Příklad Voroného diagramu je naznačen na obrázku. Zadané body jsou označeny prázdnými kroužky a hranice oblastí B_i jsou vyznačené černými čárami.



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, 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. Pro rovinné grafy bez násobných hran obecně platí, že mají nejvýše lineárně mnoho hran vzhledem k vrcholům. My ovšem neznáme počet vrcholů, nýbrž počet stěn – každá stěna odpovídá jednomu ze zadaných bodů. Proto odhad počtu hran použijeme na duál našeho grafu, čímž prohodíme vrcholy se stěnami a hran zůstane stejně. Žádnou násobnou hranu jsme tím nepřidali, ta by totiž odpovídala stěně velikosti 2 ve Voroného diagramu a takové neexistují, neboť každá stěna je ohraničena rovnými čarami.

Voroného diagram pro n zadaných bodů je tedy velký $\mathcal{O}(n)$. Dodejme, že ho 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.

1.5. 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 shora dolů vodorovnou přímkou. 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.

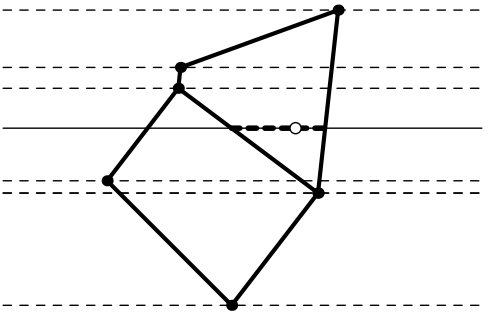
Předzpracování bude fungovat následovně. Jak je naznačeno na obrázku přerušovanými čárami, rozřežeme si celou rovinu na pásy, během kterých se průřez přímkou nemění. Pro každý z nich si pamatujeme stav stromu tak, jak vypadal průřez při procházení tímto pásem. Když chceme lokalizovat nějaký bod, nejprve půlením nalezneme pás, ve kterém se nachází y-ová souřadnice bodu. 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.

Kdybychom si ovšem uchovávali stavy stromu tak, že bychom si pro každý pás pořídili kopii celého stromu, spotřebovali bychom jenom kopírováním stromů čas i paměť $\Theta(n^2 \log n)$. Místo toho si pořídíme *persistentní* vyhledávací strom – ten si pamatuje historii všech svých změn a umí v ní vyhledávat. Přesněji řečeno, po každé operaci, která mění stav stromu, vznikne nová *verze* stromu a operace pro hledání dostanou jako další parametr identifikátor verze, ve které mají hledat.

Popíšeme jednu z možných konstrukcí persistentního stromu. Uvažujme obyčejný vyhledávací strom, řekněme AVL strom. Rozhodneme se ale, že jeho vrcholy nikdy nebudeme měnit, abychom neporušili zaznamenanou historii. Místo toho si pořídíme kopii vrcholu a tu změníme. Musíme ovšem změnit ukazatel na daný vrchol,

⟨5⟩ Pro zvídavé, kteří nemají zkoušku druhý den ráno: Detaily naleznete v zápiscích z ADS z roku 2007/2008.

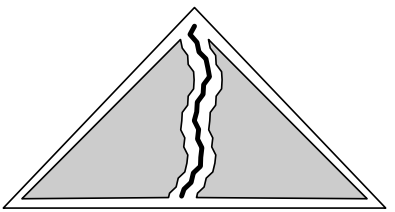
⟨6⟩ Představujme si to třeba tak, že medvědům zprovozníme server. Ten jednou schroustá celou mapu a potom co nejrychleji odpovídá na jejich dotazy. Medvědi tak nemusí v mapách nic hledat, stačí se připojit na server a počkat na odpověď.



Obr. 1.8: Mnohoúhelníky rozřezané na pásy

aby ukazoval na kopii. Proto zkopírujeme i jeho otce a upravíme v něm ukazatel. Tím pádem musíme upravit i ukazatel na otce, atd., až se dostaneme do kořene. Kopie kořene se pak stane identifikátorem nové verze.

Zkopírovali jsme tedy celou cestu mezi kořenem stromu a upravovaným vrcholem. Uchování jedné verze nás tedy strojí čas $\mathcal{O}(\log n)$ a prostor taktéž $\mathcal{O}(\log n)$. Ještě nesmíme zapomenout, že po každé operaci následuje vyvážení stromu. To ovšem upravuje pouze vrcholy, které leží v konstantní vzdálenosti od cesty z místa úpravy do kořene, takže jejich zkopírováním časovou ani prostorou složitost nezhoršíme.



Obr. 1.9: 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.

Lze to lépe? Na závěr poznamenejme, že spotřeba paměti $\Theta(\log n)$ na uložení jedné verze je zbytečně vysoká. Existuje o něco chytřejší konstrukce persistentního stromu, které stačí konstantní paměť, alespoň amortizovaně. Nastíníme, jak funguje.

Nejprve si pořídíme vyhledávací strom, který při každém vložení nebo smazání prvku provede jen amortizovaně konstantní počet *strukturálních změn* (to jsou změny hodnot a ukazatelů, zkrátka všeho, podle čeho se řídí vyhledávání a co je tedy potřeba verzovat; změna počítadla ve vrcholu u AVL-stromu tedy strukturální není). Tuto vlastnost mají třeba 2,4-stromy nebo některé varianty červeno-černých stromů.

Nyní ukážeme, jak jednu strukturální změnu zaznamenat v amortizovaně konstantním prostoru. Každý vrchol stromu si tentokrát bude pamatovat dvě své verze (spolu s časy jejich vzniku). Při průchodu od kořene porovnáme čas vzniku těchto verzí s aktuálním časem a vybereme si správnou verzi. Pokud potřebujeme zaznamenat novou verzi vrcholu, buďto na ni ve vrcholu ještě je místo, nebo není a v takovém případě vrchol zkopírujeme, což vynutí změnu ukazatele v rodiči, a tedy i vytvoření nové verze rodiče, atd. až případně do kořene. Identifikátorem verze celé datové struktury bude ukazatel na aktuální kopii kořene.

Jedna operace může v nejhorším případě způsobit zkopírování všech vrcholů až do kořene, ale jednoduchým potenciálovým argumentem lze dokázat, že počet kopií bude amortizovaně konstantní.