

# 1. Geometrické algoritmy

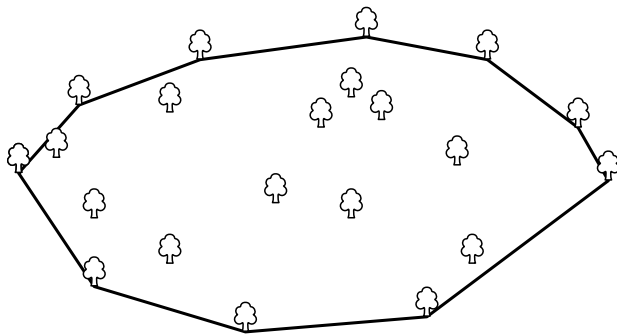
Mnoho praktických problémů má geometrickou povahu: můžeme chtít oplotit jabloňový sad nejkratším možným plotem, nalézt k dané adrese nejbližší poštovní úřadovnu, nebo třeba naplánovat trasu robota trojrozměrnou budovou.

V této kapitole ukážeme několik základních technik, jak geometrické algoritmy navrhovat. Soustředíme se přitom na problémy v rovině: ty jednorozměrné bývají triviální, vícerozměrné naopak mnohem náročnější.

## 1.1. Konvexní obal

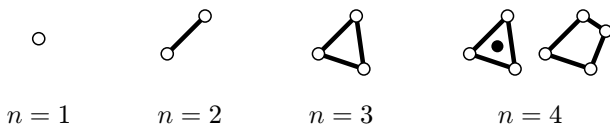
Byl jest jednou jeden jabloňový sad. Každý podzim v něm dozrávala kulaťoučká červeňoučká jablíčka, tak dobrá, že je za noci chodili otrhávat všichni tuláci z okolí. Aby alespoň nějaká jablka vydržela do sklizně, nabízí se sad oplotit. Chceme postavit plot takový, aby obklopil všechny jabloně, a spotřebovali jsme na něj co nejméně pletiva.

Méně poeticky řečeno: Je dána nějaká množina  $n$  bodů v euklidovské rovině a chceme nalézt co nejkratší uzavřenou křivku, uvnitř níž leží všechny body. Geometrická intuice nám napovídá, že hledaná křivka bude konvexní mnohoúhelník, v jehož vrcholech budou některé ze zadaných bodů, zatímco ostatní body budou ležet uvnitř mnohoúhelníka, případně na jeho hranách. Proto se mu říká *konvexní obal* zadaných bodů. (Pokud se nechcete odvolávat na intuici, trochu formálnější pohled najdete ve cvičeních 3 a 4.)



Obr. 1.1: Jabloňový sad

Pro malé počty bodů bude konvexní obal vypadat následovně:



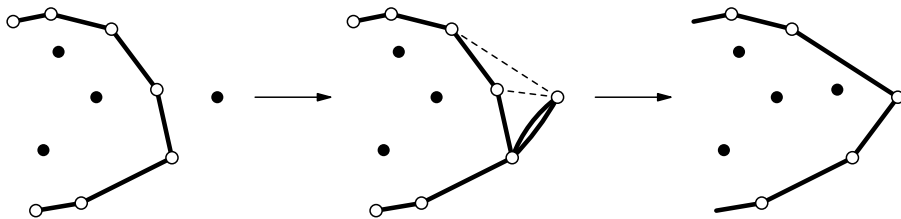
Naším úkolem tedy bude najít obal a vypsát na výstup jeho vrcholy v pořadí, ve kterém na leží na jeho hranici (buď po směru hodinových ručiček, nebo proti němu). Pro jednoduchost budeme konvexní obal říkat přímo tomuto seznamu vrcholů.

Prozatím budeme předpokládat, že všechny body mají různé  $x$ -ové souřadnice. Existuje tedy jednoznačně určený nejlevější a nejpravější bod a ty oba musí ležet na konvexním obalu. (Obecně se hodí geometrické problémy řešit nejdříve pro body, které jsou v nějakém vhodném smyslu v obecné poloze, a teprve pak se starat o speciální případy.)

Použijeme princip, kterému se obvykle říká *zametání roviny*. Budeme procházet rovinu zleva doprava („zametat ji přímkou“) a udržovat 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 odebrat 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.



Obr. 1.2: Přidání bodu do konvexního obalu

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“.

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 odebrat 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  $b_1, \dots, b_n$ .

2. Vložíme do horní a dolní obálky bod  $b_1$ :  $H = D = (b_1)$ .
3. Pro každý další bod  $b = b_2, \dots, b_n$ :
4.     Přepočítáme horní obálku:
5.         Dokud  $|H| \geq 2$ ,  $H = (\dots, h_{k-1}, h_k)$  a úhel  $h_{k-1}h_k b$  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 časovou složitost algoritmu. Setřídít body podle  $x$ -ové souřadnice 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 sestřít v čase  $\mathcal{O}(n \log n)$ , a pokud bychom měli seznam bodů již utříděný, dokážeme to dokonce v  $\mathcal{O}(n)$ .

Zbývá dořešit případy, kdy body nejsou v obecné poloze. Pokud se to stane, zkusíme všemi body nepatrně pootočit. Tím nezměníme, které body leží na konvexním obalu, 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 podle  $x$  za lexikografické.

## Orientace úhlu a determinanty

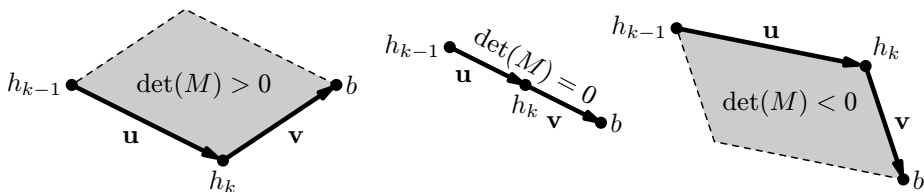
Při přepočítávání obálek jsme potřebovali testovat, zda je nějaký úhel orientovaný doleva nebo doprava. Jak na to? Ukážeme jednoduchý způsob založený na lineární algebře. Budou se k tomu hodit vlastnosti determinantu. Absolutní hodnota determinantu je objem rovnoběžnostěny určeného řádkovými vektory matice. Důležitější však je, že znaménko determinantu určuje orientaci vektorů – zda je levotočivá či pravotočivá. Protože náš problém je rovinný, budeme uvažovat determinanty matic  $2 \times 2$ .

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

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

Úhel  $h_{k-1}h_k b$  je orientován doleva, právě když  $\det M = x_1 y_2 - x_2 y_1$  je nezáporný. Možné situace jsou nakresleny na obrázku.

Determinant přitom zvládneme spočítat v konstantním čase a pokud jsou souřadnice bodů celočíselné, vystačí si i tento výpočet s celými čísly. Poznamenejme, že k podobnému vzorci se lze také dostat přes vektorový součin vektorů  $\mathbf{u}$  a  $\mathbf{v}$ .



Obr. 1.3: Jak vypadají determinanty různých znamének v rovině

## Cvičení

1. V rovině je dána množina červených a množina zelených bodů. Sestrojte přímku takovou, aby na jedné její straně ležely všechny červené body, zatímco na druhé všechny zelené.
2. V sadu je  $n$  jabloní. Sestrojte dva uzavřené ploty tak, aby každá jablň byla oplocena a celkově jste spotřebovali nejméně pletiva.
3. Naznačíme, jak konvexní obal zavést formálně. Pamatujete si ještě na lineární obaly ve vektorových prostorech? *Lineární obal*  $\mathcal{L}(X)$  množiny vektorů  $X$  je průnik všech vektorových podprostorů, které tuto množinu obsahují. Ekvivalentně je to množina všech *lineárních kombinací* vektorů z  $X$ , tedy všech součtů tvaru  $\sum_i \alpha_i x_i$ , kde  $x_i \in X$  a  $\alpha_i \in \mathbb{R}$ .

Podobně můžeme definovat *konvexní obal*  $\mathcal{C}(X)$  jako průnik všech konvexních množin, které obsahují  $X$ . Konvexní je přitom taková množina, která pro každé dva body obsahuje i celou úsečku mezi nimi. Nyní uvažujme množinu všech *konvexních kombinací*, což jsou součty tvaru  $\sum_i \alpha_i x_i$ , kde  $x_i \in X$ ,  $\alpha_i \in [0, 1]$  a  $\sum_i \alpha_i = 1$ .

Jak vypadají konvexní kombinace pro 2-bodovou a 3-bodovou množinu  $X$ ? Dokažte, že obecně je množina všech konvexních kombinací vždy konvexní a že je rovna  $\mathcal{C}(X)$ . Pro konečnou  $X$  má navíc tvar konvexního mnohoúhelníka, dokonce se to někdy používá jako jeho definice.

- 4.\* Hledejme mezi všemi mnohoúhelníky, které obsahují danou konečnou množinu bodů, ten, který má nejmenší obvod. Dokažte, že každý takový mnohoúhelník musí být konvexní a navíc rovný konvexnímu obalu množiny. (Fyzikální analogie: do bodů zatlučeme hřebíky a natáhneme kolem nich gumičku. Ta zaujme stav o nejnižší energii, tedy nejkratší křivku. My zde nechceme zabíhat do matematické analýzy, takže se omezíme na lomené čáry.)
5. Může jít sestrojit konvexní obal rychleji než v  $\Theta(n \log n)$ ? Nikoliv, alespoň pokud chceme body na konvexním obalu vypisovat v pořadí, v jakém se na jeho hranici nacházejí. Ukažte, že v takovém případě můžeme pomocí konstrukce konvexního obalu třídit reálná čísla. Naš dolní odhad složitosti třídění sice na tuto situaci nelze přímo použít, ale existuje silnější (a těžší) věta, z níž plyne, že i na třídění  $n$  reálných čísel je potřeba  $\Omega(n \log n)$  operací. Dále viz oddíl 1.5.
6. Navrhněte algoritmus pro výpočet obsahu konvexního mnohoúhelníku.

odkaz

- 7.\* Navrhnete algoritmus pro výpočet obsahu nekonvexního mnohoúhelníku. (Prozradíme, že to jde v lineárním čase.)
8. Jak o množině bodů v rovině zjistit, zda je středově symetrická?
- 9.\* Je dána množina bodů v rovině. Rozložte ji na dvě středově symetrické množiny, je-li to možné.

## 1.2. Průsečíky úseček

Nyní se zaměříme na další geometrický problém. Dostaneme  $n$  úseček a zajímá nás, které z nich se protínají a kde. Na první pohled na tom není nic zajímavého:  $n$  úseček může mít až  $\Theta(n^2)$  průsečíků, takže i triviální algoritmus, který zkusí protnout každou úsečku s každou, bude optimální. V reálných situacích nicméně počet průsečíků bývá mnohem menší, takže budeme hledat algoritmus, který má příznivou složitost nejen vzhledem k počtu bodů  $n$ , ale také k počtu průsečíků  $p$ .

Opět pro začátek předpokládejme, že úsečky leží v obecné poloze. To tentokrát znamená, že žádné tři úsečky se neprotínají v jednom bodě, průnikem každých dvou úseček je nejvýše jeden bod, krajní bod žádné úsečky neleží na jiné úsečce, a konečně také neexistují vodorovné úsečky.

Podobně jako u hledání konvexního obalu, i zde využijeme **myšlenku zametání roviny**. Budeme posouvat vodorovnou přímkou dolů a vždy, když narazíme na nový průsečík, ohlásíme jeho výskyt. Namísto **spojitého posouvání přímkou skákat po událostech**, což budou místa, kde se něco zajímavého děje.

Tím zajímavým jsou **začátky úseček**, **konce úseček** a **průsečíky úseček**. Pozice začátků a konců úseček známe předem, průsečíkové události budeme objevovat průběžně.

V každém kroku výpočtu si pamatujeme **průřez  $P$  – posloupnost úseček zrovna protnutých zametací přímkou**. Tyto úsečky máme **utříděné zleva doprava**. Navíc si udržujeme **kalendář  $K$  budoucích událostí**.

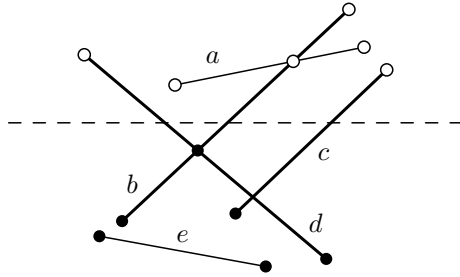
V kalendáři jsou naplánovány **všechny začátky a konce ležící pod zametací přímkou**. Navíc se **pro každou dvojici sousedních úseček v průřezu** podíváme, zda se pod zametací přímkou **protnou**, a pokud ano, tak takový průsečík také naplánujeme. Všimněme si, že těsně předtím, než se dvě úsečky protnou, musí v průřezu sousedit, takže na žádný průsečík nezapomeneme. Jen pozor na to, že **naplánované průsečíky musíme občas z plánu zase zrušit** – mezi dvojicí sousedních úseček se může dočasně vtěsnat třetí.

Jak to vypadá, můžeme sledovat na obrázku 1.4: pro čárkovanou polohu zametací přímkou leží v průřezu tučné úsečky. Kroužky odpovídají událostem: plné kroužky jsou naplánované, prázdné už nastaly. O průsečíku úseček  $c$  a  $d$  dosud nevíme, neboť se dosud nestaly sousedními.

Celý algoritmus bude vypadat následovně:

### Algoritmus PRŮSEČÍKY

1. Inicializujeme průřez  $P$  na  $\emptyset$ .



Obr. 1.4: Průřez a události v kalendáři

2. Do kalendáře  $K$  vložíme začátky a konce všech úseček.
3. Dokud  $K$  není prázdný:
  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. Přepočítáme naplánované průsečíkové události v okolí změny v  $P$  (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 nebo do vyhledávacího stromu. V každém okamžiku se v kalendáři nachází nejvýše  $3n$  událostí, takže operace s kalendářem stojí  $\mathcal{O}(\log n)$ .

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). Kdykoliv nějaká operace se stromem navštíví jeho vrchol, dopočítáme aktuální souřadnici úsečky a podle toho se rozhodneme, zda se vydat doleva nebo doprava. Jelikož průřez obsahuje vždy nejvýše  $n$  úseček, operace se stromem budou trvat  $\mathcal{O}(\log n)$ .

Při vyhodnocování každé události provedeme  $\mathcal{O}(1)$  operací s datovými strukturami, takže jednu událost zpracujeme v čase  $\mathcal{O}(\log n)$ . Všech  $\mathcal{O}(n + p)$  událostí zpracujeme v čase  $\mathcal{O}((n + p) \log n)$ , což je také časová složitost celého algoritmu.

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

## Cvičení

1. Tvrdili jsme, že  $n$  úseček může mít  $\Theta(n^2)$  průsečíků. Zkuste takový systém úseček najít.

2. Popište, jak algoritmus upravit, aby nepotřeboval předpoklad obecné polohy úseček. Především je potřeba v některých případech domyslet, co vůbec má být výstupem algoritmu.
3. Navrhněte algoritmus, který nalezne nejdelší vodorovnou úsečku ležící uvnitř daného (ne nutně konvexního) mnohoúhelníku.
4. Je dána množina obdélníků, jejichž strany jsou rovnoběžné s osami souřadnic. Spočítejte obsah jejich sjednocení.
- 5.\* Pro dané dva mnohoúhelníky vypočítejte jejich průnik (to je obecně nějaká množina mnohoúhelníků). Jednodušší verze: zjistěte, zda průnik je neprázdný.
6. Mějme množinu parabol tvaru  $y = ax^2 + bx + c$ , kde  $a > 0$ . Nalezněte všechny jejich průsečíky.
- 7.\* Co když v předchozím cvičení dovolíme i  $a < 0$ ?

## 1.3. Voroného diagramy

V daleké Arktidě bydlí Eskymáci a lední medvědi.<sup>(1)</sup> A obecnému mínění navzdory se spolu přátelí. Představte si medvěda putujícího nezměrnou polární pustinou na cestě za nejbližším iglú, kam by mohl zajít na kus řeči a pár ryb. Proto se medvědovi hodí mít po ruce Voroného diagram Arktidy:

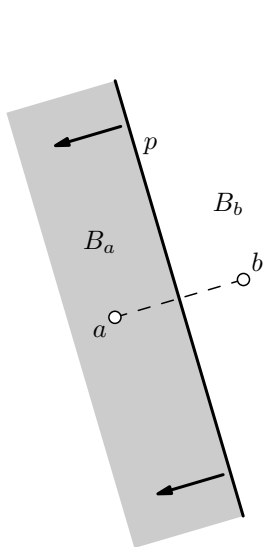
**Definice:** *Voroného diagram pro množinu bodů* neboli *míst*  $x_1, \dots, x_n \in \mathbb{R}^2$  je *systém oblastí*  $B_1, \dots, B_n \subseteq \mathbb{R}^2$ , kde  $B_i$  obsahuje ty body, jejichž vzdálenost od  $x_i$  je menší nebo rovna vzdálenostem od všech ostatních  $x_j$ .

Nahlédneme, že Voroného diagram má překvapivě jednoduchou strukturu. Nejprve uvažme, jak budou vypadat oblasti  $B_a$  a  $B_b$  pro dva body  $a$  a  $b$  (viz obrázek). Všechny body stejně vzdálené od  $a$  i  $b$  leží na přímkě  $p$  – ose úsečky  $ab$ . Oblasti  $B_a$  a  $B_b$  jsou tedy tvořeny polorovinami ohraničenými osou  $p$ .

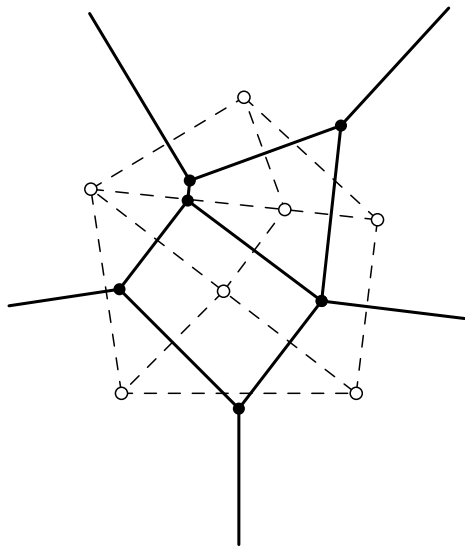
Tedy obecně tvoří množina všech bodů bližších k  $x_i$  než k  $x_j$  nějakou polorovinu. Oblast  $B_i$  obsahuje všechny body, které jsou současně bližší k  $x_i$  než ke všem ostatním bodům  $x_j$  – čili leží ve všech polorovinách současně. Každá z oblastí  $B_i$  je proto tvořena průnikem  $n - 1$  polorovin, takže je to (možná neomezený) konvexní mnohoúhelník. Příklad Voroného diagramu najdete na obrázku 1.6: zadaná místa jsou označena prázdnými kroužky, hranice oblastí  $B_i$  jsou vyznačeny černými čarami.

Není náhoda, pokud vám Voroného diagram připomíná rovinný graf. Jeho vrcholy jsou body, které jsou stejně vzdálené od alespoň tří zadaných bodů. Jeho stěny jsou oblasti  $B_i$ . Hrany jsou tvořeny částmi hranice mezi dvěma oblastmi – těmi body, které mají dvě oblasti společné. Obecně průnik dvou oblastí může být prázdná množina, bod, úsečka, polopřímka, nebo dokonce celá přímka. Dále předpokládáme,

<sup>(1)</sup> Ostatně, Arktida se podle medvědů (řecky ἀρκτος) přímo jmenuje. Jen ne podle těch ledních, nýbrž nebeských: daleko na severu se souhvězdí Velké medvědice vyjímá přímo v nadhlavíku.



Obr. 1.5: Body bližší k  $a$  než  $b$



Obr. 1.6: Voroného diagram

že celý diagram uzavřeme do dostatečně velkého obdélníku, čímž dostaneme poctivý rovinný graf s omezenými stěnami.

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 složitý takový Voroného diagram je. 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 ty neexistují, neboť každá stěna je ohraničena rovnými čarami.

Voroného **diagram pro  $n$  zadaných bodů** je tedy velký  **$\mathcal{O}(n)$** . Nyní ukážeme, jak ho zkonstruovat v čase  $\mathcal{O}(n \log n)$ .

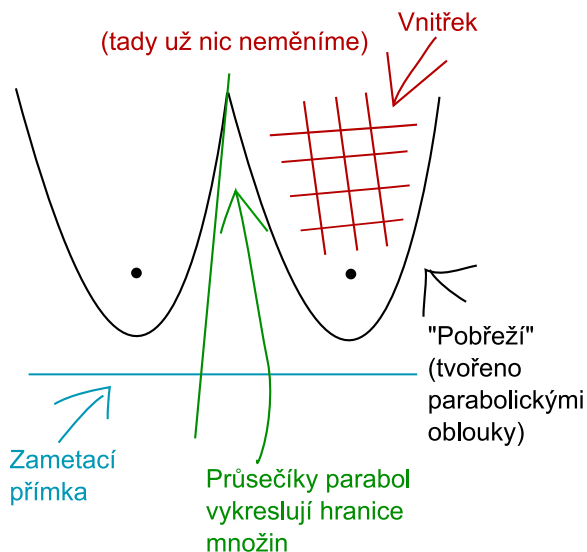
### Fortunův algoritmus\*

Budeme opět zametat rovinu shora dolů přímkou. Obvyklá představa, že nad přímkou už máme vše hotové, zde ale selže: Pokud přímka narazí na nové místo, hotová část diagramu nad přímkou se může poměrně složitě změnit. Pomůžeme si tak, že **nebudeme považovat za hotovou celou oblast nad zametací přímkou, nýbrž jen tu její část, která má blíž k některému z míst nad přímkou než ke přímce**. V této



části se už to, co jsme sestrojili, nemůže přidáváním dalších bodů změnit.

Jak vypadá hranice hotové části? Body mající stejnou vzdálenost od bodu (ohniska) jako od řídicí přímky tvoří parabolu. Hranice tudíž musí být tvořena posloupností parabolických oblouků. Krajní dva oblouky jdou do nekonečna, ostatní jsou konečné. Vzhledem k charakteristickému tvaru budeme hranici říkat *pobřeží*.



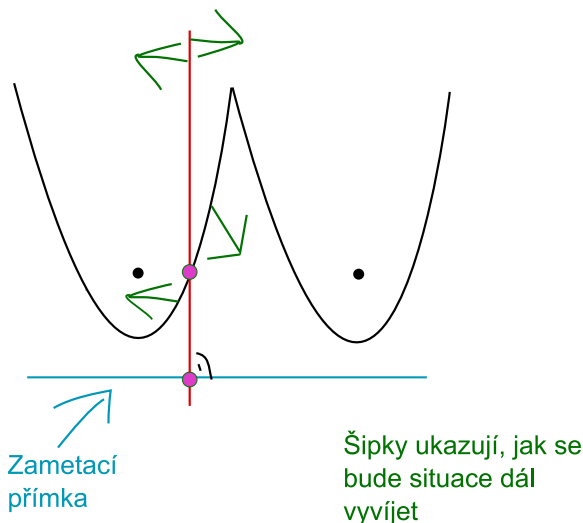
Obr. 1.7: Pobřežní linie

Posouváme-li zametací přímku, pobřeží se mění a průsečíky oblouků vykreslují hrany diagramu. Pro každý průsečík totiž platí, že vzdálený od zametací přímky stejně, jako od dvou různých míst. Tím pádem leží na hraně diagramu oddělující tato dvě místa.

Kdykoliv zametací přímka narazí na nějaké další místo, vznikne nová parabola; zprvu degenerovaná do polopřímky kolmé na zametací přímku. S dalším pohybem parabola začne rozevírat. Všimněme si, že průnik nové paraboly a pobřeží je vrcholem Voroného diagramu. Této situaci říkáme *místní událost* a vidíme ji na obrázku 1.8.

Mimo to se může stát, že nějaká parabola se rozevře natolik, že pohltí jiné a ty zmizí z pobřežní linie. Situaci sledujme na obrázku 1.9. Mějme nějaké tři paraboly jdoucí v pobřeží po sobě. Prostřední z nich je pohlcena v okamžiku, kdy se hrany vykreslované průsečíky parabol setkají v jednom bodě. Tento bod musí být stejně daleko od všech třech ohnisek, takže je středem kružnice určené těmito ohnisky.

Kde je v tomto okamžiku zametací přímka? Musí být v takové poloze, aby střed kružnice právě vykoul zpoza pobřeží. Jinými slovy musí být stejně daleko od



Obr. 1.8: Místní událost – červená kolmice je nově vznikající parabola, při postupu zametací přímky dále se bude rozevírat a vytvoří další parabolu.

středu, jako jsou ohniska, čili se kružnice dotýkat zespodu jako na obrázku 1.10. Této situaci říkáme *kružnicová událost*.

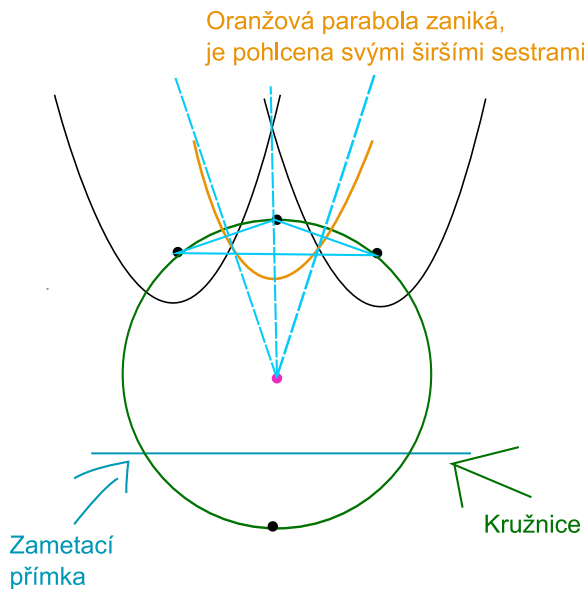
Algoritmus tedy bude **udržovat nějaký kalendář událostí** a vždy skákat zametací přímkou na následující událost. **Místní události můžeme všechny naplánovat dopředu, kružnicové budeme plánovat (a přeplánovávat) průběžně**, kdykoliv se pobřeží změní.

To je podobné algoritmu na průsečíky úseček a stejně tak budou podobné i datové struktury: kalendář si budeme uchovávat v haldě nebo vyhledávacím stromu, **pobřeží ve vyhledávacím stromu s implicitními klíči**: v každém **vrcholu si uložíme dvě paraboly, jejichž průsečíkem má vrchol být**.

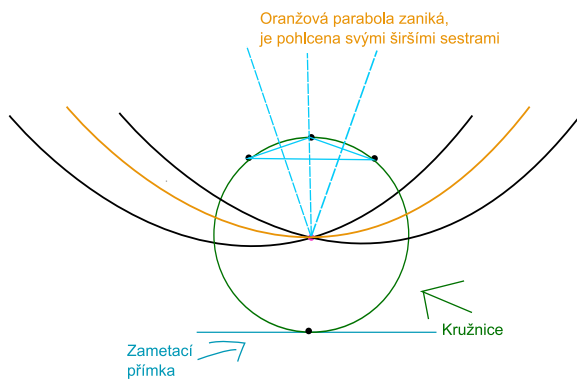
Poslední datovou strukturou bude samotný diagram, reprezentovaný grafem se souřadnicemi a vazbami hran na průsečík.

### Algoritmus FORTUNE

1. Vytvoříme kalendář  $K$  a vložíme do něj všechny místní události.
2. Založíme prázdnou pobřežní linii  $P$ .
3. Dokud kalendář není prázdný:
4.     Odebereme další událost.
5.     Je-li to místní událost:
6.         Najdeme v  $P$  parabolu podle  $x$ -ové souřadnice místa.
7.         Rozdělíme ji a mezi její části vložíme novou parabolu.



Obr. 1.9: Před kružnicovou událostí



Obr. 1.10: Kružnicová událost

8. Do diagramu zaznamenáme vrchol, v němž jedna hrana končí a dvě nové začínají.
9. Je-li to kružnicová událost:
10. Smažeme parabolu z  $P$ .
11. Do diagramu zaznamenáme vrchol, v němž dvě hrany končí a jedna začíná.
12. Po změně pobřeží přepočítáme kružnicové události ( $\mathcal{O}(1)$  jich

zanikne,  $\mathcal{O}(1)$  vznikne).

**Věta:** Fortunův algoritmus pracuje v čase  $\mathcal{O}(n \log n)$  a prostoru  $\mathcal{O}(n)$ .

*Důkaz:* Celkově nastane  $n$  místních událostí (na každé místo narazíme právě jednou) a  $n$  kružnicových (kružnicová událost smaže jednu parabolu a ty vznikají pouze při místních událostech). Z toho plyne, že kalendář i pobřežní linie jsou velké  $\mathcal{O}(n)$ , takže pracují v čase  $\mathcal{O}(\log n)$  na operaci. Jednu událost proto naplánujeme i obsloužíme v čase  $\mathcal{O}(\log n)$ , což celkem dává  $\mathcal{O}(n \log n)$ .  $\square$

## 1.4. Lokalizace bodu

Pokračujeme v problému z minulého oddílu. Máme nějakou množinu míst v rovině a chceme umět pro libovolný bod nalézt nejbližší místo (pokud jich je víc, stačí libovolné jedno). To už umíme převést na nalezení oblasti ve Voroného diagramu daných míst, do které zadaný bod padne.

Chceme tedy pro nějaký rozklad roviny na mnohoúhelníkové oblasti vybudovat datovou strukturu, která pro libovolný bod rychle odpoví, do jaké oblasti patří.

Začneme primitivním řešením bez předzpracování. Rovinu zametáme shora dolů vodorovnou přímkou, podobně jako při hledání průsečíků úseček. Udržujeme si průřez hranic oblastí zametací přímkou. Tento průřez se mění jenom ve vrcholech mnohoúhelníků. Ve chvíli, kdy narazíme na hledaný bod, podíváme se, do kterého intervalu mezi hranicemi v průřezu patří. Tento interval odpovídá jedné oblasti, kterou nahlásíme. Kalendář událostí i průřez opět ukládáme do vyhledávacích stromů, jednu událost obsloužíme v  $\mathcal{O}(\log n)$  a celý algoritmus běží v  $\mathcal{O}(n \log n)$ .

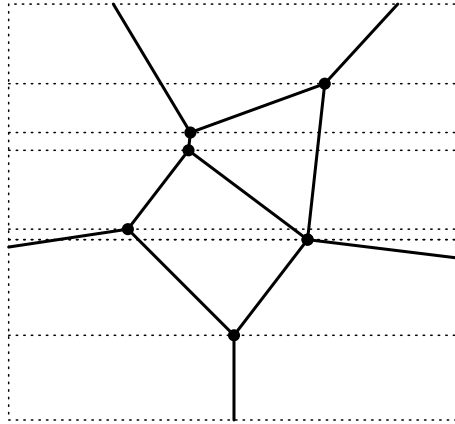
To je obhludně pomalé, dokonce pomalejší než pokaždé projít všechny oblasti a pro každou zjistit, zda v ní zadaný bod leží. Ale i z ošklivé housenky se může vyklubat krásný motýl...

Zavedeme předzpracování. Zametání oblastí necháme běžet „naprázdno“, aniž bychom hledali konkrétní bod. Rovinu rozřežeme polohami zametací přímkou při jednotlivých událostech na pásy. Pro každý pás si zapamatujeme kopii průřezu (ten se uvnitř pásu nemění) a navíc si uložíme  $y$ -ové souřadnice hranic pásů. Nyní na vyhodnocení dotazu stačí najít podle  $y$ -ové souřadnice správný pás (což jistě zvládneme v logaritmickém čase) a poté položit dotaz na zapamatovaný průřez pro tento pás.

Dotaz dokážeme zodpovědět v čase  $\mathcal{O}(\log n)$ , ovšem předvýpočet vyžaduje čas  $\Theta(n^2)$  na zkopírování všech  $n$  stromů a spotřebuje na to stejné množství prostoru.

### Persistentní vyhledávací stromy

Složitost předvýpočtu zachráníme tím, že si pořídíme persistentní vyhledávací strom. Ten si pamatuje historii všech svých změn a umí vyhledávat nejen v aktuálním stavu, ale i ve všech stavech z minulosti. Přesněji řečeno, po každé operaci, která mění stav stromu, vznikne nová verze stromu a operace pro dotazy dostanou jako další parametr identifikátor verze, ve které mají hledat.

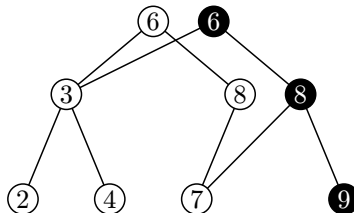


Obr. 1.11: Oblasti rozřezané na pásy

Předvýpočet tedy bude udržovat průřez v persistentním stromu a místo aby ho v každém pásu zkopíroval, jen si zapamatuje identifikátor verze, která k pásu patří.

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 porídíme kopii vrcholu a tu změníme. Musíme ovšem změnit ukazatel na daný vrchol, 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.

Strom nové verze tedy obsahuje novou cestu mezi kořenem a upravovaným vrcholem. Tato cesta se odkazuje na podstromy z minulé verze. Uchování jedné verze nás proto stojí čas  $\mathcal{O}(\log n)$  a prostor také  $\mathcal{O}(\log n)$ . Ještě nesmíme zapomenout, že po každé operaci následuje vyvážení stromu. To ovšem upravuje pouze vrcholy, které leží v konstantní vzdálenosti od cesty mezi místem úpravy a kořenem, takže jejich zkopírováním časovou ani prostorovou složitost nezhoršíme.



Obr. 1.12: Vložení prvku 9 do persistentního stromu

Celkově tedy spotřebujeme čas  $\mathcal{O}(n \log n)$  na předzpracování Voroného diagra-

mu a vytvoření persistentního stromu. Kvůli persistenci potřebuje toto předzpracování paměť  $\mathcal{O}(n \log n)$ . Dotazy vyřizujeme v čase  $\mathcal{O}(\log n)$ , neboť nejprve vyhledáme příslušný pás a poté položíme dotaz na příslušnou verzi stromu.

**Poznámka:** Persistence datových struktur je přirozená pro striktní funkcionální programovací jazyky (například Haskell). V nich neexistují vedlejší efekty příkazů, takže jednou sestavená data již nelze modifikovat, pouze vyrobit novou verzi datové struktury s provedenou změnou.

## Persistence v konstantním prostoru na verzi\*

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 tudíž 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.

odkaz

Nyní ukážeme, jak jednu strukturální změnu zaznamenat v amortizovaně konstantním prostoru. Každý vrchol stromu si tentokrát bude pamatovat až 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 spolu s časem vzniku verze.

**Věta:** Uchování jedné strukturální změny stojí amortizovaně konstantní čas i prostor.

*Důkaz:* Každé vytvoření verze vrcholu stojí konstantní čas a prostor. Jedna operace může v nejhorším případě způsobit vznik nových verzí všech vrcholů až do kořene, ale jednoduchým potenciálovým argumentem lze dokázat, že počet verzí bude amortizovaně konstantní.

Potenciál struktury definujeme jako počet verzí uchovaných ve všech vrcholech dosažitelných z aktuálního kořene. V klidovém stavu struktury jsou ve vrcholu nejvýš dvě verze, během aktualizace dočasně připustíme tři verze.

Strukturální změna způsobí zaznamenání nové verze jednoho vrcholu, což potenciál zvýší o 1, ale možná tím vznikne „tříverzový“ vrchol. Zbytek algoritmu se tříverzových vrcholů snaží zbavit: pokaždé vezme vrchol se 3 verzemi, vytvoří jeho kopii s 1 verzí a upraví ukazatel v otci, čímž přibude nová verze otce. Originálnímu vrcholu zůstanou 2 verze, ale přestane být dosažitelný, takže se už do potenciálu nepočítá.

Potenciál tím klesl o 3 (za odpojený originál), zvýšil se o 1 (za nově vytvořenou kopii s jednou verzí) a poté ještě o 1 (za novou verzi otce). Celkově tedy klesl

o 1. Proto veškeré kopírování vrcholů zaplatíme z konstantního příspěvku od každé strukturální změny.  $\square$

**Důsledek:** Existuje persistentní strom s časem amortizovaně  $\mathcal{O}(\log n)$  na operaci a prostorem amortizovaně  $\mathcal{O}(1)$  na uložení jedné verze. Pomocí něj lze v čase  $\mathcal{O}(n \log n)$  vybudovat datovou strukturu pro lokalizaci bodu o velikosti  $\mathcal{O}(n)$ , která odpovídá na dotazy v čase  $\mathcal{O}(\log n)$ .

## Cvičení

1. Je dána množina obdélníků, jejichž strany jsou rovnoběžné s osami souřadnic. Vybudujte datovou strukturu, která bude umět rychle odpovídat na dotazy typu „v kolika obdélnících leží zadaný bod?“.

## 1.5.\* Rychlejší algoritmus na konvexní obal

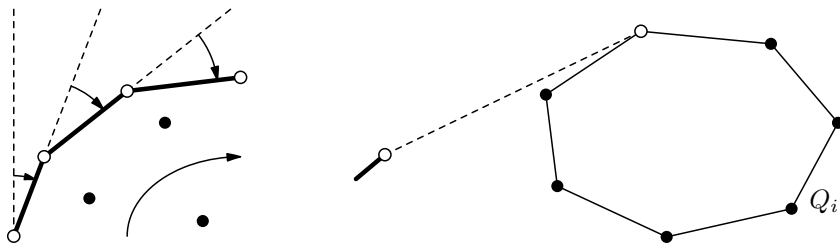
Konvexním obalem naše putování po geometrických algoritmech začalo a také jím skončí. Našli jsme algoritmus pro výpočet konvexního obalu  $n$  bodů v čase  $\Theta(n \log n)$ . Ve cvičení 1.1.5 jsme dokonce dokázali, že tato časová složitost je optimální. Přesto si předvedeme ještě rychlejší algoritmus objevený v roce 1996 Timothy Chanem. S naším důkazem je nicméně všechno v pořádku: rychlejší bude jen v některých případech: časová složitost Chanova algoritmu dosahuje  $\mathcal{O}(n \log h)$ , kde  $h$  značí počet bodů ležících na konvexním obalu.

Předpokládejme, že bychom znali velikost konvexního obalu  $h$ . Body rozdělíme libovolně do  $\lceil n/h \rceil$  množin  $Q_1, \dots, Q_k$  tak, aby v každé množině bylo nejvýše  $h$  bodů. Pro každou z těchto množin nalezneme konvexní obal pomocí obvyklého algoritmu. To dokážeme pro jednu množinu v čase  $\mathcal{O}(h \log h)$  a pro všechny v  $\mathcal{O}(n \log h)$ . Poté tyto předpočítané obaly slepíme do jednoho pomocí takzvaného *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 svou činností připomíná namotávání provázku podél konvexního obalu. Začneme bodem, který na konvexním obalu určitě leží – třeba tím nejlevějším. V každém dalším kroku nalezneme následující bod po obvodu konvexního obalu. Například tak, že projdeme všechny body a vybereme ten, který svírá nejmenší úhel s předchozí stranou konvexního obalu. Nově přidaná úsečka vyhovuje pozorování, a tudíž do konvexního obalu patří. Po  $h$  krocích se dostaneme zpět k nejlevějšimu 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  $\mathcal{O}(nh)$ .

Provázkový algoritmus funguje, ale je 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  $Q_i$ .



Obr. 1.13: Provázkový algoritmus a jeho použití v předpočítaném obalu

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 zatažení konvexního obalu. Detaily ponechme jako cvičení.

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

Popsanému algoritmu schází jedna důležitá věc: Ve skutečnosti málokdy známe 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  $\sum 2^m$ .

## Cvičení

1. Domyslete detaily hledání kandidáta „kruhovým“ půlením intervalu.