Masarykova univerzita
Fakulta informatiky

Překryvy map

Diplomová práce

Dominik Janků

Brno, jaro 2014

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: doc. RNDr. Martin Čadek, CSc.

Poděkování

Mé velké poděkování patří doc. RNDr. Martinu Čadkovi, CSc., za ochotu, trpělivost a cenné poznámky během vedení této diplomové práce.

Chtěl bych také poděkovat své přítelkyni za trpělivost, oporu a věcné poznámky k jazykové části textu.

Shrnutí

Diplomová práce představuje dva rovinné geometrické algoritmy, jmenovitě Bentlyův-Ottmanův algoritmus pro výpočet průsečíků úseček a algoritmus pro výpočet překryvu map. Kromě teoretického výkladu nabízí kompletní pseudokódy, které mohou být využity při jejich implementaci. Ta je ostatně součástí práce v podobě ukázkových programů s otevřeným zdrojovým kódem.

Klíčová slova

průsečíky úseček, algoritmus zametací přímky, Bentley-Ottman, překryvy map, rovinná podrozdělení, pseudokód, Python

Obsah

Úvod

Geometrické algoritmy se vždy řadily k nejpřitažlivějším částem informatiky. Snad proto, že se s jejich aplikací setkáváme takřka na každém kroku. Není proto příliš velkým překvapením, že jsem si jednu z jejich částí vybral jako téma této diplomové práce.

Problematika výpočtu průsečíků a překryvu map patří k těm nejzákladnějším. Setkáváme se s nimi například v grafických, geografických, nebo konstruktérských aplikacích, které by bez efektivních algoritmů nemohly pracovat. Přestože v dnešní době disponujeme řádově většími výpočetními výkony, než tomu bylo v minulosti, pořád zde existuje poptávka po algoritmech, které řeší daný problém v co nejmenší časové složitosti.

Algoritmy prezentované v této práci patří mezi ně. První z nich, *Bentleyův-Ottmanův* algoritmus [2] slouží k výpočtu průsečíků úseček, přičemž zohledňuje jejich počet ve své časové složitosti. Totéž činí i algoritmus překryvu map [3], který je popsán následovně.

Konečným cílem této práce je naprogramování aplikací, které by oba výše uvedené algoritmy implementovaly a umožnily uživateli jejich názornou ukázku.

Způsob, jakým algoritmy pracují, je popsán zvlášť v jednotlivých kapitolách. Teoretický výklad je navíc doplněn o pseudokódy, které čtenáři mohou pomoci s pochopením a případnou implementací algoritmů. Členění kapitol je následující.

První kapitola otevírá problematiku průsečíků úseček a to rekapitulací základních poznatků z oboru analytické geometrie. Ihned poté následuje představení Bentley-Ottmanova algoritmu, včetně příslušných pseudokódů.

Druhá kapitola obsahuje stěžejní část této práce – průniky map. Úvodem čtenáře seznamuje se základními pojmy a pokračuje představením dvojitě souvislého seznamu, jakožto vhodné datové struktury pro uchování map. Jednotlivé kroky algoritmu jsou podrobně

popsány a stejně jako v kapitole první, ani zde nechybí doprovodné pseudokódy.

Třetí kapitola představuje naprogramované aplikace pro demonstraci algoritmů. Uvedeny jsou použité technologie a způsob ovládání aplikací, které zároveň slouží jako uživatelská příručka.

Poslední kapitola přináší rekapitulaci dosažených výsledků.

Kapitola 1

Průsečíky úseček

Pro řešení úlohy překryvu map potřebujeme zvládnout výpočet průsečíků úseček. V této kapitole budeme pracovat s množinou úseček $\{s_1, s_2, \ldots, s_n\}$, kde jednotlivé úsečky s_i jsou určeny koncovými body v $\mathbb{R}^2$.

1.1 Výpočet průsečíku dvou úseček

Máme dány dvě úsečky: úsečku s, určenou body (p_x, p_y), (q_x, q_y) a úsečku s' určenou body (p'_x, p'_y), (q'_x, q'_y). Naším úkolem je nalézt jejich průsečík.

Pro tento účel využijeme parametrické vyjádření přímky s v rovině:

$$\begin{aligned} x &= p_x + tu_1, \\ y &= p_y + tu_2, \end{aligned} \qquad \vec{u} = (q_x - p_x, q_y - p_y), \tag{1.1}$$

kde t je tzv. parametr. Pokud platí, že $t \in [0; 1]$, pak bod (x, y) leží na úsečce s.

K nalezení průsečíku úseček s, s' odvodíme soustavu rovnic, kterou získáme z parametrického vyjádření (1.1):

$$\begin{aligned} p_x + t_1 u_1 &= p'_x + t_2 v_1, & \vec{u} &= (q_x - p_x, q_y - p_y), \\ p_y + t_1 u_2 &= p'_y + t_2 v_2, & \vec{v} &= (q'_x - p'_x, q'_y - p'_y). \end{aligned} \tag{1.2}$$

Ze soustavy rovnic (1.2) vypočteme parametr t_1 a t_2 následovně:

$$t_1 = \frac{v_1(p_y - p'_y) + v_2(p'_x - p_x)}{u_1 v_2 - u_2 v_1}, \qquad t_2 = \frac{u_1(p_y - p'_y) + u_2(p'_x - p_x)}{u_1 v_2 + u_2 v_1}.$$

Pokud nám vyjde nulový jmenovatel, znamená to, že jsou úsečky navzájem rovnoběžné (mají stejný směrový vektor) a průsečík buď neexistuje, nebo jich existuje nekonečně mnoho.

Je-li jmenovatel různý od nuly, nejsou úsečky rovnoběžné a průsečík existuje, právě když $t_1, t_2 \in [0, 1]$. V tomto případě má průsečík souřadnice

$$(p_x + t_1 u_1, p_y + t_1 u_2).$$

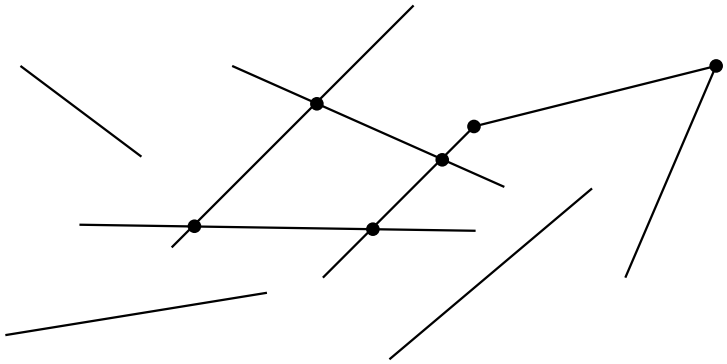
Obecně si však nevystačíme s počítáním průsečíku pouze dvou úseček. Ve většině případů potřebujeme spočítat průsečíky n úseček, kde n může být libovolně velké (konečné) číslo. Z tohoto důvodu vznikly nejrůznější algoritmy, které tuto úlohu řeší [2, 4].

1.2 Triviální algoritmus

Průsečíky hledáme tak, že vybereme libovolnou úsečku s a testujeme ji na průsečík s ostatními úsečkami. Otestované dvojice úseček si označíme a pokračujeme dále. Celkem tedy potřebujeme $\binom{n}{2}$ testů, což odpovídá časové složitosti $O(n^2)$.

Zde je třeba říct, že lepší algoritmus obecně neexistuje, neboť vždy může existovat až $\binom{n}{2}$ průsečíků a ty musíme označit. V praxi se ale nestává příliš často, že bychom jich měli tak mnoho.

Z tohoto důvodu se hledala řešení, která by počet průsečíků zohlednila při výpočtu. Jedním z nich je Bentleyův–Ottmanův algoritmus, o kterém pojednává následující část textu.

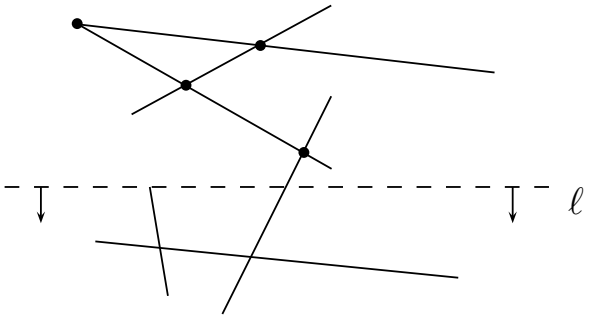


Obrázek 1.1: Průsečíky úseček

1.3 Bentleyův–Ottmanův algoritmus

Bentleyův–Ottmanův algoritmus patří do rodiny algoritmů pracujících s ideou tzv. zametací přímky [2] – myšlené horizontální přímky, která se zastavuje v tzv. událostech. Tyto události jsou procházeny odshora dolů, zleva doprava a vše co je nad nimi (a vlevo na přímce od události) je pokládáno za vyřešené.

Algoritmus dosahuje časové složitosti $O((n + k) \log n)$, kde k je počet průsečíků a n je počet úseček (viz [3, 9] pro důkaz).



Obrázek 1.2: Průchod zametací přímky ℓ

1.4 Lexikografické řazení bodů

Definujeme následující lexikografické uspořádání bodů v rovině:

$$p < q \iff p_y < q_y \vee (p_y = q_y \wedge p_x < q_x). \tag{1.3}$$

Jak už je v počítačové grafice zvykem, y-souřadnice roste na zobrazovacím zařízení odshora dolů a my se této konvence budeme držet.

Pokaždé, když budeme hovořit o horním nebo dolním bodu úsečky, máme tím na mysli lexikograficky menší nebo větší bod.

1.5 Použité datové struktury

Algoritmus využívá ke své činnosti dvě datové struktury: prioritní frontu událostí Q a vyvážený binární strom $\mathcal{T}$ [3]. Obě si rozebereme podrobněji.

1.5.1 Prioritní fronta Q

Do prioritní fronty [8] ukládáme doposud nezpracované události, což jsou význačné body, ve kterých se algoritmus zastavuje. Události jsou ve frontě řazeny lexikograficky (proto prioritní) dle uspořádání 1.3. Na začátku fronty se proto nachází nejmenší událost.

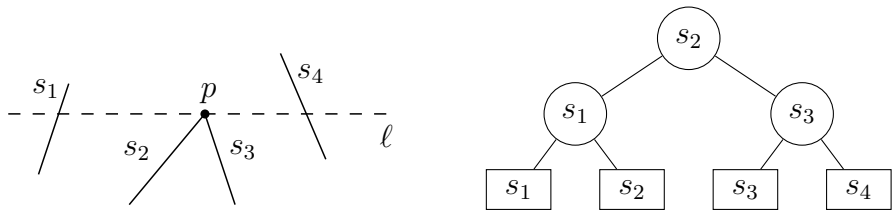
Vložení nové události do fronty musí splňovat časovou složitost $O(\log n)$. Ve stejné složitosti probíhá také ověření, zda-li se již daná událost ve frontě nenachází. Logaritmické složitosti můžeme dosáhnout využitím binárního vyhledávání (půlení intervalu).

Výběr nejmenší události ke zpracování musí probíhat v čase $O(1)$.

1.5.2 Vyvážený binární strom $\mathcal{T}$

Vyvážený binární strom [8] $\mathcal{T}$ uchovává ve svých listech aktuální pořadí úseček, které protíná zametací přímka ℓ. Algoritmus tuto informaci využívá při testování průsečíků, což bude podrobněji rozebráno v sekci 1.9.

Ve vnitřních uzlech stromu se pak nachází hodnota nejpravějšího listu levého podstromu. Ukázku stromu můžeme pozorovat na obrázku 1.3.



Obrázek 1.3: Úsečky protnuté ℓ a odpovídající strom $\mathcal{T}$

Může být trochu nezvyklé uchovávat ve stromě úsečky namísto čísel. Pokud ale dokážeme určit, která z úseček je „menší“ (myšleno více vlevo) a která „větší“ (myšleno více vpravo) je možné použít i takové uspořádání.

Abychom toho byli schopni, předpokládejme, že bod p je právě zpracovávaná událost. Zametací přímka ℓ splňuje rovnici $y = p_y$ (prochází bodem p). Dále předpokládejme existenci libovolné úsečky

s, která je určena body (x_1, y_1) a (x_2, y_2), přičemž platí, že

$$(x_1, y_1) \leq p < (x_2, y_2)$$

dle lexikografického uspořádání 1.3. Definujeme funkci

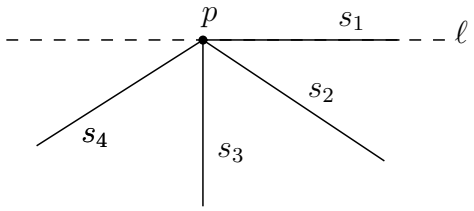
$$\xi(s) = \begin{cases} x_1 + \dfrac{(p_y - y_1)(x_2 - x_1)}{y_2 - y_1} & \text{pro } y_1 \neq y_2, \\ p_x & \text{jinak,} \end{cases} \tag{1.4}$$

která vrací x-souřadnici průsečíku úsečky s se zametací přímkou ℓ. Pokud je úsečka horizontálně orientovaná, použije se x-souřadnice události p.

Dále definujeme funkci

$$\varphi(s) = \frac{x_2 - x_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}, \tag{1.5}$$

která je rovna kosinu úhlu úsečky s svíraného s osou x. Ten budeme potřebovat v situacích, kdy prochází bodem p více úseček zároveň (viz obrázek č. 1.4). Obdržíme totiž více stejných hodnot $\xi(s_i)$, což ke korektnímu rozhodnutí o pořadí nebude stačit. Porovnání úhlů nám tuto možnost nabízí.

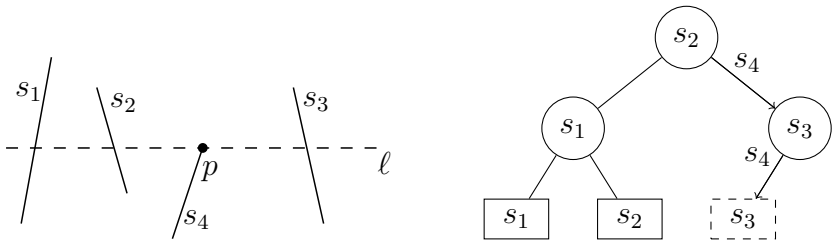


Obrázek 1.4: Více úseček prochází událostí p

Nyní již známe vše potřebné k tomu, abychom mohli definovat ostré uspořádání nad libovolnými úsečkami s, t ve stromě $\mathcal{T}$:

$$s < t \iff \xi(s) < \xi(t) \vee (\xi(s) = \xi(t) \wedge \varphi(s) < \varphi(t)). \tag{1.6}$$

Nemožnost rozhodnout, která úsečka je více nalevo, nebo více napravo znamená chybu na vstupu (úsečky se překrývají). Pro jednoduchost tedy předpokládejme, že se úsečky na vstupu překrývat nebudou.

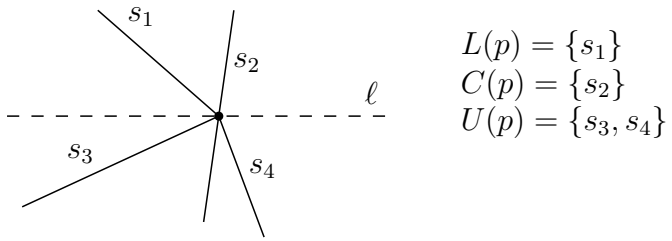


Obrázek 1.5: Přidání úsečky s_4 do stromu $\mathcal{T}$ dle $\xi(s_i)$

1.6 Množiny $U(p)$, $C(p)$ a $L(p)$

V následujícím textu se setkáme s třemi typy množin. Každá z nich je podmnožinou úseček S, které algoritmus získá na svém vstupu. Jejich význam je následující:

- $U(p)$ – množina úseček, jejichž horním bodem je p,
- $C(p)$ – množina úseček, jejichž vnitřním bodem je p,
- $L(p)$ – množina úseček, jejichž dolním bodem je p.



Obrázek 1.6: Rozdělení úseček do množin $L(p)$, $C(p)$ a $U(p)$

Množiny $U(p)$ a $L(p)$ konstruujeme právě jednou při inicializaci algoritmu (viz sekce 1.7). Množiny $C(p)$ konstruujeme v jeho průběhu a to při detekci průsečíků (viz sekce 1.9).

1.7 Inicializace a běh algoritmu

Algoritmus 1 (doplněný z [3]) nejprve inicializuje prázdnou frontu událostí Q a vyvážený binární strom $\mathcal{T}$ (ř. 1–2).

Algoritmus 1: SWEEPLINE (S)

Input: Množina úseček S.
Output: Průsečíky úseček.

```
1:  Inicializujeme prázdnou prioritní frontu událostí Q.
2:  Inicializujeme prázdný vyvážený binární strom T.
3:  for s_i ∈ S do
4:      p_u ← lexikograficky horní bod úsečky s_i
5:      p_l ← lexikograficky dolní bod úsečky s_i
6:      Přidáme úsečku s_i do množin U(p_u) a L(p_l).
7:      Přidáme body p_u a p_l do fronty Q.
8:  end
9:  while |Q| > 0 do
10:     Vybereme z fronty Q událost p.
11:     HANDLEEVENTPOINT(p)
12: end
```

Poté následuje průchod jednotlivých úseček ze vstupní množiny (ř. 3–8), při kterém provádíme následující úkony:

1. konstruujeme množiny úseček $U(p)$, $L(p)$ pro pozdější použití,

2. přidáváme koncové body úseček do fronty událostí Q.

Posledním krokem je výběr nezpracované události ze začátku fronty Q (ř. 9–12) a její zpracování algoritmem č. 2. Tak činíme do doby, než bude fronta událostí prázdná.

1.8 Zpracování události

Stěžejní částí Bentley-Ottmanova algoritmu je zpracování události p algoritmem č. 2 (vychází z [3]). Na jeho počátku zpřístupníme množiny $U(p)$, $C(p)$ a $L(p)$. V případě, že sjednocení množin obsahuje více než jednu úsečku, označíme bod p jako jejich průsečík (ř. 4–6).

Algoritmus 2: HANDLEEVENTPOINT(p)

Input: Událost p.

1: $U(p) \leftarrow$ úsečky jejichž horní bod je p
2: $L(p) \leftarrow$ úsečky jejichž dolní bod je p
3: $C(p) \leftarrow$ úsečky jejichž vnitřním bodem je p
4: **if** $| U(p) \cup C(p) \cup L(p) | \geq 2$ **then**
5: | Označíme bod p jako průsečík úseček $U(p) \cup C(p) \cup L(p)$.
6: **end**
7: Odebereme úsečky $L(p) \cup C(p)$ ze stromu $\mathcal{T}$.
8: Vložíme úsečky $U(p) \cup C(p)$ do stromu $\mathcal{T}$.
9: **if** $U(p) \cup C(p) = \emptyset$ **then**
10: | $s_l \leftarrow$ úsečka nalevo od bodu p ve stromě $\mathcal{T}$
11: | $s_r \leftarrow$ úsečka napravo od bodu p ve stromě $\mathcal{T}$
12: | FINDNEWEVENT(s_l, s_r, p)
13: **else**
14: | $s' \leftarrow$ nejlevější úsečka z $U(p) \cup C(p)$
15: | $s_l \leftarrow$ levý soused úsečky s' ve stromě $\mathcal{T}$
16: | FINDNEWEVENT(s', s_l, p)
17: | $s'' \leftarrow$ nejpravější úsečka z $U(p) \cup C(p)$
18: | $s_r \leftarrow$ pravý soused úsečky s'' ve stromě $\mathcal{T}$
19: | FINDNEWEVENT(s'', s_r, p)
20: **end**

Práce se stromem $\mathcal{T}$

Je velice důležité, abychom při manipulaci se stromem $\mathcal{T}$ (ř. 7–8) dodržovali následující pravidla:

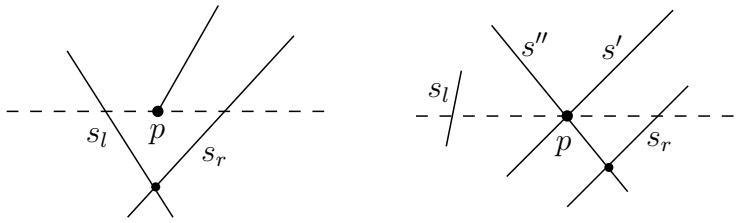
1. při odebírání úseček se řídíme uspořádáním 1.6, které je platné pro předchozí událost,
2. naopak při vkládání je platné uspořádání podle zpracovávané události p.

Činíme tak proto, že úsečky $C(p)$ mění v bodě p své pořadí. Strom ale o této změně nemůže „vědět“ a tak v něm zůstává pořadí úseček platné z předchozí události. Abychom zajistili jejich správné prohození, odstraníme je a znovu vložíme (podle zmíněných pravidel).

Tímto způsobem udržujeme platné pořadí úseček, které je nezbytné pro nalezení nových průsečíků.

1.9 Hledání nových průsečíků

Posledním krokem algoritmu je hledání nových průsečíků. To probíhá v závislosti na tom, jestli je sjednocení $U(p) \cup C(p)$ prázdné či nikoliv (alg. č. 2, ř. 9–20) [3]. Na obrázku 1.7 vidíme ilustraci obou situací. Je zřejmé, že pokud některá z úseček s_l, s_r, s', s'' neexistuje, nemá smysl hledat její průsečíky.



(a) $U(p) \cup C(p) = \emptyset$ (b) $U(p) \cup C(p) \neq \emptyset$

Obrázek 1.7: Hledání nových průsečíků

Algoritmus 3: FINDNEWEVENT(s_l, s_r, p)

Input: s_l, s_r - testované úsečky, p - bod události

```
1: if ∃ průsečík q úseček s_l a s_r ∧ q > p then
2:     if q je vnitřním bodem s_l then
3:         Přidáme úsečku s_l do množiny C(q).
4:     if q je vnitřním bodem s_r then
5:         Přidáme úsečku s_r do množiny C(q).
6:     if q ∉ Q then
7:         Přidáme bod q do fronty Q.
8: end
```

Algoritmus č. 3 řeší testování na průsečík dvou úseček podle soustavy rovnic 1.2. Pokud je průsečík nalezen a leží pod bodem událostí p, pak se přistoupí k dalšímu zpracováni. Průsečíky nad událostí nás

již nezajímají (byly vypočítány a zpracovány v předchozím průběhu algoritmu).

1.10 Výstup

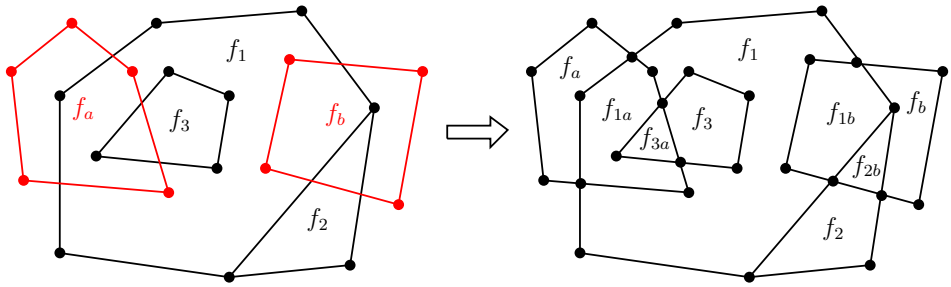
Výstupem algoritmu je seznam všech průsečíků včetně úseček, které jimi prochází.

Jak uvidíme v následující kapitole, může být výstup obohacen o další informace, které potřebujeme pro vyřešení specifického typu úlohy.

Kapitola 2

Překryvy map

Stěžejní částí této práce jsou překryvy map. Problém překryvu si můžeme jednoduše představit jako nalezení průsečíků a překrývajících se oblastí dvou zadaných map [3]. Příklad jednoduchého překryvu můžeme vidět na obrázku 2.1, oblasti jsou značeny písmenem f.



Obrázek 2.1: Překryv dvou jednoduchých map

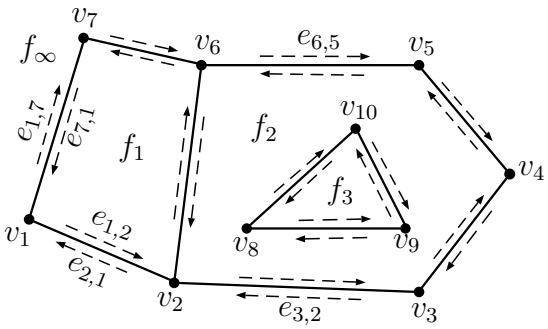
Překryvy map se uplatňují v široké škále aplikací. Nejčastěji pak v geografických výpočtech, územním projektování atp.

Když se například rozhodne o stavbě nové pozemní komunikace, je potřeba učinit řadu nezbytných kroků. Jedním z nich je zjistit, přes které pozemky projektovaná stavba povede. Na základě těchto informací pak dojde k rozdělení pozemků v katastru nemovitostí a jsou zahájeny jejich výkupy. Jedná se o ukázkový příklad, u kterého je vhodné použít překryvy map – jedna mapa bude projektované území silnice a druhá mapa bude oblast, do které je silnice zasazena.

2.1 Základní pojmy

V této části si osvětlíme některé základní pojmy, které budou nezbytné pro správné pochopení algoritmu.

Začněme ústředním pojmem celé kapitoly – *mapou*. Mapa není nic jiného, než rovinné podrozdělení. Tedy rovina, kterou rozdělují uzavřené lomené čáry složené z úseček na oblasti.



Obrázek 2.2: Příklad rovinného podrozdělení

Na obrázku 2.2 můžeme vidět jednoduché rovinné podrozdělení, skládající se ze 4 oblastí, 10 vrcholů a 22 hran (hranu chápeme jako orientovanou úsečku). Rovinné podrozdělení značíme písmenem S.

V textu se dále setkáme s následujícími pojmy:

Vrchol je koncový bod hrany, budeme jej označovat jako v_i (vertex).

Hrana je orientovaná úsečka vycházející z vrcholu v_i a končící ve vrcholu v_j. Značíme jako $e_{i,j}$ (edge). Výchozí vrchol hrany v_i označujeme jako $origin(e_{i,j})$.

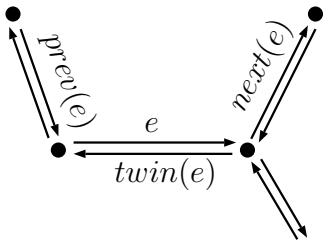
Oblast je plocha vymezená právě jednou vnější hranicí a libovolným počtem hranic vnitřních. Označujeme f_i (face).

Přilehlá oblast je oblast umístěna nalevo od hrany.

Dvojče hrany $e_{j,i}$ je hrana s opačnou orientací od hrany $e_{i,j}$. Dvojče hrany označujeme $twin(e_{i,j})$.

Následník hrany $e_{i,j}$ je hrana $e_{j,k}$ vycházející z koncového vrcholu hrany $e_{i,j}$, která má stejnou přilehlou oblast jako $e_{i,j}$. Označujeme $next(e_{i,j})$.

Předchůdce hrany $e_{j,k}$ je hrana $e_{i,j}$, která má svůj koncový vrchol totožný s počátečním vrcholem hrany $e_{j,k}$ a navíc mají obě hrany stejnou přilehlou oblast. Označujeme $prev(e_{j,k})$.



2.2 Dvojitě souvislý seznam

Vhodnou datovou strukturou pro reprezentaci mapy je *dvojitě souvislý seznam*[1]. Tvoří jej tři tabulky záznamů (relace), z nichž každá zvlášť uchovává informace o vrcholech, hranách a oblastech [3].

K základním atributům jednotlivých tabulek (viz podsekce 2.2.1) můžeme libovolně přidávat další – doplňující informace. Když například sestavujeme mapu znečištění ovzduší, bude u každé oblasti uvedena také hodnota naměřené veličiny, atp.

Dvojitě souvislým je seznam nazýván proto, jelikož propojuje vrcholy s hranami a hrany s oblastmi.

V dalším textu budeme dvojitě souvislý seznam označovat písmenem D.

1. V anglické literatuře se setkáme s názvem doubly-connected edge list (DCEL).

2.2.1 Struktura dvojitě souvislého seznamu

1. vrcholy (vertices)
   - název vrcholu[2]
   - souřadnice ($\mathbb{R} \times \mathbb{R}$)
   - ukazatel[3] na libovolnou hranu vycházející z vrcholu
2. hrany (edges)
   - název hrany
   - ukazatel na vrchol, ze kterého hrana vychází
   - ukazatel na dvojče hrany
   - ukazatel na následníka hrany se stejnou přilehlou oblastí
   - ukazatel na předchůdce hrany
   - ukazatel na přilehlou oblast (oblast vlevo dle směru orientace)
3. oblasti (faces)
   - název oblasti
   - ukazatel na jednu hranu přilehlou z vnější hranice
   - ukazatel na jednu hranu přilehlou pro každou komponentu vnitřní hranice

Názvy vrcholů, hran a oblastí mohou být libovolné[4] – jsou určeny čistě pro naše potřeby a nemají vliv na strukturu dat. Tím rozhodujícím jsou totiž ukazatelé.

Ukazatel si můžeme představit jako adresu (nebo index), na které se poukazovaný záznam nachází. Nejedná se tedy o data samotná, ale jen o informaci, kde je nalezneme. Přístup k poukazovanému záznamu probíhá v konstantním čase $O(1)$.

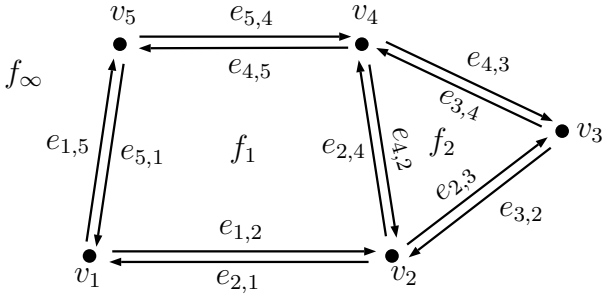
2. Pro naše účely budeme vrcholy pojmenovávat podle jejich označení, tedy v_i.
3. Ukazatel chápeme jako adresu konkrétní poukazované položky v seznamu, která je dostupná v čase $O(1)$. V programovacích jazycích jako je C, C++ se často implementuje pomocí operátoru *.
4. Názvy se dokonce mohou i shodovat, nejedná se tedy o tzv. primární klíče

Pokud budeme chtít získat ostatní informace (jako například všechny hrany vedoucí z/do vrcholu), můžeme tak zpravidla učinit v lineárním čase $O(n)$ ku celkovému množství požadovaných dat.

2.2.2 Příklad zápisu dat

Na obrázku 2.3 máme uvedeno jednoduché rovinné podrozdělení. Pro názornost si jej převedeme do dvojitě souvislého seznamu. Pokud je v názvu sloupce tabulky uveden znak * znamená to, že se jedná o ukazatel na data, nikoliv data samotná.

Výsledek převodu vidíme v tabulkách 2.1, 2.2 a 2.3.



Obrázek 2.3: Triviální rovinné podrozdělení

Název	Souřadnice	* Vycházející hrana
v_1	$(0; 10)$	$e_{1,2}$
v_2	$(10; 10)$	$e_{2,3}$
v_3	$(15; 5)$	$e_{3,4}$
v_4	$(9; 0)$	$e_{3,4}$
v_5	$(1; 0)$	$e_{4,1}$

Tabulka 2.1: Vrcholy z obrázku 2.3

2.3 Algoritmus

Nyní již známe všechny potřebné informace k tomu, abychom si mohli rozebrat princip algoritmu překryvu map. Ten bude na svém

Název	*Vrchol	*Dvojče	*Následník	*Předchůdce	*Oblast
$e_{1,2}$	v_1	$e_{2,1}$	$e_{2,4}$	$e_{5,1}$	f_1
$e_{2,1}$	v_2	$e_{1,2}$	$e_{1,5}$	$e_{3,2}$	f_∞
$e_{2,3}$	v_2	$e_{3,2}$	$e_{3,4}$	$e_{4,2}$	f_2
$e_{3,2}$	v_3	$e_{2,3}$	$e_{2,1}$	$e_{4,3}$	f_∞
$e_{3,4}$	v_3	$e_{4,3}$	$e_{4,2}$	$e_{2,3}$	f_2
$e_{4,3}$	v_4	$e_{3,4}$	$e_{3,2}$	$e_{5,4}$	f_∞
$e_{4,2}$	v_4	$e_{2,4}$	$e_{2,3}$	$e_{3,4}$	f_2
$e_{2,4}$	v_2	$e_{4,2}$	$e_{4,5}$	$e_{1,2}$	f_1
$e_{4,5}$	v_4	$e_{5,4}$	$e_{5,1}$	$e_{2,4}$	f_1
$e_{5,4}$	v_5	$e_{4,5}$	$e_{4,3}$	$e_{1,5}$	f_∞
$e_{5,1}$	v_5	$e_{1,5}$	$e_{1,2}$	$e_{4,5}$	f_1
$e_{1,5}$	v_1	$e_{5,1}$	$e_{5,4}$	$e_{2,1}$	f_∞

Tabulka 2.2: Hrany z obrázku 2.3

Název	*Hrana vnější hranice	*Hrany vnitřních hranic
f_∞	–	$\{e_{1,5}\}$
f_1	$e_{1,2}$	$\emptyset$
f_2	$e_{2,3}$	$\emptyset$

Tabulka 2.3: Oblasti z obrázku 2.3

vstupu přijímat dvojitě souvislé seznamy D_1, D_2, reprezentující rovinná podrozdělení S_1 a S_2. Jejich překryv budeme chtít vypočítat.

Výsledkem výpočtu je nové rovinné podrozdělení S, reprezentované dvojitě souvislým seznamem D.

2.3.1 Spojení seznamů

Prvním krokem algoritmu je spojení seznamů D_1 a D_2 do jednoho seznamu D. Tuto operaci provádí algoritmus 4, DCELMERGE(D_1, D_2).

Princip spočívá v postupném slučování hran a vrcholů do jediného seznamu D. Oblasti slučovat nebudeme, neboť je ještě nemáme vypočteny.

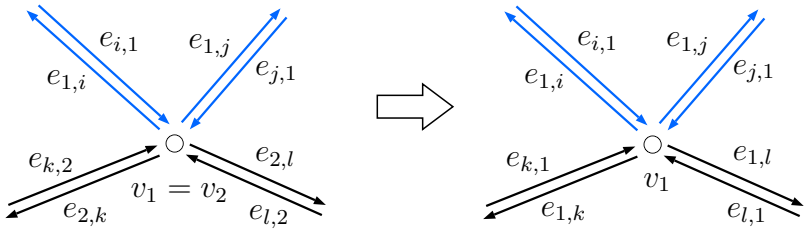
Označením všech vrcholů jako doposud nepřidaných (ř. 2) se vy-

hneme situaci, kdy bychom přidávali do $vertices(D)$ víc shodných vrcholů najednou.

Poté procházíme jednotlivé hrany prvního podrozdělení a postupně je přidáváme do $edges(D)$. Na tomto místě je potřeba zmínit, že vzájemná propojenost ukazatelů v záznamu *musí být zachována*. Toho docílíme například tak, že nebudeme přidávat jednotlivé záznamy hran a vrcholů, ale jen ukazatele na ně. Nové vrcholy a hrany budeme přidávat až ve chvíli, kdy budeme počítat nové průsečíky (viz další části).

Spolu s hranami přidáváme i jejich výchozí vrcholy za podmínky, že se již v D nenachází (ř. 7–10, 18–20). Navíc si u hran poznamenáme číslo mapy, ze které pochází (ř. 4, 13). Tuto informaci použijeme v dalších krocích algoritmu.

Obdobně postupujeme se záznamy původem z D_2. Jediný rozdíl bude spočívat v tom, že ošetříme sitauce, kdy se již vrchol se shodnými souřadnicemi ve $vertices(D)$ nachází. V takovém případě nastavíme u zpracovávané hrany ukazatel $origin(e)$ na existující vrchol ve $vertices(D)$ (ř 16–17).



Obrázek 2.4: Řešení shodných souřadnic u dvou vrcholů

Zmíněnou situaci ilustruje obrázek 2.4. Vrchol v_1 z $vertices(D_1)$ je souřadnicově shodný s vrcholem v_2 z $vertices(D_2)$. Situaci rozhodneme ve prospěch vrcholu z prvního podrozdělení.

Abychom při kontrole existence vrcholu, prováděné na řádku 16, zachovali časovou složitost $O(n \log n)$, můžeme si vytvořit pomocný binární vyhledávací strom. Uzly stromu uspořádáme dle souřadnic vrcholů. Vytvoření stromu má časovou složitost $O(n \log n)$, vyhledání vrcholu v něm pak $O(\log n)$. Tímto způsobem můžeme zjistit, zda-li se již nějaký vrchol ve $vertices(D)$ nenachází.

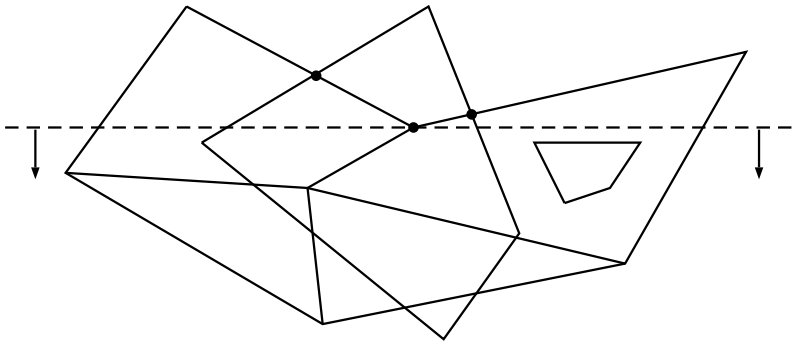
Algoritmus 4: DCELMERGE (D_1, D_2)

Input: Dvojitě souvislé seznamy D_1, D_2
Output: Datově spojený dvojitě souvislý seznam D.

```
1:  D ← prázdný dvojitě souvislý seznam.
2:  Označíme všechny vrcholy v D_1 a D_2 jako nepřidané.
3:  for e ∈ edges(D_1) do
4:      source_map(e) ← 1
5:      edges(D) ← edges(D) ∪ {e}
6:      face_label(e) ← {label(face(e))}
7:      v ← origin(e)
8:      if ¬is_added(v) then
9:          vertices(D) ← vertices(D) ∪ {v}
10:         is_added(v) ← True
11:     end
12: end
13: for e ∈ edges(D_2) do
14:     source_map(e) ← 2
15:     edges(D) ← edges(D) ∪ {e}
16:     face_label(e) ← {label(face(e))}
17:     v ← origin(e)
18:     if ∃ w ∈ vertices(D).coords(w) = coords(v) then
19:         origin(e) ← w
20:     else if ¬is_added(v) then
21:         vertices(D) ← vertices(D) ∪ {v}
22:         is_added(v) ← True
23:     end
24: end
```

2.3.2 Výpočet průsečíků

Výpočet průsečíků z dvou různých podrozdělení je dalším krokem algoritmu. Za tímto účelem využijeme nám známý algoritmus zametací přímky, který pro naše účely upravíme. Výsledek je uveden v algoritmu 5 – OVERLAYSWEEPLINE (D).



Algoritmus začíná převodem hran na úsečky, se kterými se nám bude lépe pracovat. Postup převodu je uveden na řádcích 1–10.

Začneme tím, že označíme všechny hrany z $edges(D)$ jako nezpracované. Poté převádíme jednu (nezpracovanou) hranu po druhé a vytváříme z nich úsečky. Pamatujeme si přitom, ze které hrany úsečka vznikla a to díky ukazateli na ní. Abychom se však vyhnuli vytváření identických úseček, označíme rovněž její dvojče za zpracované. Nakonec získáváme množinu úseček S, se kterou můžeme dále pracovat[5].

Následuje procházení jednotlivých úseček $s \in S$, které se od původní verze příliš neliší. Jediným krokem navíc je uchovávání počtu úseček patřící k té, či oné mapě. Tuto informaci si budeme pamatovat u jednotlivých množin $U(p)$ a $L(p)$. Za tímto účelem definujme funkci $\sigma_i(S)$, kde i je označení mapy a S je množina úseček. Pak $\sigma_i(S)$ vrací počet úseček v S s původem v mapě i. Tak například $\sigma_1(U(p))$ vrací počet úseček v $U(p)$ s původem v mapě 1.

Tuto informaci bude užitečné znát při zpracování události algoritmem 6, jak ostatně uvidíme dále.

5. Při implementaci převodní procedury bychom mohli navíc uspořit výpočetní čas tím, že bychom konverzi prováděli přímo v cyklu na řádcích 13–19, nicméně asymptotická složitost by zůstala nezměněna – $O(n)$.

Algoritmus 5: OVERLAYSWEEPLINE (D)

Input: Dvojitě souvislý seznam D.
Output: D s vypočtenými průsečíky.

Označíme všechny hrany $edges(D)$ jako nezpracované.
$S \leftarrow \emptyset$
for $e \in edges(D)$ **do**
    if *hrana* e *nebyla dosud zpracována* **then**
        $s \leftarrow segment(e)$
        $edge\_pointer(s) \leftarrow e$
        $S \leftarrow S \cup \{s\}$
        Označíme $twin(e)$ jako zpracovanou hranu.
    end
end
Inicializujeme prázdnou prioritní frontu událostí Q.
Inicializujeme prázdný vyvážený binární strom $\mathcal{T}$.
for $s_i \in S$ **do**
    $p_u \leftarrow$ lexikograficky horní bod úsečky s_i
    $p_l \leftarrow$ lexikograficky dolní bod úsečky s_i
    Přidáme úsečku s_i do množin $U(p_u)$ a $L(p_l)$.
    Upravíme čítače původu úseček tak, aby funkce σ_i vracela korektní hodnotu.
    Přidáme body p_u a p_l do fronty Q.
end
while $|Q| > 0$ **do**
    Vybereme z fronty Q událost p.
    OVERLAYHANDLEEVENTPOINT(p)
end

Obsluha události

Obsluha události p doznala výraznějších změn oproti své původní verzi uvedené v algoritmu 2. Její podobu můžeme vidět v algoritmu 6 – OVERLAYHANDLEEVENTPOIN(p).

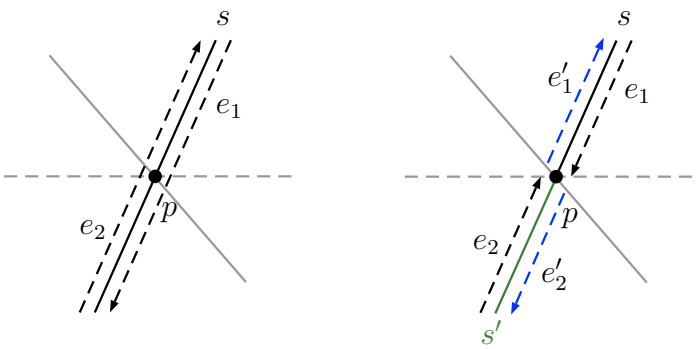
Při obsluze události vycházíme z faktu, že nehledáme průsečíky úseček, které mají původ v téže mapě – ty již známe (jinak by mapa nebyla korektní). Hledáme *pouze* takové průsečíky, které mají původ v obou mapách současně. Když je takový průsečík nalezen, provedeme potřebné úpravy v D tak, aby oblast nad p byla zpracovaná.

Pro přehlednost si postup rozčleníme do tří částí:

1. eliminace úseček $C(p)$ rozdělením na horní a dolní část,
2. *propojení* (nastavení předchůdců a následníků) hran směřujících z/do p,
3. hledání nových průsečíků pod p.

Obsluha události – eliminace úseček $C(p)$

Při obsluze události p je nezbytné provést rozdělení všech úseček v množině $C(p)$. Tento krok činíme proto, abychom vytvořili korektní dvojitě souvislý seznam. Každý vrchol v něm může být pouze začátkem nebo koncem hrany, nemůže ležet uvnitř ní.



Obrázek 2.5: Princip algoritmu SPLITCPSEGMENT(s, p)

Algoritmus 6: OVERLAYHANDLEEVENTPOINT(p)

Input: Událost p.

$U(p) \leftarrow$ úsečky jejichž horní bod je p
$L(p) \leftarrow$ úsečky jejichž dolní bod je p
$C(p) \leftarrow$ úsečky jejichž vnitřním bodem je p
for $s \in C(p)$ **do**
    $s' \leftarrow$ SPLITCPSEGMENT(s, p)
    $L(p) \leftarrow L(p) \cup \{s\}$
    $C(p) \leftarrow C(p) \setminus \{s\}$
    $U(p) \leftarrow U(p) \cup \{s'\}$
end
s_l, $s_r \leftarrow$ CLOCKWISECONNECTUPPER(p)
$left\_segment(p) \leftarrow$ úsečka nejblíže vlevo od bodu p ve stromě $\mathcal{T}$.
t_l, $t_r \leftarrow$ CLOCKWISECONNECTLOWER(p)
if $\exists\, s_l \wedge \exists\, t_l$ **then**
    DOCLOCKWISECONNECT(t_l, s_l, p)
    DOCLOCKWISECONNECT(s_r, t_r, p)
if $\exists\, s_l \wedge \neg\exists\, t_l$ **then**
    DOCLOCKWISECONNECT(s_r, s_l, p)
if $\neg\exists\, s_l \wedge \exists\, t_l$ **then**
    DOCLOCKWISECONNECT(t_l, t_r, p)
if $U(p) = \emptyset$ **then**
    $s_l \leftarrow$ úsečka nalevo od bodu p ve stromě $\mathcal{T}$
    $s_r \leftarrow$ úsečka napravo od bodu p ve stromě $\mathcal{T}$
    FINDNEWEVENT(s_l, s_r, p)
else
    $s' \leftarrow$ nejlevější úsečka z $U(p)$
    $s_l \leftarrow$ levý soused úsečky s' ve stromě $\mathcal{T}$
    FINDNEWEVENT(s', s_l, p)
    $s'' \leftarrow$ nejpravější úsečka z $U(p)$
    $s_r \leftarrow$ pravý soused úsečky s'' ve stromě $\mathcal{T}$
    FINDNEWEVENT(s'', s_r, p)
end

Rozdělení úseček provádí algoritmus 7, SPLITCPSEGMENT (s, p). Zároveň dochází k úpravě záznamů v D, přičemž na výstupu obdržíme nově vzniklou úsečku s'. Princip rozdělení úsečeky je ilustrován na obrázku 2.5.

Samotné rozdělení úsečky ale nestačí. Ještě je potřeba ji odebrat z množiny $C(p)$ a zároveň přidat její rozdělenou horní část do množiny $L(p)$. Spodní část úsečky (úsečka s') přidáme do množiny $U(p)$. Pro postup viz algoritmus 6, ř. 6–8.

Výše popsané rozdělení aplikujeme postupně na všechny úsečky v $C(p)$ a tím, že úsečky zároveň i odebíráme, nemusíme $C(p)$ v dalších krocích algoritmu uvažovat – bude to vždy prázdná množina.

Algoritmus 7: SPLITCPSEGMENT (s, p)

Input: Událost p a úsečka $s \in C(p)$
Output: Nová úsečka s' a nové hrany v D

1: $e_1 \leftarrow$ hrana úsečky s, taková že $origin(e_1) < p$.
2: $e_2 \leftarrow twin(e_1)$
3: $e'_1 \leftarrow$ nový záznam v $edges(D)$
4: $origin(e'_1) \leftarrow p$
5: $twin(e'_1) \leftarrow e_1$
6: $twin(e_1) \leftarrow e'_1$
7: $next(e'_1) \leftarrow next(e_2)$
8: $source\_map(e'_1) \leftarrow source\_map(e_1)$
9: $face\_label(e'_1) \leftarrow face\_label(e_2)$
10: $e'_2 \leftarrow$ nový záznam v $edges(D)$
11: $origin(e'_2) \leftarrow p$
12: $twin(e'_2) \leftarrow e_2$
13: $twin(e_2) \leftarrow e'_2$
14: $next(e'_2) \leftarrow next(e_1)$
15: $source\_map(e'_2) \leftarrow source\_map(e_2)$
16: $face\_label(e'_2) \leftarrow face\_label(e_1)$
17: **return** úsečka s' tvořená z hran e_2, e'_2

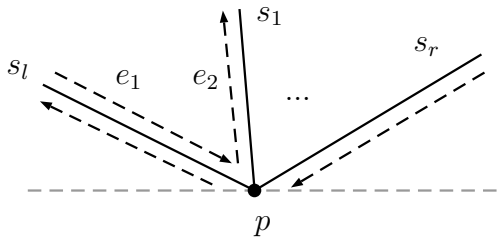
Obsluha události – propojení hran

Nyní přichází na řadu správné nastavení předchůdců a následníků hran úseček $U(p) \cup L(p)$ směřujících do/z p.

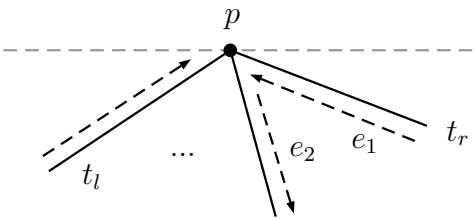
Hrany propojujeme podle následujících pravidel:

- následník hrany směřující do p je hrana nejblíže po směru hodinových ručiček směřující od p,
- předchůdce hrany směřující od p je hrana nejblíže proti směru hodinových ručiček, směřující do p.

Jelikož nemáme k dispozici všechny úsečky seřazené po/proti směru hodinových ručiček, využijeme strom $\mathcal{T}$. Z něj tuto informaci lze vyčíst, ale bude nezbytné rozdělit si problém na dvě části: horní a dolní, jak můžeme vidět na obrázcích 2.6 a 2.7.



Obrázek 2.6: Zpracování horní části – úsečky $L(p)$



Obrázek 2.7: Zpracování dolní části – úsečky $U(p)$

Princip algoritmu 8, CLOCKWISECONNECTUPPER(p), popřípadě algoritmu 9, CLOCKWISECONNECTLOWER(p), spočívá v propojení

vnitřních hran úseček. Ty postupně procházíme zprava doleva (potažmo zleva doprava) a propojujeme je pomocí jednoduchého algoritmu 10, DOCLOCKWISECONNECT(s, t, p).

Algoritmus 8: CLOCKWISECONNECTUPPER(p)

Input: Událost p a úsečky $L(p)$, $U(p)$.
Result: Propojení hran úseček z množiny $L(p)$.

1: **if** $| L(p) | > 0 \wedge \sigma_1(L(p) \cup U(p)) > 0 \wedge \sigma_2(L(p) \cup U(p)) > 0$ **then**
2: $s_l \leftarrow$ nejlevější úsečka z $L(p)$
3: $s_r \leftarrow$ nejpravější úsečka z $L(p)$
4: $s'_l \leftarrow s_l$
5: **while** $s'_l \neq s_r$ **do**
6: $s'_r \leftarrow$ pravý soused úsečky s'_l ve stromě $\mathcal{T}$
7: DOCLOCKWISECONNECT(s'_l, s'_r, p)
8: $s'_l \leftarrow s'_r$
9: **end**
10: **end**
11: Odebereme úsečky $L(p)$ ze stromu $\mathcal{T}$.
12: **return** s_l, s_r

Oba výše uvedené algoritmy 8 a 9 vrací na svém výstupu nejlevější a nejpravější úsečku zpracovávané části (pokud existují). Výstupní úsečky mohou být samozřejmě totožné: v případě, kdy existuje jen jedna úsečka v $U(p)$ nebo v $L(p)$.

Pokud je dolní část úseček prázdná ($U(p) = \emptyset$), propojíme spolu pouze hrany nejlevější a nejpravější úsečky z $L(p)$. Pokud je prázdná část horní ($L(p) = \emptyset$), propojíme spolu pouze hrany nejlevější a nejpravější úsečky z $U(p)$.

Propojování provádíme za předpokladu, že platí následující podmínka:

$$\sigma_1(L(p) \cup U(p)) > 0 \wedge \sigma_2(L(p) \cup U(p)) > 0.$$

Tedy, že existuje alespoň jedna úsečka z každé mapy. Tato podmínka je obsažena jak v algoritmu 8, tak v algoritmu 9. Oba navíc přidávají podmínku na neprázdnost zpracovávaných úseček.

Tímto způsobem se můžeme vyhnout zbytečnému propojování hran původem z téže mapy – je již hotové.

Algoritmus 9: CLOCKWISECONNECTLOWER(p)

Input: Událost p a úsečky $L(p)$, $U(p)$.
Result: Propojení hran úseček z množiny $U(p)$.

Vložíme úsečky $U(p)$ do stromu $\mathcal{T}$.
if $|\ U(p)\ | > 0 \wedge \sigma_1(L(p) \cup U(p)) > 0 \wedge \sigma_2(L(p) \cup U(p)) > 0$ **then**
    $t_l \leftarrow$ nejlevější úsečeka z $U(p)$
    $t_r \leftarrow$ nejpravější úsečeka z $U(p)$
    $t'_r \leftarrow t_r$
    while $t'_r \neq t_l$ **do**
        $t'_l \leftarrow$ levý soused úsečky t'_r ve stromě $\mathcal{T}$
        DOCLOCKWISECONNECT(t'_r, t'_l, p)
        $t'_r \leftarrow t'_l$
    end
    return t_l, t_r
end

Algoritmus 10: DOCLOCKWISECONNECT(s, t, p)

Input: Událost p, která je společným koncovým bodem úseček s, t.
Result: Nastaví předchůdce a následníka hran úseček s, t při události p.

$e_1 \leftarrow$ hrana úsečky s taková, že $origin(e_1) \neq p$
$e_2 \leftarrow$ hrana úsečky t taková, že $origin(e_2) = p$
$next(e_1) \leftarrow e_2$
$prev(e_2) \leftarrow e_1$

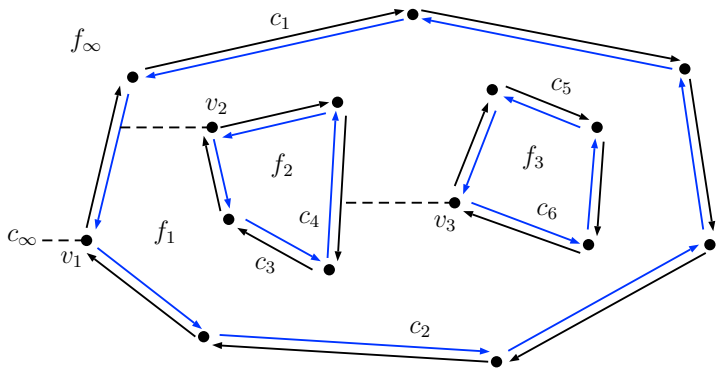
2.3.3 Zpracování oblastí

Nyní, když známe všechny průsečíky a máme správně propojeny hrany, můžeme přistoupit k určování oblastí nového podrozdělení. To budeme provádět ve třech krocích:

1. zjištění hranic oblastí,
2. sestavení speciálního grafu a nalezení jeho komponent souvislosti,
3. vytvoření odpovídajících záznamů ve $faces(D)$.

Hranice oblasti

Každá oblast podrozdělení je určena svou vnější hranicí a libovolným počtem hranic vnitřních. Hranici oblasti tvoří posloupnost hran $e_1, e_2, ..., e_n, e_{n+1} = e_1$, přičemž platí, že hrana e_{i+1} je následníkem hrany e_i. Takovou posloupnost hran budeme nazývat cyklem a označovat jako c_i. Platí, že počet vnějších hranic odpovídá počtu omezených oblastí. Jedna oblast je neomezená (bez vnější hranice). Pro ni můžeme vytvořit fiktivní vnější cyklus, který budeme označovat c_∞.

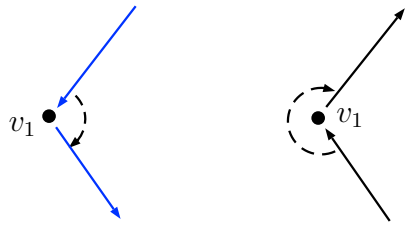


Obrázek 2.8: Cykly podrozdělení

Na obrázku 2.8 můžeme vidět jednotlivé hranice oblastí v rovinném podrozdělení. Černě jsou znázorněny vnitřní hranice a modře jsou znázorněny vnější hranice oblastí. Tak například oblast f_1 je vymezena vnější hranicí c_2 a vnitřními hranicemi c_3 a c_5.

Určení vnější a vnitřní hranice

Typ hranice určíme podle úhlu, který svírají hrany procházející nejlevějším bodem cyklu (v případě, že jich existuje více, vezmeme nejnižší bod). Pokud je tento úhel menší než $180°$, jedná se o vnější hranici oblasti. V opačném případě se jedná o vnitřní hranici oblasti.



Algoritmy 11 a 12 nám umožňují zjitit typ hranice podle svíraného úhlu (zjišťujeme výpočtem znaménka determinantu vhodné matice 2×2).

Algoritmus 11: INNER (c)

Input: Cyklus c s určeným nejlevějším bodem a hranou z něj vycházející $leftmost\_edge(c)$.

Output: Pravda, pokud cyklus tvoří vnitřní hranici oblasti.

1: $e \leftarrow leftmost\_edge(c)$
2: $p \leftarrow origin(prev(e))$
3: $q \leftarrow origin(e)$
4: $r \leftarrow origin(next(e))$
5: **return** $(q_x - p_x) * (p_y - r_y) - (r_x - p_x) * (p_y - q_y) < 0$

Algoritmus 12: OUTER (c)

Input: Cyklus c

Output: Pravda, pokud je cyklus na vstupu vnější hranicí oblasti, jinak nepravda.

1: **return** $\neg$INNER(c)

Zjištění cyklů

Seznam všech cyklů v rovinném podrozdělení vrací na svém výstupu algoritmus 13, OVERLAYCYCLES(D). Ten navíc ke každému cyklu ukládá informaci o jeho nejlevějším bodě – $leftmost\_vertex(c)$ a hraně z něj vycházející – $leftmost\_edge(c)$.

Princip algoritmu spočívá v postupném procházení nezpracovaných hran z $edges(D)$. Z jedné nezpracované hrany postupně procházíme její následníky až narazíme na hranu, u které jsme začali. Tím algoritmus projde celý cyklus. Navštívené hrany přitom označíme jako zpracované a poznamenáme si u nich také cyklus, do kterého patří (například ukazatelem).

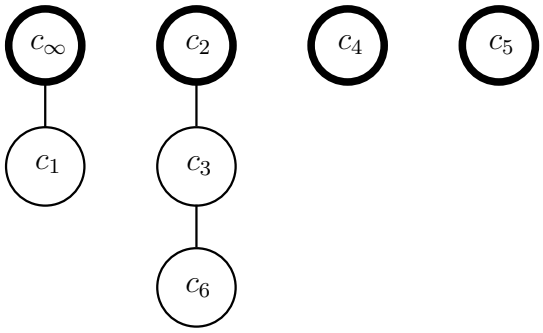
Navíc je vhodné (uvidíme později) poznamenat si názvy přilehlých oblastí hran, které cyklus tvoří (ř. 5, 24–25).

Každý cyklus je jednoznačně určen svým indexem – $index(c)$.

Komponenty souvislosti

Jak už bylo řečeno, každá oblast podrozdělení má svou vnější hranici a libovolný počet hranic vnitřních. Abychom mohli vložit záznamy o oblastech do $faces(D)$ je nezbytné určit, které hranice vymezují plochu jedné oblasti.

Řešení problému spočívá v nalezení *komponent souvislosti* v následujícím grafu. Uzly grafu tvoří jednotlivé cykly, hrany pak vytváříme následovně:



Obrázek 2.9: Komponenty souvislosti z obrázku 2.8

Algoritmus 13: OVERLAYCYCLES (D)

Input: Dvojitě souvislý seznam D

Output: Seznam cyklů vnitřních a vnějších hranic C s nejlevějším bodem a hranou.

Označíme všechny hrany v $edges(D)$ jako nezpracované.
$C \leftarrow$ prázdný seznam cyklů indexovaných od nuly.
$c_\infty \leftarrow$ uměle vytvořený cyklus vnější hranice (oblast f_∞)
$index(c_\infty) \leftarrow 0$
$face\_labels(c_\infty) \leftarrow \{(1, "f_\infty"), (2, "f_\infty")\}$
Přidáme cyklus c_∞ do seznamu C.
$i \leftarrow 1$
for $e \in$ *nezpracovaná hrana z* $edges(D)$ **do**
    $c \leftarrow$ nový cyklus
    $index(c) \leftarrow i$
    $face\_labels(c) \leftarrow \emptyset$
    $v \leftarrow origin(e)$
    $leftmost\_vertex(c) \leftarrow v$
    $leftmost\_edge(c) \leftarrow e$
    $e' \leftarrow e$
    repeat
        $cycle(e') \leftarrow c$
        $v' \leftarrow origin(e')$
        if $v'_x < v_x \vee (v'_x = v_x \wedge v'_y < v_y)$ **then**
            $leftmost\_vertex(c) \leftarrow v'$
            $leftmost\_edge(c) \leftarrow e'$
            $v \leftarrow v'$
        end
        $l \leftarrow \{(source\_map(e), face\_label(e))\}$
        $face\_labels(c) \leftarrow face\_labels(c) \cup l$
        $e' \leftarrow next(e')$
        Označíme hranu e' jako zpracovanou.
    until $e \neq e'$
    Přidáme cyklus c do seznamu C pod indexem i.
    $i = i + 1$
end
return C

Procházíme jeden vnitřní cyklus za druhým a hledáme k němu jeho levého souseda. To děláme tak, že vezmeme nejlevější bod cyklu (pokud je jich více, rozhoduje ten nejníže položený) a vedeme z něj myšlenou horizontální čáru vlevo. Tam, kde se čára jako první protne s jinou úsečkou dostáváme opačně orientované hrany dvou cyklů. Vybereme z něj ten, který má shodnou přilehlou oblast s právě zpracovávaným cyklem. Tyto cykly v grafu spojíme hranou [3].

Nalézt sousední cyklus vlevo není obtížné, pokud si u zpracovávané události pamatujeme usečku nejblíže nalevo od ní (tak činíme v algoritmu 6 na ř. 11).

Po vytvoření všech takových hran daného grafu dostáváme komponenty souvislosti, které představují jednotlivé oblasti. Uzly komponenty tvoří vnitřní a vnější hranice dané oblasti.

Když bychom se vrátili k podrozdělení z obrázku 2.8, pak by sestavené komponenty souvislosti byly shodné s těmi, které jsou uvedeny na obrázku 2.9. Navíc si můžeme u podrozdělení povšimnout přerušovaných horizontálních čar, které symbolizují propojování sousedních cyklů. Vidíme tak, že levým sousedem cyklu c_6 je cyklus c_3. Cykly vnější hranice jsou u komponent znázorněny zvýrazněným kruhem.

Algoritmické vytvoření komponent souvislosti

Algoritmus 14 představuje jeden z možných způsobů jak komponenty souvislosti vytvářet.

Prvním krokem je vytvoření prázdného seznamu komponent. Do něj budeme postupně ukládat jednotlivé komponenty. Dále definujeme funkci $component(c)$ na cyklu c, která vrací komponentu cyklu. Tedy komponentu, ve které se cyklus nachází.

Následuje postupné zpracovávání cyklů. Pokud má cyklus přiřazenou komponentu souvislosti, neděláme nic. V opačném případě provedeme následující kroky.

Pokud cyklus tvoří *vnější hranici* oblasti, pak k němu vytvoříme zcela novou komponentu souvislosti (ř. 5) a uložíme k ní potřebné informace (ř. 6–8).

Algoritmus 14: OVERLAYCOMPONENTS (C)

Input: Seznam cyklů C
Output: Seznam komponent souvislosti G

```
1:  G ← prázdný seznam komponent souvislosti
2:  foreach c ∈ C do
3:      if ¬∃ component(c) then
4:          if OUTER(c) then
5:              g ← nová komponenta v G
6:              outer(g) ← c
7:              inners(g) ← ∅
8:              component(c) ← g
9:          else
10:             C' = {c}
11:             repeat
12:                 c' ← cyklus nalevo od cyklu c
13:                 if ¬∃ c' then
14:                     c' ← c_∞ (index 0 v seznamu cyklů)
15:                 if OUTER(c') ∧ ¬∃ component(c') then
16:                     Totéž, co na řádcích 6–9, ale s cyklem c'.
17:                 end
18:                 if INNER(c') ∧ ¬∃ component(c') then
19:                     C' ← C' ∪ {c'}
20:             until ¬∃ component(c')
21:             g ← component(c')
22:             foreach c' ∈ C' do
23:                 component(c') ← g
24:                 inners(g) ← inners(g) ∪ {c'}
25:             end
26:         end
27: end
```

Pokud cyklus představuje *vnitřní hranici* oblasti, je situace složitější. Musíme k němu totiž nalézt související cyklus (cyklus nalevo od nejlevějšího bodu), který má buď to nastavenou komponentu souvislosti nebo se jedná o vnější hranici oblasti, ke které komponentu souvislosti vytvoříme (ř. 15–17).

Jak můžeme vidět, algoritmus nepostupuje přesně tak, jak bylo uvedeno v myšlence sestavování komponent souvislosti, nicméně jeho výstup je korektní. Navíc, a to je důležitější, pracuje v časové složitosti $O(n)$ vzhledem k n-cyklům na vstupu. Nezvyšuje tedy celkovou časovou složitost překryvu map (viz sekce 2.5).

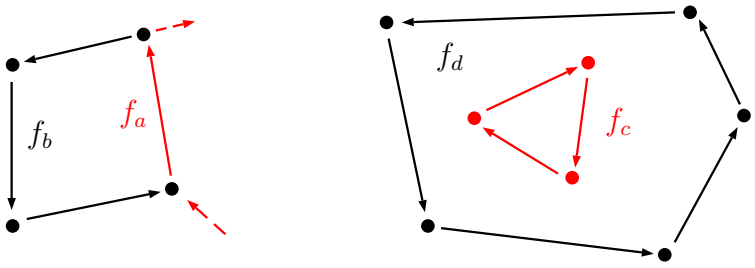
Určení původních názvů oblastí

Ještě předtím, než provedeme převod komponent souvislosti do tabulky oblastí $faces(D)$, určíme jejich původní názvy.

Platí, že každá nová oblast (reprezentovaná komponentou souvislosti) pochází právě z jedné oblasti první mapy a právě z jedné oblasti druhé mapy. Touto oblastí může být samozřejmě i neomezená oblast f_∞.

V zásadě existují dva možné způsoby, jak původní názvy zjistit.

První způsob spočívá v tom, že projdeme všechny cykly, ze kterých je komponenta souvislosti tvořena. Každý z cyklů totiž obsahuje informaci o původních přilehlých oblastech hran, které jsme získali v algoritmu 13.



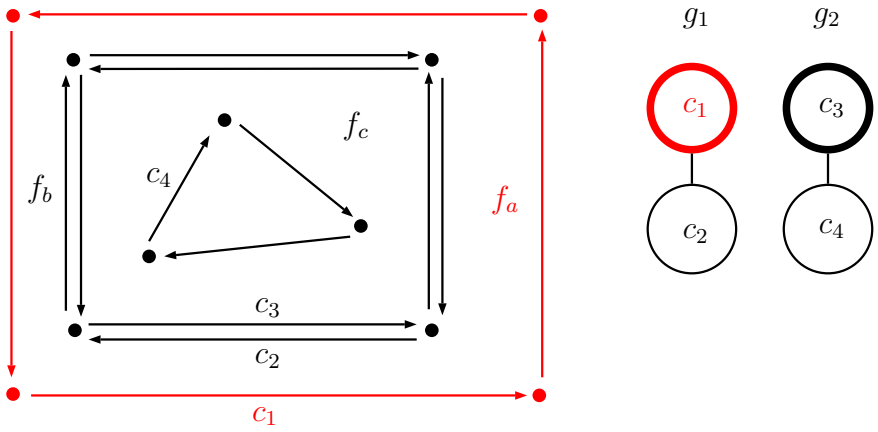
Obrázek 2.10: Získání názvu oblastí z cyklů komponenty

V levé části obrázku 2.10 vidíme cyklus, který je tvořen ze tří hran původem z první mapy (vyznačeny černě) a jedna hrana původem z mapy druhé (vyznačena červeně). V pravé části pak vidíme vnější

cyklus původem z první mapy a vnitřní cyklus původem z druhé mapy.

Názvy oblastí poznamenané u cyklů jsou uloženy jako množina uspořádaných dvojic $(i, label)$, kde i je číslo původní mapy a $label$ je název původní oblasti. Je přitom zřejmé, že množina názvů oblastí může mít maximálně dva prvky. Pokud jich je méně (jeden), musíme zjistit původní oblast z chybějící mapy. To provedeme druhým způsobem.

Ten spočívá v tom, že budeme postupně procházet vnější hranice nadřazených komponent, dokud název oblasti chybějící mapy nezískáme. Nadřazenou komponentou myslíme takovou komponentu, ve které se nachází cyklus opačný od cyklu vnější hranice aktuální komponenty.



Obrázek 2.11: Překryv a komponenty souvislosti

Uveďme si to na příkladu z obrázku 2.11. Máme zde komponentu g_2, která je jednobarevná (neobsahuje hrany původem z různých map). Přesto leží v oblasti z druhé mapy a sice oblasti f_a. Vezmeme proto cyklus opačný (c_2) k cyklu vnější hranice (c_3) a zjistíme, ve které komponentě se nachází – $component(c_2) = g_1$. Z vnějšího cyklu komponenty g_1 získáváme chybějící původní oblast f_a.

Algoritmus 15, FINDFACELABELS (G), vykonává výše uvedený postup.

Algoritmus 15: FINDFACELABELS (G)

Input: Seznam komponent souvislosti G
Result: Nastavené popisky oblastí u komponent souvislosti G

```
1:  for g ∈ G do
2:      L ← ∅
3:      for c ∈ {outer(g)} ∪ inners(g) do
4:          L ← L ∪ face_labels(c)
5:          if | L | = 2 then
6:              Ukonči procházení cyklů.
7:      end
8:      if | L | < 2 then
9:          g' ← g
10:         repeat
11:             c ← outer(g')
12:             e ← twin(leftmost_edge(c))
13:             g' ← component(cycle(e))
14:             c' ← outer(g')
15:             Přidáme do L jen takovou uspořádanou dvojici z
                face_labels(c'), která pochází z chybějící mapy.
16:         until | L | < 2
17:     end
18:     face_labels(g) ← L
19: end
```

Vytvoření záznamů ve $faces(D)$

Nyní zbývá jediné: vytvořit odpovídající záznamy oblastí v tabulce $faces(D)$. Ty vytvoříme podle obdržených komponent souvislosti následovně:

Nejprve vytvoříme nový záznam oblasti ve $faces(D)$. Nastavíme vnější a vnitřní hranice podle komponenty souvislosti a stejně tak nastavíme nový název oblasti.

Poté projdeme všechny hrany hraničních cyklů a nastavíme ukazatel na novou přilehlou oblast.

Pseudokód je uveden v algoritmu 16, OVERLAYFACES(G).

Algoritmus 16: OVERLAYFACES (G)

Input: Seznam komponent souvislosti G
Result: Spočítaná mapa překryvu D

```
1:  for g ∈ G do
2:      f ← nový záznam v tabulce faces(D)
3:      E ← ∅
4:      for c ∈ inners(g) do
5:          E ← E ∪ {leftmost_edge(c)}
6:      end
7:      inner_edges(f) ← E
8:      outer_edge(f) ← leftmost_edge(outer(g))
9:      label(f) ← face_labels(g)
10:     foreach c ∈ {outer(g)} ∪ inners(g) do
11:         e ← leftmost_edge(c)
12:         e' ← e
13:         repeat
14:             face(e') ← f
15:             e' ← next(e')
16:         until e ≠ e'
17:     end
18: end
```

2.4 Úplný algoritmus

Spojením jednotlivých algoritmů do jednoho celku dostáváme úplnou verzi překryvu map.

Pseudokód je uveden v algoritmu 17, MAPOVERLAY (D_1, D_2).

Algoritmus 17: MAPOVERLAY (D_1, D_2)

Input: Dvojitě souvislé seznamy D_1, D_2.
Output: Spočítaný překryv map D

1: $D \leftarrow$ DCELMERGE (D_1, D_2)
2: OVERLAYSWEEPLINE (D)
3: $C \leftarrow$ OVERLAYCYCLES (D)
4: $G \leftarrow$ OVERLAYCOMPONENTS (C)
5: OVERLAYFACES (G)
6: **return** D

2.5 Časová složitost překryvu map

Časová složitost překryvu dvou map je odvozena od jejich komplexity (počtu záznamů v dvojitě souvislém seznamu).

Věta 2.5.1. *Nechť S_1 je rovinné podrozdělení s komplexitou n_1, S_2 je rovinné podrozdělení s komplexitou n_2 a nechť $n = n_1+n_2$. Pak překryv dvou podrozdělení S_1 a S_2 může být proveden v časové složitosti $O(n \log n + k \log n)$, kde k je komplexita překryvu.*

Důkaz věty 2.5.1 je uveden v [3].

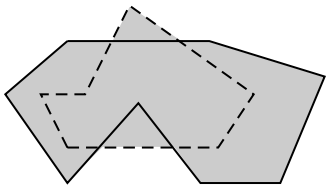
2.6 Booleovy operace nad polygony

Jednou z aplikací překryvu map jsou Booleovy operace nad dvěma polygony. Výsledkem operace je nové rovinné podrozdělení, které není nutně polygonem [3].

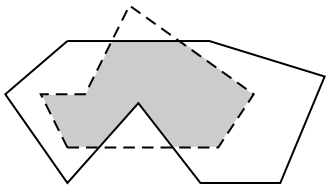
Předpokládejme existenci polygonu $\mathcal{P}_1$, který je zadán rovinným podrozdělením S_1 jako oblast nesoucí označení $\mathcal{P}_1$. Obdobně předpokládejme existenci polygonu $\mathcal{P}_2$. Každému polygonu přiřadíme

rovinné podrozdělení se dvěma oblastmi, kde jedna oblast je polygon a druhá její doplněk.

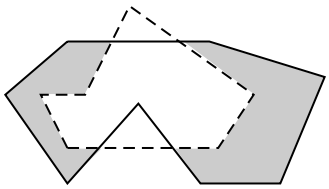
Sjednocení polygonů $\mathcal{P}_1 \cup \mathcal{P}_2$ provedeme tak, že v překryvu zahrneme pouze ty oblasti, které nesou označení buď to jen $\mathcal{P}_1$, nebo jen $\mathcal{P}_2$, popřípadě obojí.



Průnik dvou polygonů $\mathcal{P}_1 \cap \mathcal{P}_2$ provedeme tak, že v překryvu zahrneme pouze takové oblasti, které nesou označení jak $\mathcal{P}_1$, tak $\mathcal{P}_2$ zároveň.



Rozdíl dvou polygonů $\mathcal{P}_1 \setminus \mathcal{P}_2$ provedeme tak, že ve výsledném překryvu zahrneme pouze takové oblasti, které nesou označení $\mathcal{P}_1$, ale ne $\mathcal{P}_2$.



Kapitola 3

Ukázkové programy

Ukázkové programy tvoří nedílnou součást této diplomové práce. Jsou celkem dva:

1. Sweepline
2. MapOverlay

Oba programy implementují stejnojmenné algoritmy a umožňují jejich praktickou ukázku.

3.1 Použité technologie

Cílem bylo použít takové technologie, které by umožňovaly co největší přenositelnost programů napříč operačními systémy. Neméně důležitým cílem byla také dobrá čitelnost zdrojového kódu.

Programovací jazyk Python nabízí obojí [7] a z tohoto důvodu byl vybrán pro implementaci programů. Využita byla jeho verze 2.7.

3.1.1 Python

Python patří mezi nejrozšířenější a nejoblíbenější skriptovací jazyky současnoti. Kombinuje imperativní, objektové a funkcionální prvky programování, přičemž zachovává výbornou čitelnost zdrojového kódu. Ta je mimo jiné dána i specifickou formou syntaxe, při které se jednotlivé bloky kódu tvoří jejich odsazením [7].

Další nespornou výhodou je rozsáhlá standardní knihovna, díky níž můžeme výrazným způsobem omezit využití knihoven třetích stran [1].

Python je zpravidla standardní součástí všech linuxových distribucí. Nachází se rovněž na systémech Mac OS X. Výjimku tvoří operační systémy Windows, u kterých je nezbytné provést ruční instalaci.

3.1.2 Blist

Jedinou výjimku tvoří modul *Blist*, který není součástí standardní knihovny. Obsahuje datové struktury `sortedlist` a `sortedset` [6], které v programech využíváme.

V prvním případě se jedná o řazené pole dle zvoleného klíče (může být i funkce), přičemž operace na něm prováděné (vložení, smazání, vyhledání prvku) se realizují se shodnou časovou složitostí jako v binárním vyhledávacím stromě [6]. Ten je ostatně použit v implementaci pole.

Druhá zmiňovaná struktura je rovněž seřazeným polem, má však chování množin: nemůžeme mít v poli více než jeden unikátní objekt, nehledě na to, kolik jich do pole přidáme (tato operace neskončí chybou). Strukturu využíváme při práci s množinami $U(p)$, $C(p)$, $L(p)$, které jsou uvedeny v předchozích kapitolách.

Instalace

Instalaci knihovny můžeme provést pomocí standardního nástroje Easy Install:

```
easy_install blist
```

nebo pomocí sofistikovanějšího správce balíčků PIP:

```
pip install blist
```

3.1.3 Tkinter

Grafické uživatelské rozhranní obou programů využívá standardní knihovny Tkinter[1], která je součástí jazyka Python [1].

Tkinter umožňuje používání standardních ovladacích prvků jako jsou tlačítka, textová pole, popisky atp. (souhrně widgety) a jednak použití tzv. plátna (Canvas), na které můžeme vykreslovat geometrické útvary [5].

1. Jedná se o zkratku z anglického Tk interface.

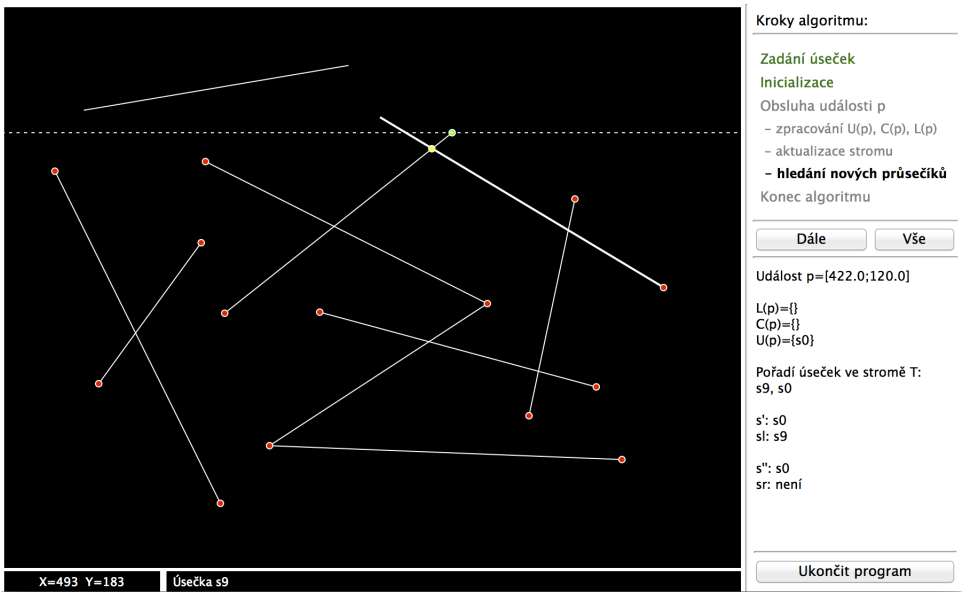
3.1.4 Pomocné třídy

Vzhledem k tomu, že v obou programech pracujeme s geometrickými objekty, je k dispozici pomocný modul `geotools.py`, který definuje třídu `Point` reprezentující rovinný bod a třídu `Segment` reprezentující úsečku.

Bod má navíc definové uspořádání dle lexikografického uspořádání 1.3. Úsečka pak obsahuje metodu na výpočet průsečíku s jinou úsečkou atp.

3.2 Program Sweepline

Algoritmus zametací přímky demonstruje program Sweepline. Jeho zdrojové kódy se nachází v souboru `sweepline.py`. Po spuštění souboru interpretem jazyka Python uvidíme obrazovku rozdělenou na dvě části: levou a pravou (viz obrázek 3.1).



Obrázek 3.1: Obrazovka programu Sweepline.

Levá část programu představuje plochu, na které se zobrazují úsečky a grafické operace s nimi. Vlevo dole můžeme vidět aktuální

pozici kurzoru, vzhledem k hornímu pravému rohu okna. Vpravo od souřadnic se nachází informativní popis objektu, který je nejblíže ke kurzoru myši. Uživatel tak může pohodlně zjistit význam objektu, který jej zajímá.

V pravé části aplikace se nachází přehled jednotlivých kroků algoritmu a jednotlivé ovládací prvky.

3.2.1 Ovládání programu

Uživatel začíná tím, že zadává jednotlivé úsečky myší, na kterých chce algoritmus simulovat. Poté klikne na tlačítko *Dále*, které jej přesune na další krok algoritmu. Aktuální fáze algoritmu je zvýrazněna tučným řezem písma. Již prošlé kroky se zobrazují zelenou barvou.

Má-li jeden krok algoritmu více dílčích kroků, lze je všechny provést kliknutím na tlačítko *Vše*. Toho lze například využít u kroku *Inicializace*, při kterém se přidávají do fronty událostí krajní body úseček. Uživatel se tak nemusí proklikávat každou úsečkou zvlášť.

Program lze v kteroukoliv dobu ukončit kliknutím na tlačítko *Ukončit program* situované v pravém dolním rohu.

3.2.2 Významy barev bodů

V průběhu algoritmu se objevuje hned několik možných obarvení bodů. Zde je jejich význam:

- *červený* – nezpracovaný bod události, pocházející z koncového bodu úsečky,
- *zelený* – aktuálně zpracovávná událost,
- *žlutý* – objevený průsečík, doposud nezpracovaný,
- *bílý* – zpracovaný průsečík úseček.

3.3 Program MapOverlay

Algoritmus průniků map implementuje program MapOverlay. Jeho zdrojový kód se nachází v souboru `mapoverlay.py`.

3.3.1 Ovládání programu

Ovládání programu se nikterak neliší od ovládáni toho předchozího. Výjimkou je zadávání vstupních dat, které se neděje ručně (z důvodu složitosti), ale načtením mapy již hotové ze souboru. Jeho formát je popsán v následující části.

3.3.2 Struktura souboru s mapou

Pro uchovávání mapy v souboru je využit značkovací jazyk XML. Struktura značek a jejich atributů je následující:

```
<map>
  <vertices>
   <vertex id="v1" x="10" y="15" edge="e1" />
   ...
  </vertices>
  <edges>
    <edge id="e1" origin="v1" twin="e5" next="e2"
          face="f0"/>
    ...
  </edges>
  <faces>
    <face id="f0">
      <border type="inner" edge="e1" />
    </face>
    <face id="f1">
      <border type="outer" edge="e5" />
    </face>
    ...
  </faces>
</map>
```

Všechna data se nachází uvnitř párové značky `<map>`. Nalezneme zde tři seznamy, které kopírují strukturu dvojitě souvislého seznamu: data o vrcholech `<vertices>`, hranách `<edges>` a oblastech `<faces>`.

Atribut `id` je unikátním identifikačním pojmenováním objektu a nesmí se tedy stát, že by existovaly dva objekty se stejným iden-

tifikátorem. Taková situace bude programem považována za chybu na vstupu.

Ostatní atributy vrcholů, hran nebo oblastí se řídí konvenčním pojmenováním, které v této práci běžně užíváme.

U záznamů oblastí rozlišujeme vnější a vnitřní hranici atributem `type` u značky `<border>`. Více než jedna vnější hranice u oblasti znamená chybu na vstupu.

Předdefinované mapy jsou k nalezení v adresáři `maps` na přiloženém CD a lze je použít jako vstup programu.

Závěr

Závěrem lze konstatovat, že cíle stanovené v úvodu této práce byly splněny.

Přinosem v problematice výpočtu průsečíků úseček je definování způsobu práce s binárním stromem $\mathcal{T}$. V literatuře [2, 3, 9] totiž není explicitně uvedeno, jakým způsobem se rozhodovat při procházení stromu vůči zadanému bodu. Jeden z možných způsobů je popsán v podsekci 1.5.2.

Největší objem původní práce však představuje kapitola druhá: prakticky všechny uvedené pseudokódy musely být nově vymyšleny. Mnohé postupy byly navíc oproti těm z [3] vylepšeny a zjednodušeny.

Naprogramování dvou ukázkových aplikací je neméně významným přínosem. Vzhledem k tomu, že programy prakticky nejsou závislé na zvoleném prostředí, může se s touto problematikou seznámit velké množství zájemců.

Možnosti k dalšímu rozvoji vidím v ošetření některých speciálních situací na vstupu, jako jsou dvě navzájem se překrývající úsečky. Dále bude možné rozšiřovat funkcionalitu ukázkových programů dle přání uživatelů.

Seznam algoritmů

Literatura

[1] Python 2.7.7 documentation. 18.5.2014, [online].
URL [https://docs.python.org/2/]

[2] Bentley, J. L.; Ottmann, T. A.: *Algorithms for reporting and counting geometric intersections*. IEEE Trans. Comput., 1979.

[3] Berk, M.; Kreveld, M.; Cheong, O.; aj.: *Computational Geometry*. Springer, 2008, ISBN 978-3-540-77973-5.

[4] Chazelle, B.; Edelsbrunner, H.: An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM 39*, 1992: s. 1–54.

[5] Lundh, F.: An Introduction to Tkinter. 2005, [online].
URL [http://elfbot.org/tkinterbook/]

[6] Stutzbach, D.: Blist Documentation. 2010, [online].
URL [http://stutzbachenterprises.com/blist/]

[7] Summerfield, M.: *Python 3*. Computer press, 2010, ISBN 978-8-025-12737-7.

[8] Wróblewski, P.: *Algoritmy*. Computer press, 2004, ISBN 80-251-0343-9.

[9] Čadek, M.: Geometrické algoritmy, kap. 2. 2013, [online].
URL [http://tinyurl.com/qzzk5xb]

Příloha A

Obsah přiloženého CD

Přiložený kompaktní disk je nedílnou součástí této práce. Obsah kořenového adresáře je následující:

maps Adresář s předdefinovanými mapami.

geotools.py Modul usnadňující práci s geometrickými objekty.

dcel.py Modul pro práci s dvojitě souvislým seznamem.

sweepline.py Program Sweepline.

mapoverlay.py Program MapOverlay.