ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta stavební

DIPLOMOVÁ PRÁCE

Praha 2009 **Jana HRNČÍŘOVÁ**

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební
Katedra mapování a kartografie

Algoritmy geometrické generalizace pro PostGIS

Diplomová práce

Vedoucí diplomové práce:
Ing. Jiří Cajthaml, Ph.D.

Vypracovala:
Bc. Jana Hrnčířová

Prohlášení:

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně, pouze za použití uvedených pramenů, citované literatury a software.

V Čáslavi dne 12.12. 2008

……

Jana Hrnčířová

Poděkování:

Děkuji touto cestou vedoucímu diplomové práce panu Ing. J. Cajthamlovi, Ph.D. za jeho ochotu, cenné rady a připomínky poskytované v průběhu vypracovávání diplomové práce.

Abstrakt:

HRNČÍŘOVÁ, Jana: Algoritmy geometrické generalizace pro PostGIS. České vysoké učení technické v Praze. Diplomová práce. Leden 2009

Diplomová práce se zabývá generalizací a algoritmy, které se používají pro geometrickou generalizaci. Po teoretickém úvodu, ve kterém je obecně popsána generalizace a její metody, následuje rešerše literatury a popis vybraných algoritmů. V rámci této práce je vytvořena funkce, která umožňuje generalizaci geometrických objektů v prostorové databázi PostGIS. Funkce je napsána v jazyce Java a z prostředí PostGIS spouštěna s využitím PL/Java.

Klíčová slova:

Geografický informační systém (GIS), generalizace, PL/Java, PostGIS, Douglas-Peuckerův algoritmus, Whyattův algoritmus, Reumann-Witkamův algoritmus, Langův algoritmus, prostorová databáze, geografické objekty

Abstract:

HRNČÍŘOVÁ, Jana: Algorithms of geometric generalization for PostGIS. Czech Technical University in Prague. Diploma work. January 2009

The diploma work is focused on generalization and algorithms for geometric generalization. The first part of this work defines and describes generalization and its methods. After literature search follows characterization of selected algorithms. The component of this work is function in Java, which can be run from spatial database PostGIS.

Keywords:

Geographic information system (GIS), generalization, PL/Java, PostGIS, Douglas-Peucker algorithm, Whyatt algorithm, Reumann-Witkam algorithm, Lang algorithm, spatial database, geographic objects

Obsah

1 Úvod

V souvislosti s rozvojem digitální kartografie přestává být proces sestavování mapy a s ním spojené generalizace závislý na osobě kartografa, ale objevují se snahy o co největší automatizaci a generalizaci řízenou algoritmy počítačových programů.

Výzkumu teoretických zásad a matematickému vyjádření kartografické generalizace bylo věnováno značné úsilí. První matematické formulace některých dílčích řešení generalizace se objevily již po 2. světové válce v pracích sovětských kartografů. Tyto poznatky se týkaly hlavně výběru sídel na mapách, protože jsou to objekty diskrétního charakteru, jejichž prostorové rozlišení a kvalitativní i kvantitativní charakteristiky lze dobře sledovat a vyhodnocovat.

V 60. letech 19. století předběhla výpočetní technika uplatnění v kartografické praxi a to způsobilo zvýšený zájem o automatizaci procesu tvorby map. Byly formulovány zákony výběru ve tvaru odmocniny (F. Töpfer v roce 1962) a dále zákony výběru ve tvaru mocninných funkcí (E. Srnka v roce 1968). Vývoj algoritmů pro automatickou kartografickou generalizaci probíhá od 70. let.

Dnes je digitální produkce map samozřejmostí, ale stupeň automatizace některých přípravných prací stále není příliš vysoký. Aktuálním problémem je interaktivní generalizace při zobrazování map na internetu. Geografická data jsou totiž uložena v databázi na serveru a generalizace dat by značně urychlila jejich přenos a vykreslování ve webovém prohlížeči.

Cílem práce je prostudovat dostupnou literaturu a následně použít vybrané generalizační algoritmy v praxi.

1.1 Kartografická generalizace

Každé kartografické dílo je abstrakcí a zjednodušením obrazu reálného světa. Ať už jsou tato díla v klasické papírové podobě nebo ve formě digitální, bez generalizace se při jejich tvorbě neobejdeme.

Definice kartografické generalizace dle ČSN 73 046: Kartografická generalizace spočívá ve výběru, geometrickém zjednodušení a zevšeobecnění objektů, jevů a jejich vzájemných vztahů pro jejich grafické vyjádření v mapě, ovlivněné účelem, měřítkem mapy a vlastním předmětem kartografického znázorňování.

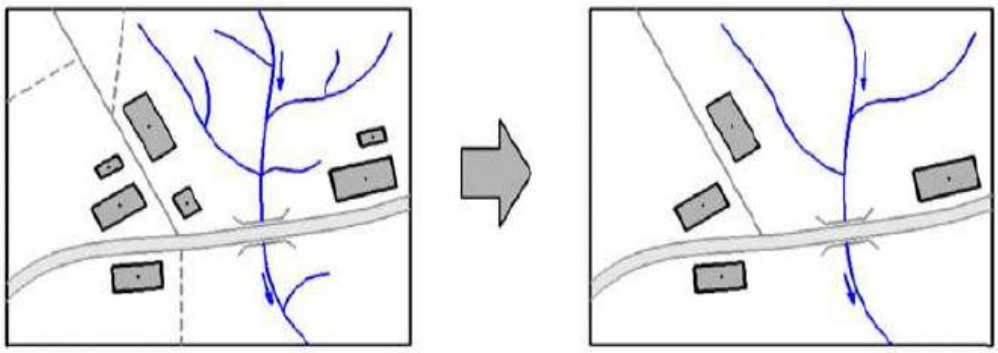
K procesu generalizace existují dva přístupy. Jedním z nich je tzv. geoprostorová generalizace, při které přechodem z většího měřítka do menšího dochází pouze ke zjednodušování tvaru prvků. Tyto postupy neřeší vztahy mezi jednotlivými prvky mapy. Druhým přístupem je právě kartografická generalizace, jež řeší konflikty mezi prvky mapy, aby nedocházelo ke ztrátě souvislostí a aby bylo možno provést tisk kartografického díla v daném měřítku.

Metody kartografické generalizace lze dle [1] rozdělit do několika skupin. Jednotlivými metodami se budou zabývat následující kapitoly.

1.1.1 Generalizace výběrem

Výběr prvků může být prováděn na základě dvou různých metod. Jedná se o cenzální výběr a normativní výběr. Oba tyto způsoby jsou detailně popsány v [2].

Cenzální výběr je založen na stanovení kvalitativní nebo kvantitativní hranice, od níž se v mapě zobrazí veškeré prvky reality. Prvky tuto hranici nesplňující se nezobrazí. Tento způsob výběru je běžný u map velkých a středních měřítek. Jeho výhodou je snadná aplikace při rozhodování, zda prvek má být zobrazen, nevýhodou je schematičnost nerespektující komplex dalších působících činitelů vyjadřujících makrostrukturu okolí zájmového prvku. Nepřihlíží k charakteru území, k významu vyjadřovacích prostředků a ani k zaplnění mapy kresbou. Obrázek 1 ukazuje generalizaci, při níž byl uplatněn cenzální výběr.



Obrázek 1: Ukázka cenzálního výběru (převzato z [3])

Cílem normativního výběru je stanovit procentní normu, tj. kolik procent objektů v realitě bude zobrazeno na mapě. Při určování normy se vychází buď z jednoduchého zákona odmocniny (1), který zachovává stejnou grafickou zátěž v podkladové a odvozené mapě jiného měřítka, nebo rozšířeného zákona odmocniny (2). Rozšířený zákon odmocniny již na rozdíl od předchozího respektuje význam prvku a případné jiné rozměry mapových značek na odvozené mapě.

$$n_o = n_p \sqrt{\frac{M_p}{M_o}} \qquad (1)$$

$$n_o = n_p \cdot c_v \cdot c_z \cdot \sqrt{\frac{M_p}{M_o}} \qquad (2)$$

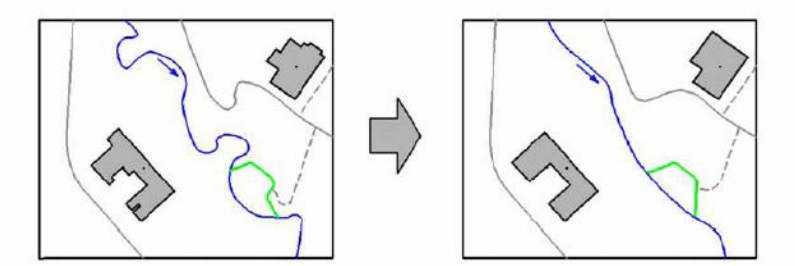
kde:

n_o	výsledný počet prvků na odvozené mapě
n_p	počet prvků na podkladové mapě
M_o	měřítkové číslo odvozené mapy
M_p	měřítkové číslo podkladové mapy
c_v	konstanta významu prvku
c_z	konstanta poměru velikosti mapových značek v odvozené a podkladové mapě

1.1.2 Geometrická generalizace

Pokud má být mapa dobře čitelná a názorná, je nutno v její kresbě uplatnit generalizaci obrysů a tvarů. Geometrická generalizace provádí řízenou redukci obsahu mapy na základě geometrických parametrů jednotlivých prvků. Snaží se z mapy odstranit takové prvky, jejichž geometrické parametry nejsou významné v celkovém kontextu kartografického díla. Při tomto procesu jsou často používány i pomocné geometrické struktury jako Delaunayova triangulace, topologická kostra, Voronoiova teselace (viz [1]).

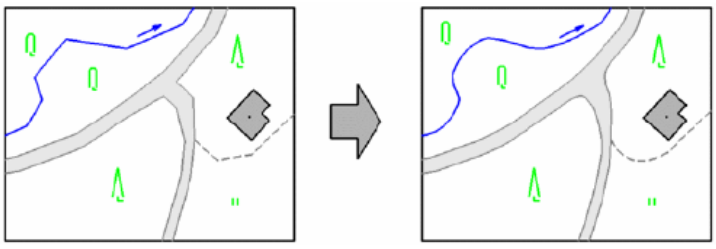
Mezi operace patřící ke geometrické generalizaci se řadí zjednodušení, vyhlazení, zlepšení, posun a pootočení. Zjednodušování se týká linií a ploch. Při této operaci je nutné zachovat koncové body, aby nebyly porušeny topologické vazby. Ukázka se nachází na obrázku 2.



Obrázek 2: Zjednodušení linií a ploch (převzato z [3])

U ploch by měla zůstat zachována původní výměra. V případě budov může dojít k nahrazení polygonu bodovou značkou nebo naopak k maximální generalizaci, kdy je polygon bez ohledu na původní tvar nahrazen obdélníkem.

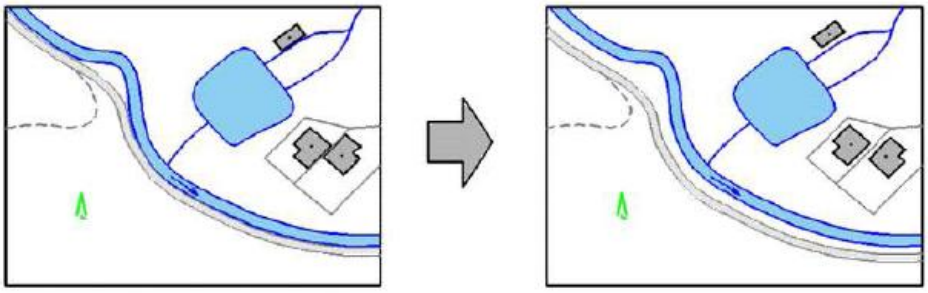
Vyhlazením se zvyšuje estetičnost kresby a provádí se hlavně ve dvou případech. Pokud byla původní kresba provedena lomenou čarou, nebo když je původní kresba spojnicí souřadnic bodů, dle měřického náčrtu.



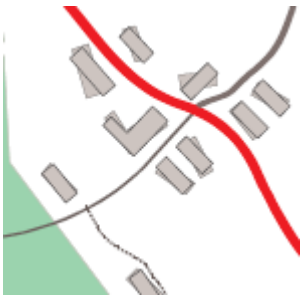
Obrázek 3: Vyhlazení (převzato z [3])

Vylepšování linií zkvalitňuje mapový obraz. Příkladem je zvýraznění meandrů vodního toku.

Posunutí a pootočení spadají do oblasti harmonizace map. Posunutí na mapě se používá pro lepší zviditelnění sousedících objektů. Objekty s menší prioritou se odsouvají. Během těchto operací dochází ke ztrátě přesnosti datového modelu.



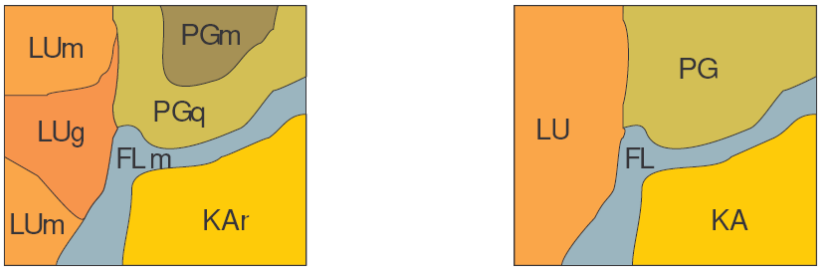
Obrázek 4: Posunutí (převzato z [14])



Obrázek 5: Pootočení (převzato z [16])

1.1.3 Generalizace reklasifikací

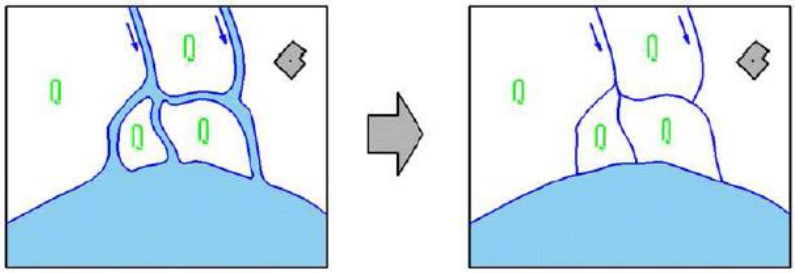
Změna klasifikace je specifickou modelovou generalizační technikou a uplatňuje se při transformaci jednoho datového modelu na druhý. V rámci klasifikace lze rozdělovat nebo slučovat jednotlivé skupiny elementů (viz [3]). Obrázek znázorňuje sloučení tříd na základě podobných hodnot.



Obrázek 6: Generalizace reklasifikací (převzato z [16])

1.1.4 Generalizace prostorovou redukcí

Generalizace je provedena změnou dimenze generalizovaného prvku. Existují tři základní varianty. Jednou z nich je změna oblasti na lomenou čáru. Používá se u plošných objektů výrazně protáhlého tvaru. Další je přechod oblasti na bod. Tato varianta je vhodná pro plošné objekty malých rozměrů, které není možné v daném měřítku zobrazovat samostatně. Bodový symbol je umisťován do těžiště oblasti. V posledním případě se z linie stává bod. Při této technice se často aplikují skeletonizační algoritmy.



Obrázek 7: Prostorová redukce (převzato z [3])

1.1.5 Generalizace ploch

Do této skupiny patří základní operace s plochami: sjednocení ploch (agregace), rozdělení plochy a zrušení plochy (eliminace). Metoda je používána u oblastí malých rozměrů, méně významných či oblastí nesplňujících nějakou podmínku. Při těchto operacích je následně nutno řešit hranice sousedních ploch.

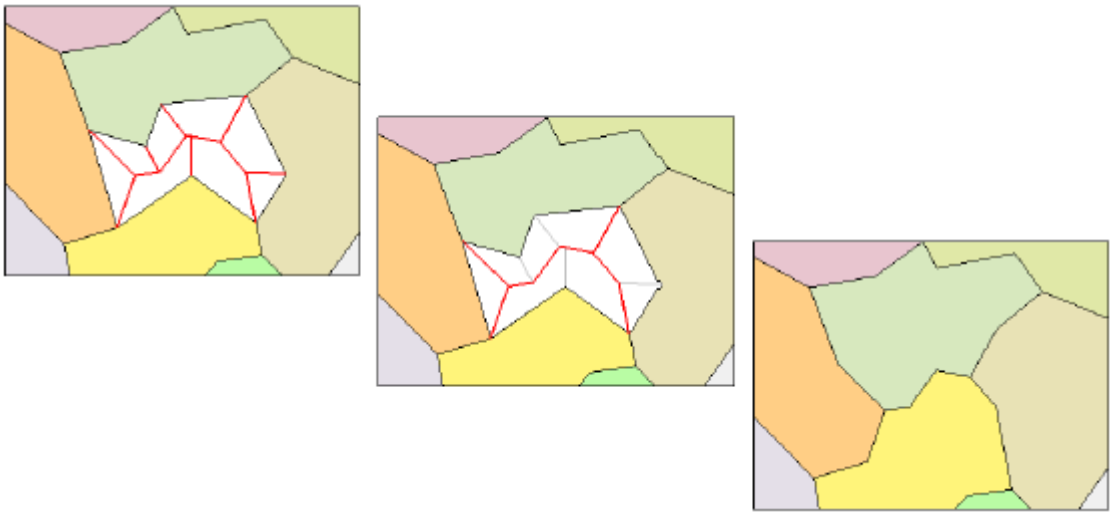
Generalizace agregací je používána v případě, že objekt již není možné v měřítku mapy zobrazovat samostatně. Při agregaci ploch do algoritmu vstupují nejméně dvě uzavřené oblasti, které spolu mohou, ale nemusí sousedit. Aby bylo možno generalizaci provést, nesmí se mezi nimi nacházet nějaký důležitý objekt. Plocha nově vzniklé oblasti by měla být podobná jako součet ploch do generalizace vstupujících. Kritéria výběru sousední oblasti mohou být jak geometrického, tak atributového charakteru. Slučovány bývají sousedící oblasti s podobným využitím či sousedící oblasti s nejdelší společnou hranicí. Ukázka agregace se nachází na obrázku 8.

Obrázek 8: Sloučení ploch (převzato z [16])

K rozdělení ploch dochází většinou v případě, kdy tvar objektu obsahuje zúžené části (viz obrázek 9), které by v požadovaném měřítku nemohly být reprezentovány jako plochy.

Obrázek 9: Rozdělení ploch (převzato z [16])

Při eliminaci dochází k rozdělení plochy rušené oblasti mezi sousední oblasti, které s ní mají alespoň jednu společnou hranu. Otázkou je, jakým způsobem vést dělící hranice uvnitř generalizované oblasti. Řešení nabízejí skeletonizační algoritmy, kdy se nad generalizovanou oblastí vygeneruje skeleton, a dělící hranice představují hrany skeletonu. Tento algoritmus, ale není univerzálně použitelný. Ne vždy lze oblast rozdělit beze zbytku. Popsaný postup také není možné uplatnit ve všech typech mapových děl (viz [1]). Použití skeletonizace je zřejmé z obrázku 10.



Obrázek 10: Generalizace skeletonizací (převzato z [1])

1.1.6 Generalizace atributů

Nejčastější generalizační technikou aplikovanou na atributová data je výběr konkrétních atributů, které zůstanou zachovány, na základě vymezení zájmové oblasti nebo na základě uživatelem stanovených omezujících podmínek.

1.2 Rešerše literatury

Oblast vojenské kartografie trápily v 60. letech minulého století příliš dlouhé lhůty tvorby nových nebo obnovovaných map. To dalo podnět ke zkoumání zcela nových technologií mapové tvorby, které byly založeny na matematických principech. K dané problematice se vztahuje disertační práce *Matematicko-logické modelování procesu generalizace v kartografii* od Erharta Srnky [4]. Jeho snahou bylo vytvoření vhodných a obecně fungujících modelů kartografické generalizace, které by odpovídaly vědeckým kritériím a současně byly únosné z hlediska programového zpracování a praktické realizace. Objektivnost kartografické generalizace měl zajistit otevřený matematický systém, vyhovující všem možným požadavkům na specifikaci řešení. Vycházel z toho, že výskyt každého prvku je možno vyjádřit několika kvantitativními ukazateli (statistickými znaky) hustoty ve vztažné územní jednotce (počet, délka, plocha prvků aj.). Hlavním výsledkem práce je matematicko-logický model generalizace, který je definován soustavou mocninných funkcí pro prvky čárového typu, plošného prvku a bodového typu. Mocninné parametry určují variabilitu výstupních údajů v dílčích modelech generalizace v závislosti na příslušných vstupních charakteristikách. Teorie matematicko-logického modelování procesu kartografické generalizace byla úspěšně použita při tvorbě základních československých kartografických děl. Patří sem Československý vojenský atlas (Naše vojsko, Praha, 1965), Vojenský zeměpisný atlas (MNO, Praha, 1975) a československé vojenské topografické mapy.

Na práci E. Srnky navazují další práce. Jednou z nich je disertace *Automatizace některých pracovních postupů při využití matematických modelů generalizace* od Pavla Kovaříka [5], která se zabývá kartografickým výběrem a aproximací kartografických křivek. Autor navrhuje metodu kartografického výběru, která bere v úvahu více charakteristik mapových prvků. Více vstupních proměnných přispívá k objektivnosti kartografického výběru. Pro aproximaci kartografických křivek vytváří metodu používající kruhové oblouky, které na sebe tečně navazují. Tím se docílí úspory paměti při ukládání rozsáhlých datových souborů vznikajících digitálním sejmutím z mapy.

Josef Janošec navazuje prací *Příspěvek k systémovému řešení metod automatizované kartografické generalizace* [6]. Tato práce přináší systémové řešení metody zevšeobecňování tvarů a automatizovanou tvorbu topografických map databankovými technologiemi. Základem databankových technologií je segmentace

procesu na pořizování dat; organizaci dat; banku kartografických dat; výběr, generalizaci a přípravu grafického výstupu dat; grafický výstup; kartoreprodukční proces zpracování grafického výstupu.

Dle současných příspěvků k tématu generalizace lze usuzovat, že problematika generalizace se dělí na dva hlavní proudy. Jedna skupina autorů se zabývá generalizací stavebních objektů, při níž je důležité zachovat rovnoběžnosti stran a pravoúhlost objektu. Druhá skupina se zajímá převážně o liniové prvky a uzavřené oblasti obecného tvaru.

Mezi publikace, které řeší generalizaci zástavby, patří disertační práce *Zur Automation der Generalisierung topographischer Karten mit besonderer Berüchsichtigung grossmassstäbiger Gebäudedarstellunge* od Wilfrieda Staufenbiela z Univesität Hannover (1973). Uvádí zde návrh algoritmu na generalizaci zástavby z velkého do středního měřítka. Využívá Deutche Grundkarte v měřítku 1:5000, kterou převádí do měřítka 1:25000. Principy navrženého algoritmu jsou využívány v některých GIS software.

Na dílo W. Staufenbiela navazuje svojí diplomovou prací Radim Štampach. Ve své práci *Automatizovaná kartografická generalizace stavebních objektů z katastrálních map do map středních měřítek* [7] implementoval původně navržený algoritmus do prostředí ArcView 3.2.

Monika Sester se ve svém článku *Generalization based on least squares adjustment* [8] zabývá využitím metody nejmenších čtverců při generalizaci sídel. V části navrženého algoritmu opět využívá principy Staufenbielova algoritmu.

Jedním z nejnovějších příspěvků je článek Tomáše Bayera *Kartografická generalizace stavebních objektů prostřednictvím 2D množinových operací* [9]. Popsaný postup kartografické generalizace lze shrnout do několika kroků. Nejprve je nalezen obdélník v obecné poloze, který je opsaný množině vrcholů původního objektu tak, aby byl jeho obsah minimální. Tento obdélník je upraven, aby jeho obsah odpovídal výměře původního objektu, ale poměr stran a natočení zůstal zachován. Hledá se tedy stejnolehlý obdélník. Následuje aplikace množinové operace sjednocení takto vzniklých obdélníků. Při tomto postupu nevznikají nežádoucí průsečíky a jsou zachovány topologické vztahy mezi prvky.

Odlišný přístup k procesu generalizace představuje *Cartographic generalization using primitives and constraints* [10] od autorů Claus Brenner a Monika Sester. Navrhují použití geometrických primitiv, kontejnerů a omezujících podmínek jako prostředků kartografické generalizace. Geometrická primitiva představují např. body, linie nebo kružnice. Kontejnery jsou kolekce primitiv s jistým společným chováním. Omezující podmínky mohou být logického charakteru jako incidence, kolmost, rovnoběžnost nebo metrické jako vzdálenost, úhel, poloměr… Řešení se potom pro dané omezující podmínky hledá pomocí soustav algebraických rovnic.

Využití teorie grafů popisuje Dalibor Moravec ve své práci *Geoinformatické modelování s topologickým generalizačním algoritmem* [11]. Generalizaci představuje jako kartografické a geoinformatické modelování, které obvykle vychází z již existujícího kartografického modelu. Základem jsou topologické relace, jež slouží k podchycení prostorových vztahů mezi objekty. Vyjadřují, zda spolu objekty incidují, sousedí nebo se obsahují. Topologický generalizační algoritmus je vhodný pro slučování objektů plošného typu a jejich náhradu symbolickým znázorněním. Aby bylo možno modelovat výchozí systém oblastí, přiřadí se mu graf. Vrcholy grafu představují oblasti a hranami jsou interpretovány topologické relace sousedství mezi oblastmi. Vrcholům je přidělena kardinalita. Kardinalita může být stanovena topologicky, to znamená stupni vrcholů, nebo z tematických dat. Podstata algoritmu spočívá ve vyhledávání hran grafu pro každý vrchol a testování hodnoty kardinality sousedících vrcholů. Jestliže jsou prohledány a zpracovány výchozí množiny vrcholů a hran, vzniknou množiny vrcholů a hran generalizovaného grafu, v němž jsou některé hrany a vrcholy vypuštěny. Nově vzniklé množiny vrcholů a hran jsou výchozími množinami pro další krok generalizace. V každém kroku lze integrovaným vrcholům ponechat původní kardinalitu nebo přiřadit kardinalitu danou sumou kardinalit sousedících vypouštěných vrcholů a původní kardinality vrcholu. Algoritmus lze modifikovat pro konkrétní situace nastavením topologických, metrických nebo tematických omezení a různými hladinami účinnosti. Modelování nad systémem oblastí bylo teoreticky odvozeno až po vývoj programů v jazyce C a nasazení geoinformačního systému ARC/INFO pro běžné manipulace s daty a vizualizace.

Christopher Gold a David Thibult v článku *Map generalization by skeleton retraction* [12] ukazují použití skeletonu při kartografické generalizaci vrstevnic, polygonů, říční sítě a skenovaných map. Doporučují tuto metodu, protože zachovává topologii na rozdíl od generalizačních algoritmů jako je Douglas-Peuckerův algoritmus.

Aplikováním běžných generalizačních algoritmů na blízké objekty může totiž docházet k překrývání a vzniku nechtěných průsečíků mezi generalizovanými prvky.

Ucelený přehled o generalizačních algoritmech, které se týkají liniových prvků a uzavřených oblastí se nachází v publikaci *Algoritmy v digitální kartografii* [1], jejímž autorem je T. Bayer.

2 Generalizační algoritmy

Při generalizaci pomocí algoritmů může docházet k některým negativním efektům, na které si je zejména při kartografické generalizaci potřeba dávat pozor. Mezi nechtěné situace patří vznik nežádoucích tvarů prvků, vznik nežádoucích průsečíků generalizovaných prvků či ztráta souvislostí mezi objekty. Následující kapitoly čerpají z [1].

2.1 Algoritmy pro zjednodušení lomených čar

Algoritmy lze rozdělit do několika kategorií. Jedná se o algoritmy nezávislé na tvaru prvku, lokální algoritmy, rozšířené lokální algoritmy a globální algoritmy. Jak již vyplývá z pojmenování skupin, nejvhodnější pro kartografickou generalizaci jsou globální algoritmy, protože v každém kroku generalizace posuzují geometrické parametry testovaného vrcholu vzhledem ke všem vrcholům lomené čáry a zohledňují tak skutečný tvar prvku.

2.1.1 Algoritmy nezávislé na tvaru prvku

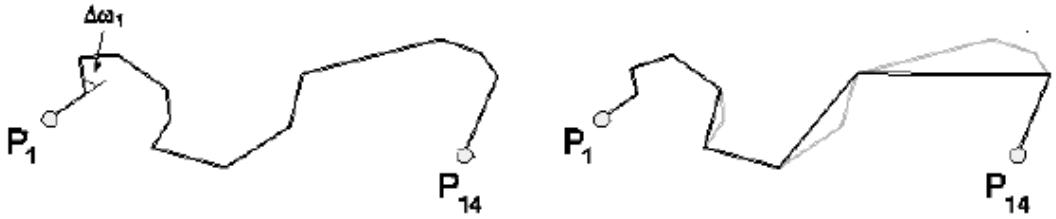
Algoritmy nezávislé na tvaru prvku patří k nejjednodušším. Do této kategorie spadá vynechání n-tého nebo náhodného vrcholu lomené čáry. Tímto způsobem mohou být odstraněny tvarově významné vrcholy. Algoritmy lze použít pouze k redukci nadměrného množství dat.

2.1.2 Lokální algoritmy

Lokální algoritmy odstraňují takové vrcholy, které nesplňují vzhledem k sousedním vrcholům nějakou geometrickou podmínku.

Podmínkou může být délkové kritérium. Při generalizaci jsou potom vynechávány vrcholy, které mají menší vzdálenost od předchozího vrcholu než je zvolené kritérium.

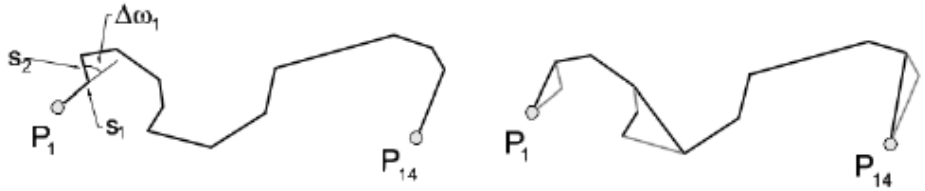
Jiný algoritmus rozhoduje o odstranění vrcholu na základě velikosti změny vrcholového úhlu. Odstraněn je vrcholový bod úhlu, jehož rameno má podobný směr jako rameno předchozí (viz obrázek 11).



Obrázek 11: Generalizace testováním změny velikosti vrcholového úhlu. (převzato z [1])

Dále do této kategorie patří algoritmy, které zohledňují více podmínek najednou. Příkladem je algoritmus, který testuje vzdálenost vrcholu od strany a velikost změny úhlu. Při generalizaci jsou posuzovány trojice za sebou následujících vrcholů P_i, P_{i+1}, P_{i+2}. Vynechávají se vrcholy P_{i+2}, které mají od strany P_i, P_{i+1} vzdálenost menší než zadané délkové kritérium nebo je změna úhlu mezi oběma prvky menší než stanovená hodnota.

Jenksův algoritmus počítá vzdálenosti s_i, s_{i+1} mezi třemi po sobě jdoucími body. Pokud je alespoň jedna z vypočtených vzdáleností menší než délkové kritérium nebo změna úhlu menší než úhlové kritérium, je bod P_{i+1} odebrán. Ukázku tohoto algoritmu znázorňuje obrázek 12.



Obrázek 12: Generalizace lomené čáry Jenksovým algoritmem. (převzato z [1])

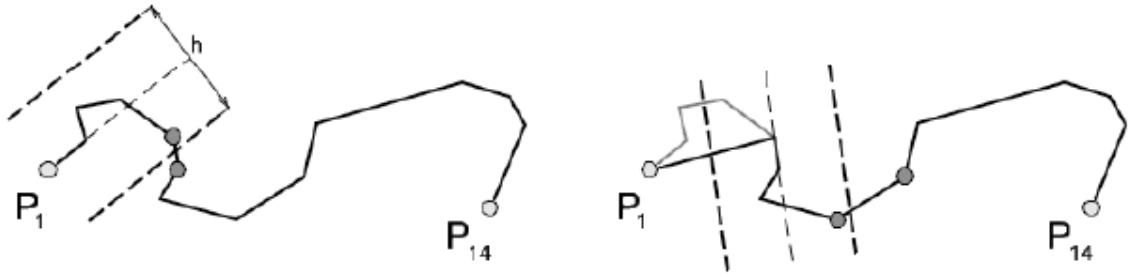
Do skupiny lokálních algoritmů spadá i posuzování velikosti kolmé vzdálenosti. Vygeneruje se spojnice mezi body P_i, P_{i+2} a počítá se vzdálenost bodu P_{i+1} od tohoto segmentu. Je-li vzdálenost menší než stanovené kritérium, je bod vypuštěn.

2.1.3 Rozšířené lokální algoritmy

Rozšířené lokální algoritmy neposuzují vrchol lomené čáry pouze vzhledem ke svým sousedním vrcholům, ale pracují s množinou bodů. Snaží se již zohlednit skutečný tvar prvků.

Reumann-Witkamův algoritmus (1974)

Do této kategorie patří Reumann-Witkamův algoritmus. Jeho základem je opakovaná tvorba koridoru o šířce *h* rovnoběžně se spojnicí dvou po sobě jdoucích bodů lomené čáry tak, že spojnice představuje jeho osu. Nejprve je koridor umístěn rovnoběžně s první stranou lomené čáry. Najde se průsečík s nejbližším segmentem lomené čáry. Tento segment bude tvořen vrcholy P_i, P_{i+1}. Všechny vrcholy ležící v koridoru s výjimkou prvního a posledního se odstraní. Nový koridor je tvořen rovnoběžně s hranou, která byla protnuta původním koridorem. Opět se hledá průsečík s nejbližším segmentem lomené čáry směrem vpřed a opakuje se předchozí postup. Princip je zřejmý z obrázku 13.

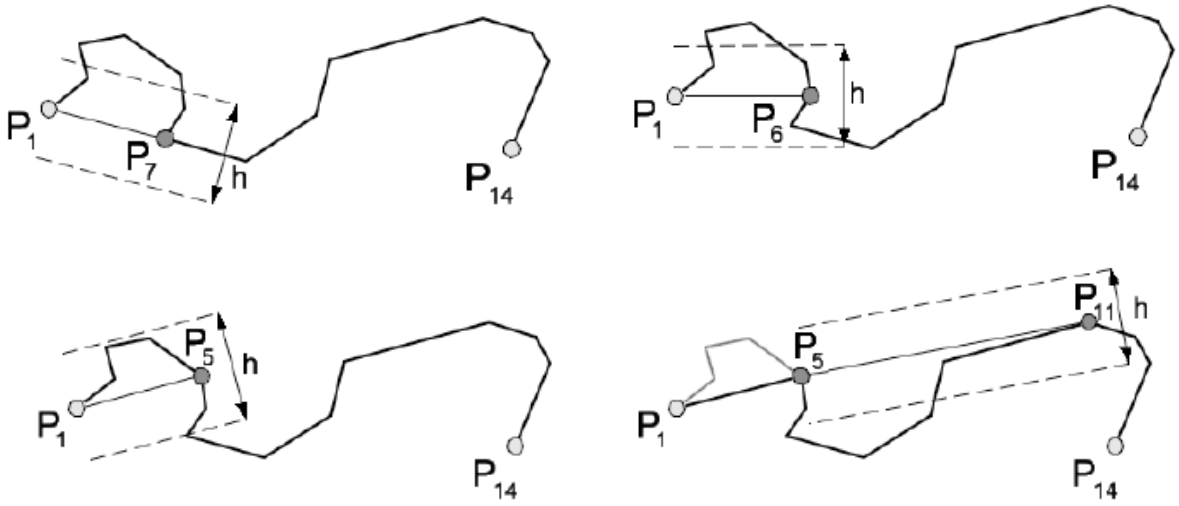


Obrázek 13: Princip Reumann-Wikamova algoritmu. (převzato z [1])

Langův algoritmus (1969)

Langův algoritmus také používá koridor o šířce *h*, ale tentokrát není tvořen rovnoběžně se stranami lomené čáry, ale rovnoběžně se spojnicí bodů P_i, P_{i+n}, kde *n* udává počet bodů, se kterými se v jednom kroku pracuje. Pokud alespoň jeden z bodů $<P_{i+1}, P_{i+n-1}>$ padne vně koridoru, zůstanou všechny body beze změny. Následuje vytvoření koridoru rovnoběžně se spojnicí bodů P_i, P_{i+n-1} a znovu se zjišťuje, zda alespoň jeden z bodů $<P_{i+1}, P_{i+n-2}>$ padne vně koridoru. Takto se postupuje, dokud nenastane situace, že žádný bod se nenachází vně koridoru nebo dokud nevyčerpáme všechny možnosti

vytvoření spojnice mezi prvním bodem a ostatními body z množiny bodů, které jsou k dispozici pro daný krok. Pokud žádný bod nepadne vně koridoru, jsou všechny body uvnitř koridoru kromě prvního a posledního odstraněny. Poslední bod se stává počátečním bodem pro vytvoření nového koridoru v dalším kroku algoritmu. Postup naznačuje obrázek 14.

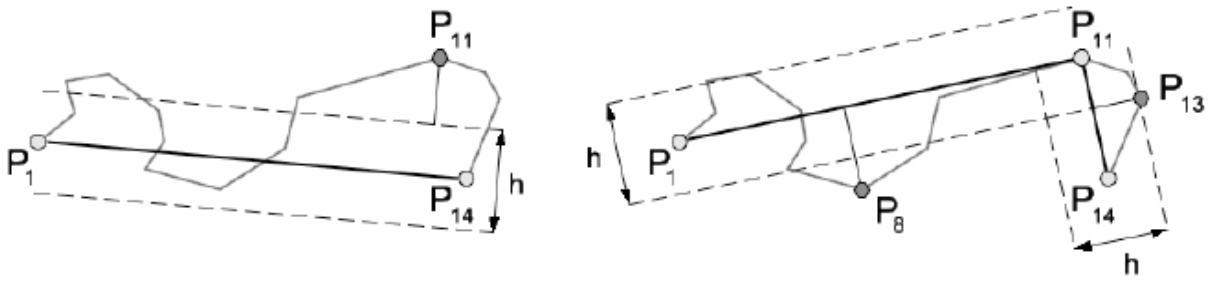


Obrázek 14: Langův algoritmus. Zvoleno n = 6. (převzato z [1])

2.1.4 Globální algoritmy

Douglas-Peuckerův algoritmus (1973)

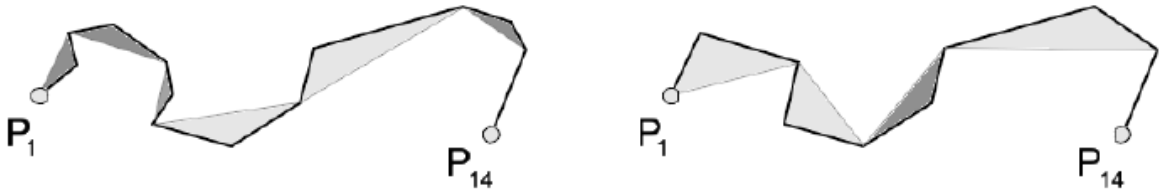
Asi nejznámějším algoritmem, který se řadí do skupiny globálních algoritmů je Douglas-Peuckerův algoritmus. Tento algoritmus také pracuje s koridorem, ale postupuje opačně než výše zmíněné algoritmy. V prvním kroku je lomená čára nahrazena úsečkou, která spojuje první a poslední bod. Potom se hledá nejvzdálenější bod od této spojnice, a pokud je vně koridoru vytvořeného kolem spojnice, je přidán jako nový vrchol čáry. V dalším kroku je lomená čára tvořena dvěma úsečkami (viz obrázek 15). Na každou z nich je tento postup rekurzivně opakován. Algoritmus končí, když se žádný bod nenachází vně koridoru. Douglas-Peuckerův algoritmus je jedním z nejčastěji používaných generalizačních algoritmů v digitální kartografii.



Obrázek 15: Douglas-Peuckerův algoritmus. (převzato z [1])

Whyattův algoritmus (1993)

Whyattův algoritmus určuje obsahy trojúhelníků, které jsou tvořeny vždy ze tří za sebou následujících vrcholů. Pokud je obsah menší než stanovená hodnota, prostřední vrchol je odstraněn a trojúhelníky přegenerovány. Neexistuje-li žádná plocha menší než minimální kritérium, výpočet končí. Princip algoritmu naznačuje obrázek 16.



Obrázek 16: Whyattův algoritmus. (převzato z [1])

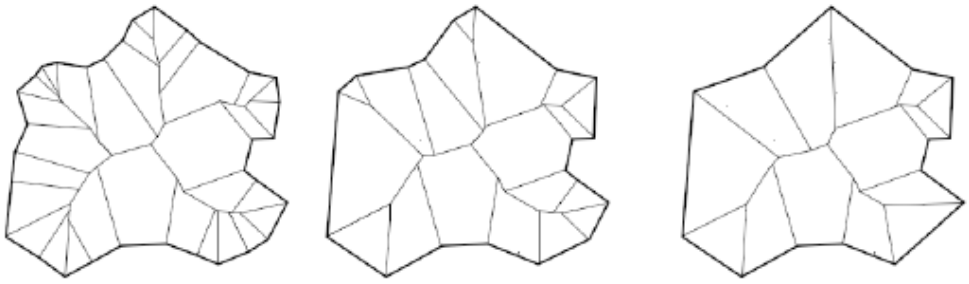
2.2 Algoritmy pro generalizaci uzavřených oblastí

Generalizaci uzavřených oblastí lze provést třemi způsoby. Dekompozicí na lomené čáry, pomocí modifikovaných algoritmů pro lomené čáry nebo s využitím skeletonizačních algoritmů (viz [1]).

Metoda rozdělení uzavřené oblasti na několik lomených čar se používá, pokud oblast sousedí alespoň s jednou další oblastí. Uzavřená oblast je převedena na posloupnost lomených čar, které na sebe navazují v uzlových bodech. Tyto body musí zůstat zachovány, aby nedošlo ke ztrátě topologických vazeb. Každá takto vzniklá lomená čára je generalizována samostatně a v průběhu generalizace se provádějí testy, zda nevznikají průsečíky generalizované hrany s hranami ostatních prvků.

Algoritmy, které původně byly vytvořeny pro generalizaci lomených čar, se dají upravit tak, aby je bylo možno použít i pro uzavřené oblasti. Musí se zachovat poloha uzlových bodů. Nejčastěji je využíván modifikovaný Douglas-Peuckerův algoritmus. Nejprve se musí nalézt počáteční aproximace. Pokud oblast nesousedí s jinou oblastí, neexistuje žádný uzlový bod a první aproximací je čtyřúhelník, jehož strany spojují body min-max boxu vygenerovaného nad oblastí. V případě, že má oblast jeden dotykový bod s jinou oblastí, je výchozí spojnicí úsečka spojující uzlový bod s bodem generalizované oblasti, který je od uzlu nejvzdálenější. Když generalizovaná oblast sousedí s jednou oblastí, existují dva uzlové body a do algoritmu vstupuje jejich spojnice. Pokud oblast sousedí s více než jednou oblastí, je první aproximací uzavřená oblast, která spojuje uzlové body generalizované oblasti. Na hrany se poté rekurzivně aplikuje Douglas-Peuckerův algoritmus.

Generalizace pomocí skeletonu je založena na myšlence, že tvarově jednoduché objekty mají jednoduchou topologickou kostru. Nejprve se vygeneruje topologická kostra generalizované oblasti a poté je na ni aplikován nějaký algoritmus na zjednodušení lomené čáry. Odstraněním vrcholu skeletonu dochází ke zjednodušení tvaru oblasti. Princip je zřejmý z obrázku 17.



Obrázek 17: Generalizace postupnou redukcí skeletonu (převzato z [1])

3 Generalizační algoritmy jako funkce prostorové databáze PostGIS

Algoritmy vhodné pro kartografickou generalizaci jsou v rámci této diplomové práce implementovány v jazyce Java. V prostředí PostGIS je poté lze spouštět jako funkce, protože podporuje procedurální jazyk PL/Java.

3.1 PostGIS

Open source produkt PostGIS je vyvíjen společností Refractions Research Inc. jako výzkumný projekt, který se zabývá technologií prostorové databáze. PostGIS rozšiřuje relační databázový systém PostgreSQL o možnost práce s prostorovými objekty. PostGIS splňuje OpenGIS standardy, a může tedy sloužit jako datový server pro různé geografické informační systémy.[13]

Přináší nové datové typy, operátory, funkce a tabulky, které poskytují prostor pro metadata. Nový datový typ *geometry* je universální pro všechny prostorové objekty. Linie, bod, polygon a další prostorové objekty definované dle OGC mohou být v databázi uloženy jako typ *geometry*. Mezi nové operátory patří průnik (&&) a kompletně obsažen (@). Pro snadnější práci s geografickými objekty existuje řada funkcí. Funkce se týkají vytváření prostorových tabulek, vztahů mezi jednotlivými objekty, editace dat a jejich výstupů. Novými tabulkami jsou `SPATIAL_REF_SYS` a `GEOMETRY_COLUMNS`. První z tabulek obsahuje informace o souřadnicových systémech a kartografických zobrazeních. Druhá určuje geometrii, dimenzi a použitý souřadnicový systém.

3.1.1 Definice geometrických objektů

OGC určuje dva způsoby, jak lze vyjádřit prostorové objekty. Obě formy záznamu Well-Known Text (WKT) a Well-Known Binary (WKB) obsahují informaci o typu objektu a souřadnice, které určují jeho tvar. Také je požadován identifikátor SRID, který dává souřadnicový systém.

Základními prostorovými objekty jsou GEOMETRYCOLLECTION, POLYGON, MULTIPOLYGON, LINESTRING, MULTILINESTRING, MULTIPOINT, POINT.

Příklady zadávání prostorových objektů z textu (WKT) :

```
POINT (0 0)

MULTIPOINT (0 0, 1 2)

LINESTRING (0 0, 1 1, 1 2)

MULTILINESTRING ((0 0, 1 1,1 2),(2 3, 3 2, 5 4))

POLYGON ((0  0, 4 0, 4 4, 0 4, 0 0),(1 1, 2 1, 2 2, 1 2, 1 1))

MULTIPOLYGON (((0  0, 4 0, 4 4, 0 0),(2 1, 3 1, 3 2, 1 1)),((-1 -1,
-1 -2, -2 -2, -1 -1)))

GEOMETRYCOLLECTION (POINT (2 3), LINESTRING (2 3, 3 4))
```

3.1.2 Vytváření vlastních funkcí

Databázový systém PostGIS nad PostgreSQL nabízí velké množství funkcí pro práci s geodaty, ale někdy je potřeba vytvořit si své vlastní funkce. K tomu lze využít některý z procedurálních jazyků (PL/pgsql, PL/perl, PL/python, PL/tcl, PL/Java). Funkce se vytvářejí pomocí příkazu CREATE FUNCTION. Zjednodušená syntaxe je následující:

```
CREATE FUNCTION  jméno_funkce ([typ argumentu[,]])
RETURNS  typ_návratové_hodnoty
AS  'definice_funkce'
LANGUAGE   'jméno_jazyka';
```

Definice funkce se liší dle použitého jazyka. Může obsahovat zdrojový kód nebo SQL dotaz, pokud se jedná o funkci v jazyce SQL. Z níže uvedeného příkladu je zřejmé, že definicí funkce je také relativní cesta k nějaké již napsané funkci.

```
CREATE FUNCTION generalizacedp(geometry, double precision)
RETURNS  geometry
AS 'cz.hrncirova.diplomka.DP.generalizace'
LANGUAGE 'java';
```

3.1.3 PL/Java

Procedurální jazyk Java lze instalovat současně s databázovým systémem PostgreSQL a nadstavbou PostGIS nebo ho přidat manuálně. V případě manuální instalace je nutno dle návodu editovat konfigurační soubor postgresql.conf a nastavit systémovou proměnnou JRE_HOME. Tato zkušenost platí pro operační systém Windows XP. Z pročtených manuálů však vyplývá, že instalace na OS Linux odpovídá manuální instalaci ve Windows.

Po instalaci PL/Java je do databáze přidáno schéma `sqlj`. Nové schéma obsahuje čtyři tabulky `classpath_entry`, `jar_entry`, `jar_repository` a `typemap_entry`. Dále jsou k dispozici obslužné funkce, pomocí kterých jsou tyto tabulky naplňovány či editovány. Pokud je potřeba spustit nějaké vlastní funkce napsané v jazyce Java, musí se nastavit umístění JAR souboru, který obsahuje všechny potřebné třídy. K tomu slouží funkce `install_jar(<jar_URL>, <jar_name>, <deploy>)`. První argument specifikuje umístění JAR souboru, druhý je pouze název tohoto souboru bez přípony a poslední argument nabývá hodnoty `true` nebo `false` v závislosti na tom, zda chceme použít „ deployment descriptor“ nebo ne. Veškeré nainstalované JAR soubory se ukládají do tabulky `jar_repository` a relativní cesty k třídám z těchto souborů

	entryid [PK] serial	entryname character varying(200)	jarid integer	entryimage bytea
53	44052	postgis/Point.class	154	<Binary data>
54	44053	postgis/MultiLineString.class	154	<Binary data>
55	44054	postgis/LinearRing.class	154	<Binary data>
56	44055	cz/hrncirova/diplomka/Whyatt.class	154	<Binary data>
57	44056	cz/hrncirova/diplomka/Algoritmy.class	154	<Binary data>
58	44057	cz/hrncirova/diplomka/Reumann.class	154	<Binary data>
59	44058	cz/hrncirova/diplomka/DP.class	154	<Binary data>

Obrázek 18: Část tabulky `jar_entry`

do `jar_entry`. Samozřejmě existují také funkce na změnu nebo mazání nainstalovaných JAR souborů. Tabulka `classpath_entry` obsahuje aktuálně používané JAR soubory, jež se definují funkcí `set_classpath(<schema>, <classpath>)`. „ Schema“ udává název schématu, pro které se JAR soubory nastavují a druhý argument funkce obsahuje jeden nebo více názvů JAR oddělených dvojtečkou. Aby bylo možno v PL/Java používat SQL datové typy, existuje funkce `add_type_mapping(<sql type>, <java class>)` a informace o tomto propojení se ukládají do tabulky `typemap_entry`.

	jarid [PK] serial	jarname character va	jarorigin character varying(500)	jarowner name	jarmanifest text	deploymentd integer
1	50	deploy	file:C:\Program Files\Postgr	postgres		
2	51	examples	file:C:\Program Files\Postgr	postgres	Manifest-Version: 1.0	31
3	69	vy	file:C:\Program Files\Postgr	postgres	Manifest-Version: 1.0	
4	154	generalizace	file:C:\Program Files\Postgr	postgres	Manifest-Version: 1.0	

Obrázek 19: Ukázka tabulky `jar_repository`

3.1.4 Funkce `Simplify(geometry,double precision)`

V prostředí PostGIS již existuje jedna funkce, která slouží ke generalizaci dat. Tato funkce je napsána v jazyce C, využívá Douglas-Peuckerův algoritmus, ale neřeší nechtěné průsečíky, které během generalizace mohou vzniknout. Prvním argumentem je geometrický objekt, jenž má být generalizován. Druhý udává šířku koridoru. Funkce umí zpracovat (multi) linie a (multi) polygony. Pokud je na vstupu jiný typ geometrie, žádná generalizace neproběhne a jsou vrácena původní data.

4 Navržená generalizační funkce

Pro geometrickou generalizaci byly vybrány čtyři generalizační algoritmy. Jedná se o Reumann-Witkamův algoritmus, Langův algoritmus, Douglas-Peuckerův algoritmus a Whyattův algoritmus. Řešení spočívá v tvorbě funkce, kterou lze spouštět z prostředí PostGIS. Jedním ze vstupních parametrů funkce je výběr generalizačního algoritmu.

Zdrojový kód je napsán v jazyce Java v prostředí Eclipse SDK s využitím knihovny *org.postgis*, která obsahuje třídu `Geometry` a definuje všechny potřebné prostorové objekty. Stačilo tedy vytvořit pouze funkce, které s těmito objekty pracují. Jediným novým objektem je třída `Vektor`.

4.1 Definice funkce

generalizace(geometry[], charakter varying, double precision, integer)

```
CREATE FUNCTION generalizace(geometry [], charakter
                varying, double precision, integer)
RETURNS  SETOF geometry
AS 'cz.hrncirova.diplomka.Rozdeleni.generalizace'
LANGUAGE 'java';
```

První parametr funkce představuje vstupní data. Druhým je určen generalizační algoritmus. Hodnoty textového řetězce pro jednotlivé algoritmy jsou uvedeny v tabulce 1.

Hodnota	Algoritmus
dp	Douglas-Peuckerův
lang	Langův
whyatt	Whyattův
reumann	Reumann-Witkamův

Tabulka 1: Argument funkce určující generalizační algoritmus

Další argument udává šířku koridoru nebo minimální kritérium obsahu trojúhelníka pro Whyattův algoritmus. Poslední parametr funkce má význam pouze pro Langův algoritmus, kde určuje počet bodů, se kterými se pracuje v jednom kroku generalizace. Při práci s jiným algoritmem lze doplnit 0 nebo nechat libovolné číslo. Tato hodnota do výpočtu zahrnuta nebude.

Klíčové slovo `SETOF` u návratového typu v definici funkce znamená, že funkce nebude vracet pouze jednu hodnotu, ale bude se jednat o sadu hodnot.

Z prostředí PostGIS může být funkce volána pomocí následujícího SQL příkazu:

```
CREATE TABLE silnicel AS SELECT generalizace(SELECT ARRAY
(SELECT the_geom FROM silnice),'lang', 30, 5);

ALTER TABLE silnicel ADD COLUMN id serial PRIMARY KEY;
```

Vstupní data do funkce `generalizace(the_geom[], 'lang', 30, 5)` z výše uvedeného příkladu pochází z existující tabulky `silnice`, která obsahuje sloupec `the_geom` typu `geometry`. Generalizovaná data se uloží do nově vytvořené tabulky `silnicel`.

4.2 Popis funkce

Generalizační funkce obsahuje pět tříd. Jmenovitě se jedná o třídy `Rozdeleni`, `Algoritmy`, `GeneralizaceLinie`, `GeneralizacePlochy` a `Vektor`. Zdrojový kód všech níže popsaných tříd je přiložen na konci diplomové práce.

4.2.1 Třída `Rozdeleni`

Tato třída obsahuje jedinou funkci `generalizace(String [], String, double, int)`, která načítá vstupní data a na základě testování volí, zda půjde o generalizaci liniových (`MULTILINESTRING`, `LINESTRING`) nebo plošných objektů (`MULTIPOLYGON`, `POLYGON`). Pokud by na vstupu byla mapová vrstva bodů či prostorový objekt typu `GEOMETRYCOLLECTION`, žádná generalizace se neprovede

a vrácena budou nezměněná data. První parametr funkce představuje celý sloupec tabulky z databáze v PostGIS. Jedná se o pole hodnot typu `geometry`. Funkce není volána na každý řádek tabulky zvlášť, protože by nemohly být počítány průsečíky mezi jednotlivými objekty. Návratová hodnota je typu `Iterator`, aby mohl být opět vrácen celý sloupec tabulky. Argumenty funkce se shodují s definicí funkce v PostGIS (viz kapitola 4.1).

Na generalizaci liniových objektů je volána funkce `linie (ArrayList<LineString>, ArrayList<Point>, String, double, int)` ze třídy `GeneralizaceLinie` a pro plošné objekty funkce `plochy (ArrayList<Polygon>, ArrayList<LinearRing>, ArrayList<Point>, String, double, int)` z třídy `GeneralizacePlochy`.

4.2.2 Třída `GeneralizaceLinie`

Funkce `linie (ArrayList<LineString>, ArrayList<Point>, String, double, int)` připraví vstupní data pro použití generalizačním algoritmem. Prvním argumentem jsou vstupní data. Další parametr předává všechny body vstupních linií. Na základě tohoto seznamu jsou vyhledány uzlové body. Mohlo by se zdát, že tento argument je zbytečný, když do funkce vstupují data jako objekty typu `LineString`. Je ale zbytečné procházet znovu všechny linie. Body jsou do seznamu uloženy již při načítání dat z databáze. Poslední tři argumenty jsou shodné s původně volanou funkcí.

Nejprve jsou vyhledány uzlové body, které jsou důležité pro zachování topologie generalizovaných dat. Uzlovým bodem je v případě liniových dat myšlen bod, ve kterém dochází k větvení. Problém řeší cyklus, který prochází vstupní hodnoty a určuje kolikrát je v nich který bod obsažen. Každý bod vyskytující se více než jednou je označen jako uzel. Křivka se rozpadne na posloupnost úseků, které jsou tvořeny částmi linie mezi uzly nebo mezi uzlem a počátečním či koncovým bodem. Na tyto úseky je volán generalizační algoritmus. Následuje spojení bodů z jednotlivých úseků zpět do `LineString`.

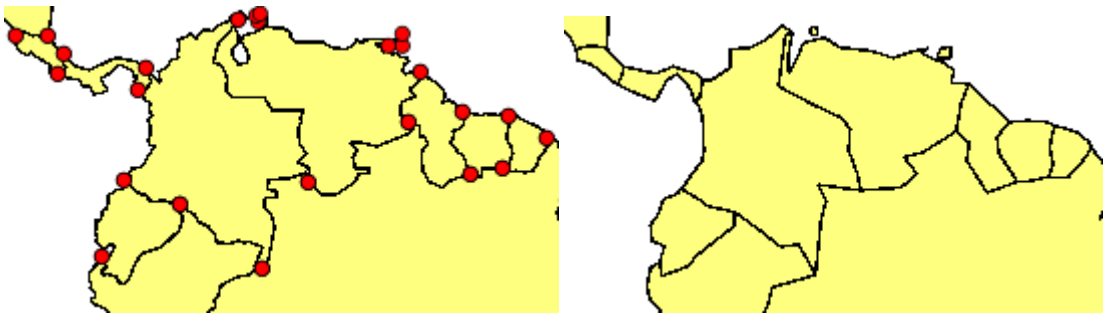
Hledání uzlů celý proces značně zpomaluje, ale po zkušenostech s testovanými daty, bylo zjištěno, že vždy nejsou dodrženy zásady správné vektorizace a linie nejsou

v každém uzlu ukončeny. Příkladem je silniční síť. Má být jedna linie úsek silnice mezi dvěma nejbližšími křižovatkami nebo celá část silnice s označením první třídy?

4.2.3 Třída `GeneralizacePlochy`

Úloha třídy `GeneralizacePlochy` je stejná jako u třídy `GeneralizaceLinie`. Základní funkce `plochy(ArrayList<Polygon>, ArrayList<LinearRing>, ArrayList<Point>, String, double, int)` má od předchozí funkce rozdílné pouze první dva argumenty. Také se jedná o vstupní data, ale protože polygony jsou komplikovanější než linie, jsou zde parametry dva. Důvodem je opět ušetření dalších průchodů vstupních dat.

Hledání uzlových bodů je v případě ploch odlišné. Rozdíl je v tom, že každý bod vyskytující se ve vstupních datech více než jednou nemusí být uzlem. Uzel je určitě bod existující více než dvakrát. Pokud se daný bod vyskytuje v datech právě dvakrát, leží na společné hranici dvou polygonů. Takový bod může být uzlem, pokud se před ním nebo za ním nachází bod, jenž byl nalezen pouze jednou. Tato oblast by nesousedila s dalšími oblastmi po celém svém obvodu. V případě, že plocha nesousedí se žádnou oblastí, je nalezen čtyřúhelník, jehož rohy tvoří uzlové body. Jedná se o body obsahující maximální nebo minimální x-ovou a y-ovou souřadnici z daného polygonu. V situaci, kdy polygon obsahuje pouze jeden uzlový bod, je k němu přidán bod, jenž je v rámci oblasti od tohoto uzlového bodu nejvzdálenější. Ukázka s označením nalezených uzlů se nachází na obrázku 20.



Obrázek 20: Označení uzlových bodů (vlevo) a následná generalizace (vpravo)

Polygony se poté rozpadnou na linie, které jsou označeny dle toho, do kterého polygonu původně patřily, jestli se jedná o vnější hranici nebo o hranici díry,

a také o kolikátý úsek polygonu se jedná. Pokud je linie také částí sousedního polygonu, je do tohoto jednoznačného identifikačního klíče přidán údaj i o tomto polygonu a detailním umístění v něm. Kolem jednotlivých linií jsou vygenerovány konvexní obálky a poté jsou generalizovány podle vybraného algoritmu. Následuje poskládání linií zpět do polygonů. Tento způsob se jeví jako nejefektivnější. Každá linie je generalizovaná pouze jednou a nemůže dojít k rozdílným výsledkům, pokud by v rámci jednoho polygonu generalizační algoritmus postupoval opačným směrem než u sousedního. I když je v průběhu generalizačních algoritmů testován vznik nechtěných průsečíků, některé v datech zůstávají. Jedná se o případy, kdy mezi počáteční a koncový bod linie není v rámci generalizace vložen žádný další bod. Protože krajní body nejsou z důvodu zachování vazeb na další objekty odstraněny, může dojít ke křížení oblastí. Tento problém se týká oblastí, které nesousedí s žádnou další oblastí. Následuje jejich dodatečné testování a po nalezení průsečíku jsou odstraněny. Většinou se jedná o oblasti nedůležité v rámci generalizace, o čemž svědčí i maximální generalizace linií. Může ale dojít k odstranění významné oblasti. K tomuto radikálnímu kroku bylo přistoupeno, protože pouhé odstranění kritické hrany nevedlo vždy k optimálnímu výsledku. Někdy dokonce vzniknul další průsečík. Na závěr proběhne test, zda některá oblast neleží uvnitř jiné oblasti.

Tato třída obsahuje dvě pomocné funkce. Funkce `reverse(List<Point> l)` seřadí hodnoty seznamu v opačném pořadí. Druhou vytvořenou funkcí je `getKey (Map<ArrayList<Integer>>, List<Point> m, List<Point> l)`. Funkce předává klíč k zadané hodnotě z kontejneru typu `Map`.

4.2.4 Třída `Algoritmy`

Třída `Algoritmy` zahrnuje funkce, které provádějí vlastní generalizaci a další pomocné funkce. Během generalizace dochází k testování případného vzniku nechtěných průsečíků linie se sebou samou i s ostatními již generalizovanými liniemi. Proto mezi parametry funkcí patří nejen seznam bodů původní linie a geometrické hodnoty potřebné pro nastavení algoritmu, ale i seznam bodů z již generalizovaných úseků dané linie, konvexní obálky všech linií a seznam linií. Linie, které již byly generalizovány, průběžně nahrazují původní linie seznamu, který slouží k detekci průsečíků. Použité generalizační

algoritmy byly popsány v kapitolách 2.1.3 a 2.1.4, takže na tomto místě nebudou detailně rozebírány. Osvětleny budou další funkce, které se v této třídě nacházejí.

Douglas - Peuckerův algoritmus

Generalizaci tímto algoritmem provádí funkce `generalizacedp(List<Point> body, SortedMap<Integer,Point> nove, double h, int s, int k, ArrayList<LineString> obalky, ArrayList<LineString> testvse)`. Prvním argumentem je seznam bodů linie, která vstupuje do procesu generalizace. Kontejner `SortedMap<Integer,Point>` obsahuje body nově vytvářené linie. Tento parametr je potřeba, protože funkce volá sama sebe a k již existujícím bodům přidává další body v následujících krocích generalizace. Body obsažené v tomto kontejneru jsou také použity pro detekci průsečíků vznikající linie se sebou samou. Parametr `h` určuje míru generalizace, `s` a `k` označují počáteční a koncový bod pro daný krok generalizace. Poslední dva argumenty funkce předávají seznam linií a jejich obálky. Funkce vrací objekt, který obsahuje původní pozice bodů generalizované linie a k nim dané body. Tyto dvojce jsou vzestupně seřazeny podle pozice, která je klíčovou hodnotou dvojice.

Algoritmus začíná vložením počátečního a koncového bodu původní linie do linie nové. V každém kroku generalizace je nalezen nejvzdálenější bod od spojnice počátečního a koncového bodu, který splňuje kritérium minimální vzdálenosti. V případě, že takový bod neexistuje, není žádný vrchol vložen a algoritmus končí. Nalezený bod je do generalizované linie vložen pouze v případě, že ani jedna ze vznikajících hran nezpůsobí průsečík s danou linií nebo s jinou existující linií. Pokud by průsečík vznikl, použije se bod s druhou největší vzdáleností od spojnice a opět proběhne test existence průsečíku. Tento postup se opakuje, dokud není nalezen bod, který nezpůsobí průsečík, nebo dokud se nevyčerpají všechny body mezi počátečním a koncovým bodem. V tomto případě není dodrženo kritérium minimální vzdálenosti od spojnice počátečního a koncového bodu. Pokud byl nalezen bod se vzdáleností vyhovující kritériu a nemohl být vložen, aby nezpůsobil průsečík, je z hlediska vystižení průběhu čáry lepší vložit bod, který toto kritérium nesplňuje, než žádný bod.

Whyattův algoritmus

Funkce `generalizacew(List<Point> bodyl, double smin, ArrayList<LineString> obalky, ArrayList<LineString> testvse)` provádí generalizaci Whyattovým algoritmem. První parametr funkce představuje body původní linie. Druhým argumentem je kritérium minimálního obsahu. Následuje seznam všech linií a jejich obálek. Vracen je seznam bodů generalizované linie.

Během generalizace dochází k testování obsahů trojúhelníků vzniklých třemi po sobě jdoucími body linie. Pokud vypočítaný obsah není větší než stanovené kritérium, je prostřední bod vypuštěn. Poté se testuje existence průsečíku nově vzniklé hrany s ostatními hranami linie i s ostatními liniemi. Pokud by vznikl průsečík, je vyřazený bod vložen zpět. Tento postup je na celou linii opakován, dokud existují trojúhelníky, které nevyhovují minimálnímu kritériu.

Reumann-Witkamův algoritmus

Reumann-Witkamův algoritmus prezentuje funkce `generalizacer (List<Point> bodyl, double h, ArrayList<LineString> obalky, ArrayList<LineString> testvse)`. Jako u předchozích generalizačních funkcí parametry funkce představují seznam bodů vstupní linie, šířka koridoru, obálky všech linií a ostatní linie. Vracen je seznam bodů generalizované linie.

Algoritmus je založen na tvorbě koridorů, jehož osy jsou hrany linie. Nejprve je vytvořen koridor kolem první hrany a všechny další hrany, které leží mezi počátečním bodem a první hranou, která je protnuta vytvořeným koridorem se vypustí. Funkce je napsána tak, že počítá kolmé vzdálenosti jednotlivých vrcholů linie od počáteční hrany a pro každou další hranu porovnává, zda vzdálenost počátečního i koncového bodu je menší než šířka koridoru. Pokud je vzdálenost počátečního bodu menší než šířka koridoru a vzdálenost koncového bodu větší než šířka koridoru, musí hrana koridor protínat. Všechny body mezi počátečním vrcholem startovací hrany a počátečním vrcholem hrany protnuté koridorem by měli být vypuštěny. Postupně se vypouští jeden bod za druhým a vždy se testuje vznik nechtěného průsečíku. Pokud by tento jev nastal, bod bude do linie zpět vložen. Hrana, která byla protnuta koridorem, se stává osou dalšího

koridoru a celý postup se opakuje do té doby, než nebude nalezen žádný průsečík. V tomto případě se vypouští všechny body kromě posledního vrcholu linie.

Langův aloritmus

Funkce `generalizacel(List<Point> bodyl, double h, int p, ArrayList<LineString> obalky, ArrayList<LineString> testvse)` aplikuje na vstupní linii Langův algoritmus. Do funkce vstupují body původní linie, šířka koridoru, počet bodů v jednom kroku generalizace, obálky všech existujících linií a seznam těchto linií. Vrácen je seznam bodů generalizované linie.

Na začátku je spojen první bod linie s bodem, který je o *p* bodů dále. Počítají se kolmé vzdálenosti bodů mezi nimi od této spojnice. Pokud je aspoň jeden bod vzdálenější než je šířka koridoru, všechny body v linii zůstávají a spojí se počáteční bod s bodem vzdáleným o *p-1*. Opět se počítají vzdálenosti zbývajících bodů od spojnice. Tento postup se opakuje do chvíle, kdy žádný z bodů nepadne vně koridoru nebo už nejsou k dispozici další body pro vytvoření spojnice. Když žádný z bodů nepadne vně koridoru, všechny kromě bodů tvořících spojnici se vypustí. Vypouští se jeden po druhém a probíhá testování vzniku nechtěného průsečíku. Pokud by průsečík existoval, vypuštěný bod se vloží zpět. V následujícím kroku generalizace je tvořena spojnice mezi koncovým bodem předchozí poslední úsečky a bodem, který je od něj o *p* bodů dále. Generalizace končí, pokud je *p* větší než počet zbývajících bodů linie. V tomto případě je koncovým bodem spojnice poslední vrchol linie a následuje testování vzdálenosti ostatních bodů od vytvořené linie, jak je popsáno výše.

Převod linie na seznam bodů

Aby se dalo s body linie lépe pracovat, je geometrický typ `LineString` převeden pomocí funkce `linetopoint(LineString ls)` na seznam bodů. Poté lze jednotlivé body editovat, přidávat i odstraňovat.

Vzdálenost dvou bodů

Funkce `delka(Point a, Point b)` počítá vzdálenost *d* mezi dvěma body podle vztahu (3). Hodnoty x_a, y_a jsou souřadnicemi bodu A a hodnoty x_b, y_b představují souřadnice bodu B. Návratovou hodnotou je typ `double`.

$$d = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} \tag{3}$$

Obsah trojúhelníka

Obsah trojúhelníka `obsah(Point a, Point b, Point c)` je počítán ze souřadnic podle Heronova vzorce (4). Návratovou hodnotou je typ double.

$$S = \sqrt{s(s-a)(s-b)(s-c)} \tag{4}$$

kde $s = \frac{a+b+c}{2}$ a *a*,*b*,*c* jsou strany trojúhelníka.

Kolmá vzdálenost bodu od přímky

Vstupem do funkce `kolmavzd(Point a, Point b, Point c)` jsou souřadnice třech bodů. První dva body definují přímku a u třetího bodu je určována kolmá vzdálenost od přímky. Tato vzdálenost je výškou ve vzniklém trojúhelníku. K výpočtu je využit opět Heronův vzorec (viz výše) a výpočet obsahu trojúhelníka *S* na základě znalosti výšky v_c (5) a délky příslušné strany *c* v trojúhelníku.

$$v_c = \frac{2S}{c} \tag{5}$$

Maximum

Do funkce `maximum(ArrayList<Double> a)` vstupuje seznam hodnot typu double. Ty jsou ukládány do kontejneru `SortedMap`. Tento kontejner uchovává dvojice hodnot. První z těchto hodnot je klíčem. Jedná se o jedinečný identifikátor v rámci kontejneru. Všechny dvojice jsou řazeny dle klíčové hodnoty. V této funkci jsou klíčovými

hodnotami vstupující data. K nim je přiřazována pozice ze vstupního seznamu hodnot. Vrácený objekt tedy obsahuje seznam vzestupně seřazených klíčových hodnot a jejich původní pozici.

Pokud by nastala situace, že se některá klíčová hodnota vyskytuje v seznamu vícekrát, je původní pozice nahrazena nově nalezenou.

Existence průsečíku dvou hran

Pro detekci průsečíků byly vytvořeny tři varianty funkce `prusecik`. Jednotlivé funkce se liší seznamem argumentů.

Základní varianta funkce `prusecik(Point p0, Point p1, ArrayList<Point> test, ArrayList<LineString> obalky, ArrayList <LineString> testvse)`, je volána funkcemi pro generalizaci linie, aby se zabránilo vzniku nechtěných průsečíků linie se sebou samou nebo dvou různých linií. Návratová hodnota je typu `boolean` a na začátku funkce je nastavena na hodnotu `false`. První dva parametry funkce představují počáteční a koncový bod hrany, která by vznikla vložením koncového bodu do generalizované linie. Další argument představuje body právě generalizované linie. Nejprve proběhne výpočet průsečíku testované hrany se všemi existujícími hranami dané linie. Pokud je nalezen průsečík, je návratová hodnota funkce nastavena na hodnotu `true` a výpočet je přerušen. Pokud není nalezen průsečík, funkce pokračuje hledáním průsečíku s ostatními liniemi. Tento test probíhá tak, že se hledá průsečík s obálkami linií. Pouze pokud existuje průsečík s obálkou, testuje se vznik průsečíku přímo s liniemi. Po nalezení průsečíku je návratová hodnota funkce nastavena na hodnotu `true` a výpočet je ukončen.

Funkce `prusecik (Point p0, Point p1, ArrayList<LineString> testvse)` slouží k nalezení průsečíku hrany dané počátečním a koncovým bodem s některou z existujících linií, které jsou zadány seznamem objektů typu `LineString`. Návratová hodnota je opět typu `boolean` a na začátku funkce je nastavena na hodnotu `false`. Pokud je průsečík nalezen, změní se na hodnotu `true` a výpočet je přerušen.

Poslední varianta funkce `prusecik(Point p0, Point p1, Polygon testvse)` se od předchozích funkcí liší. Návratovou hodnotu totiž není hodnota typu

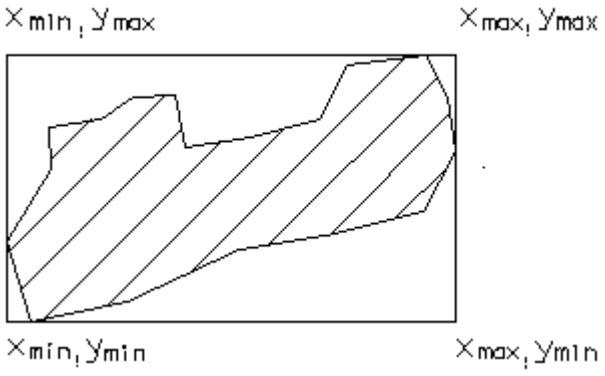
`boolean`, která říká, zda průsečík existuje či ne, ale návratová hodnota je typu `integer` a udává počet nalezených průsečíku testované polopřímky s polygonem. Princip hledání průsečíku je stejný jako u výše uvedených funkcí, ale v okamžiku nalezení průsečíku není výpočet přerušen. Proměnná, do které se ukládá počet nalezených průsečíků, se zvýší o jednu a výpočet pokračuje, dokud nejsou použity všechny hrany polygonu.

Konvexní obálka množiny bodů

Konvexní obálku množiny bodů určuje funkce `hull(List<Point> body)`. Parametr funkce představuje seznam bodů linie, jejíž konvexní obálka je požadována. Postup tvorby obálky je naznačen v kapitole 4.3.2. Vrací se seznam bodů obálky.

Souřadnice minimálního opsaného obdélníka oblasti

Funkce `minmax(Polygon p)` určuje minimální opsaný obdélník oblasti, která je zadána polygonem. Body polygonu jsou seřazeny podle souřadnice *x* a také podle souřadnice *y*. Na výstupu je seznam bodů, které představují rohy hledaného obdélníka. Souřadnice prvního bodu jsou tvořeny minimální souřadnicí *x* (x_{min}) a minimální souřadnicí *y* (y_{min}) ze všech vstupních bodů. Následují body o souřadnicích [x_{max}, y_{min}], [x_{max}, y_{max}] a [x_{min}, y_{max}]. Minimální opsaný obdélník se nachází na obrázku 21.



Obrázek 21: Minimální opsaný obdélník

4.2.5 Třída `Vektor`

Objekt typu `Vektor` obsahuje dva privátní členy. Jsou jimi hodnoty typu `double`. Tento objekt odpovídá matematickému pojetí vektoru, takže členy přestavují souřadnicové rozdíly počátečního a koncového bodu vektoru. Parametry konstruktoru jsou právě hodnoty souřadnicových rozdílů. Dále jsou k dispozici metody pro předávání hodnot vektoru a pro nastavení těchto hodnot.

4.3 Řešení topologických problémů

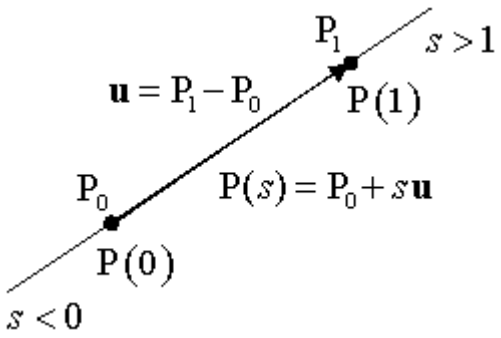
Generalizace úzce souvisí s topologií. Vypuštěním bodů, které nejsou významné z hlediska tvaru křivky, může dojít ke ztrátě důležitých vazeb v datech. Klíčovou roli zde hraje zachování počátků a konců liniových prvků, uzlových bodů polygonů a všech dalších bodů, ve kterých dochází k nějakému větvení čar. Dodržení této zásady ale nestačí. Během generalizace dochází také ke vzniku nechtěných průsečíků. Jedná se o průsečíky linie se sebou samou a o průsečíky s ostatními objekty. Při generalizaci ploch dochází kromě křížení objektů ještě k tomu, že se jedna plocha může stát podmnožinou plochy jiné.

Navržená funkce v průběhu generalizačního algoritmu eliminuje vznik nechtěných průsečíků testováním, zda nově vkládaný či vypouštěný vrchol vytvoří hranu, která protíná některou hranu nově vznikající linie nebo již generalizovaných linií. Pokud by došlo ke vzniku průsečíku, vrchol nebude vypuštěn či vložen. Protože porovnávání se všemi existujícími hranami by bylo časově náročné, využijí se konvexní obálky vytvořené kolem linií. Pouze pokud testovaná hrana protíná obálku, následuje počítání průsečíku se všemi hranami dané linie. Určení průsečíku a konstrukce konvexní obálky jsou popsány níže.

Ověření, zda oblast není podmnožinou jiné oblasti, probíhá na základě Rayova algoritmu. Princip vysvětluje kapitola 4.3.3.

4.3.1 Průsečík dvou hran

Princip testování existence průsečíku spočívá v parametrickém vyjádření dvou přímek, jejichž segmenty jsou dané hrany. Každá přímka je tedy určena počátečním (P_0) a koncovým (P_1) bodem testované hrany. Situaci popisuje obrázek 22. Rovnice přímky je $P(s) = P_0 + s\boldsymbol{u}$. Parametr s představuje libovolné reálné číslo. Pokud bude s z intervalu <0,1>, nachází se bod $P(s)$ na úsečce P_0P_1. Pro druhou přímku platí $Q(t) = Q_0 + t\boldsymbol{v}$.

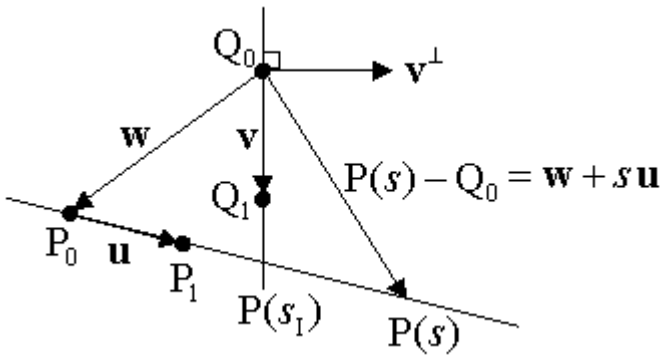


Obrázek 22:Parametrické vyjádření přímky (převzato z [15])

Průsečík může být označen $I = P(s_I) = Q(t_I)$. Vektor $\boldsymbol{v}$ lze psát jako $\boldsymbol{v} = \boldsymbol{w} + s\boldsymbol{u}$ (viz obrázek 23). Využitím vlastností kolmých vektorů vznikne $0 = \boldsymbol{v}^{\perp}(\boldsymbol{w} + s\boldsymbol{u})$. Dalšími úpravami se z daného vztahu vyjádří parametr s_I (6) a analogicky se dospěje k parametru t_I (7). Průsečík existuje v případě, že $s_I \in \langle 0,1\rangle \wedge t_I \in \langle 0,1\rangle$.

$$s_I = \frac{-v^{\perp}w}{v^{\perp}u} = \frac{v_2w_1 - v_1w_2}{v_1u_2 - v_2u_1} \qquad (6)$$

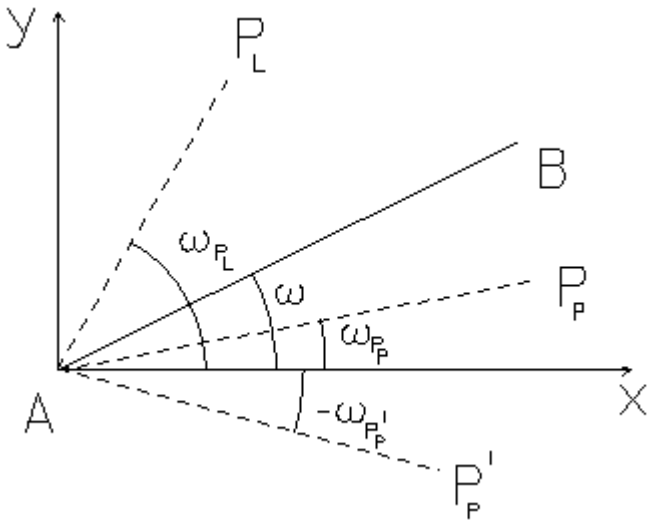
$$t_I = \frac{u^{\perp}w}{u^{\perp}v} = \frac{u_1w_2 - u_2w_1}{u_1v_2 - u_2v_1} \qquad (7)$$



Obrázek 23: Průsečík dvou přímek (převzato z [15])

4.3.2 Konvexní obálka množiny bodů

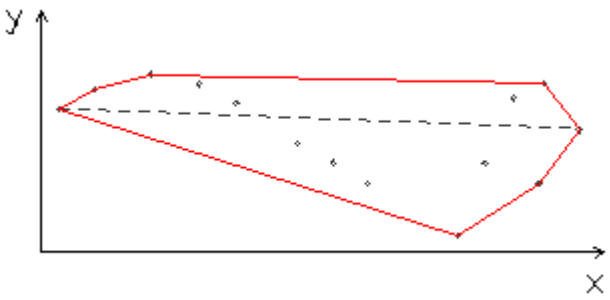
Prvním krokem je seřazení množiny bodů podle souřadnice *x*. Následuje spojení prvního a posledního bodu uspořádané množiny. Touto úsečkou jsou body rozděleny na dvě skupiny. Z bodů nad úsečkou je tvořena horní část obálky a z druhé půlky spodní část obálky. Do obálky vždy vstupuje bod, který se od poslední hrany obálky vyskytuje nejvíce vlevo. Pokud se vlevo žádný bod nevyskytuje, je vložen bod, jenž je nejméně vpravo. Zda je bod od úsečky vpravo nebo vlevo lze zjistit z rozdílů směrníků hrany a spojnice počátečního bodu hrany a testovaného bodu. Pokud je souřadnice *x* počátečního bodu hrany menší než souřadnice *x* koncového bodu a zároveň platí $(\omega - \omega_P) > 0$, bod se nachází vpravo. Pokud je tento rozdíl záporný, leží testovaný bod vlevo. Tato situace nastává při tvorbě horní části obálky. Během tvorby spodní obálky je souřadnice *x* počátečního bodu větší než souřadnice bodu koncového a znaménko rozdílu směrníků je opačné. Situaci popisuje obrázek 24. Bod P_L se nachází vlevo od úsečky *AB* a body P_P, P_P' leží vpravo.



Obrázek 23: Poloha bodu vzhledem k orientované úsečce

Na začátku procesu je do obálky vložen bod s nejmenší souřadnicí *x*. V tuto chvíli, kdy obálka obsahuje pouze jeden bod, není k dispozici žádná hrana. Jako startovací hrana poslouží úsečka dělící množinu bodů na horní a dolní skupinu. Od této orientované úsečky s počátkem v bodě o nejmenší souřadnici *x* se najde bod, který je nejvíce vlevo. Vloží se do obálky a postupuje se v tvorbě horní obálky. Pokud se vlevo žádný bod

nevyskytuje, je horní obálka tvořena startovací úsečkou. Výpočet končí v bodě s maximální souřadnicí *x*. Od tohoto bodu je analogicky určena dolní část obálky. Výsledek vznikne spojením horní a dolní obálky. Ukázka konvexní obálky se nalézá na obrázku 25. Obálka množiny bodů je vyznačena červenou barvou.



Obrázek 24: Konvexní obálka množiny bodů

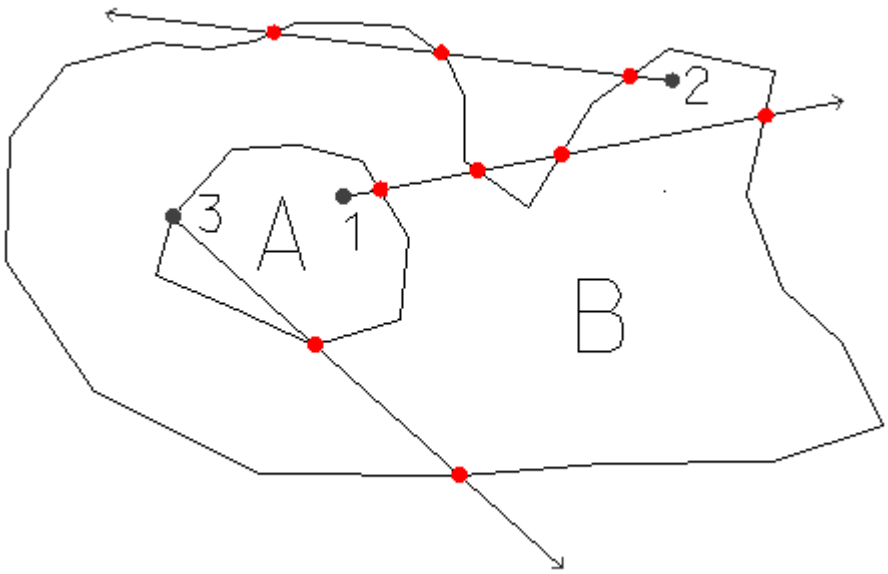
Při tomto postupu jsou body do obálky vkládány ve správném pořadí. Během každého testování jsou v případě horní obálky porovnávány pouze body o vyšší souřadnici *x*, než je stávající vrchol hrany, a u dolní obálky, kdy se postupuje opačně, pouze body o nižší souřadnici *x*.

4.3.3 Vzájemná poloha bodu a uzavřené oblasti

Zjišťování, zda je oblast zahrnuta v nějaké jiné oblasti, je analogické k detekci vzájemné polohy bodu a oblasti. Pokud neexistuje žádný průsečík mezi dvěma oblastmi, mohou nastat následující situace. Oblasti leží vedle sebe, nebo je jedna z oblastí součástí druhé. Pokud se zájmová oblast nachází uvnitř jiné oblasti, poté i všechny body zájmové oblasti leží uvnitř. Proto stačí určit polohu libovolného bodu zájmové oblasti vzhledem k oblasti druhé.

Jedním z algoritmů, které slouží k detekci vzájemné polohy bodu a oblasti, je Ray Algorithm (viz [1]). Metoda je známá také jako paprskový algoritmus a je postavena na tom, že pokud se bod nalézá uvnitř oblasti, musí polopřímka vedená z tohoto bodu způsobit minimálně jeden průsečík s oblastí. Počet průsečíků záleží na složitosti tvaru oblasti, ale vždy se jedná o lichý počet. V případě, že se bod nachází mimo oblast, je počet průsečíků s oblastí nulový nebo sudý. Chybný výpočet může způsobit polopřímka procházející vrcholem uzavřené oblasti, protože ve vrcholu jedna hrana končí a druhá

začíná a místo jednoho průsečíku by byly nalezeny dva. Průsečík neexistuje, pokud je hrana oblasti podmnožinou testovací polopřímky. Názorná ukázka se nachází na obrázku 26. Bod 1 je uvnitř oblasti A, se kterou má polopřímka jeden průsečík a mimo oblast B, protože s ní existují čtyři průsečíky.



Obrázek 25: Ray Algorithm

Implementace spočívá v porovnávání minimálních opsaných obdélníků oblastí. Na základě souřadnic rohů těchto obdélníků se zjistí, které oblasti mohou být uvnitř jiné. Na podezřelé dvojice je aplikován test o počtu průsečíků dle paprskového algoritmu a oblasti nacházející se uvnitř jsou vypuštěny. Polopřímka vedená s oblasti, jež by mohla být uvnitř, je zadána pomocí prvního a třetího bodu jejího hraničního polygonu. Mohly by to být jakékoliv dva body, které nejdou hned po sobě, nebo libovolné body z této oblasti. Pro jednoduchost byly zvoleny body z hraničního polygonu. Pokud by byly zvoleny dva po sobě jdoucí body z tohoto polygonu, mohlo by dojít k chybnému vypuštění oblasti, která vyplňuje tzv. díru. Na obrázku tuto díru vyplňuje oblast A. V tomto případě by totiž hrana vnitřní hranice obklopující oblasti byla podmnožinou testovací přímky. Průsečík by neexistoval. Vznikl by pouze průsečík s vnější hranicí a kvůli lichému počtu průsečíků by byla tato oblast chybně vypuštěna.

4.3.4 Nevyřešené problémy

Přes veškerou snahu zamezit vzniku nechtěných průsečíků, existují případy, ve kterých průsečíky v datech zůstávají. Tato situace nastává ve dvou níže popsaných případech.

Pokud jsou po generalizaci křížící se linie tvořeny pouze spojnicí počátečního a koncového bodu nebo uzlových bodů, nenastala žádná možnost vypustit či vložit bod a průsečík v generalizovaných datech zůstává. Řešením by mohlo být odstranění celé linie, ale není jistota, zda tato linie není například částí řeky ústící do moře. Řeka nemůže skončit uprostřed pevniny. Tyto průsečíky tedy odstraněny nejsou.

Druhým problémem je překryt konvexních obálek původních linií. Během detekce průsečíků jsou testovány linie, jejichž obálku protíná testovaná hrana. V případě, že zájmová hrana leží uvnitř obálky jiné linie, neexistuje s touto obálkou průsečík a není tedy ani zjištěno, zda hrana linii protíná.

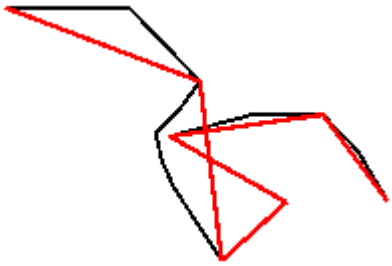
V PostGIS existuje funkce, která dokáže průsečíky odhalit. Tuto funkci je možné použít na generalizovaná data. Uživatel musí danou linii individuálně posoudit a popřípadě opravit nebo smazat.

4.4 Porovnání dat generalizovaných různými algoritmy

Vytvořená funkce byla testována na různých datech, která byla k dispozici. Liniové prvky jsou zastoupeny řekami Severní Ameriky a silnicemi Starého Města pod Sněžníkem. Z plošných dat se jedná o hranice krajů ČR a hranice států Severní Ameriky, oblasti Karibského moře a části Jižní Ameriky.

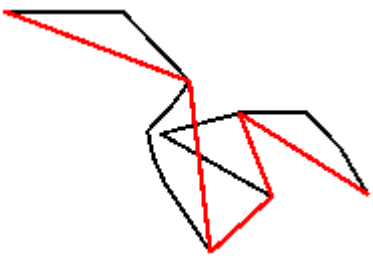
4.4.1 Řešení nechtěného průsečíku generalizované linie se sebou samou

Následujících pět obrázků ukazuje, jak se který algoritmus vypořádá se vznikem nechtěného průsečíku linie se sebou samou. Na prvním obrázku (viz Obrázek 27) se nachází výsledek funkce `simplify(geometry, double precision)`, která je již obsažena v PostGIS. Černá barva značí původní linii a barva červená je výsledkem

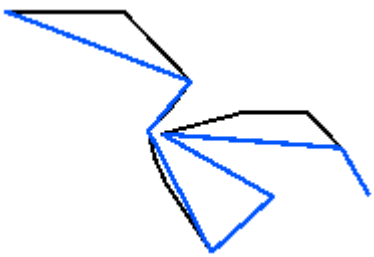


Obrázek 26: Generalizace Dougles-Peuckerovým algoritmem bez kontroly vzniku nechtěných průsečíků (h = 0.15)

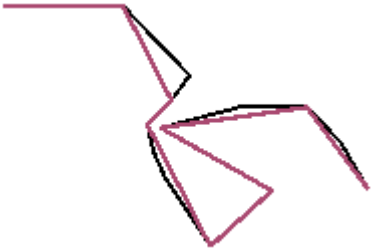
generalizace. Další ukázky patří generalizačním algoritmům, jež byly implementovány v rámci této diplomové práce. Původní linie má ve všech případech černou barvu. Výsledky generalizace jsou vyznačeny červeně (Douglas-Peuckerův algoritmus), modře (Langův algoritmus), zeleně (Whyattův algoritmus) a fialově (Reumann-Witkamův).



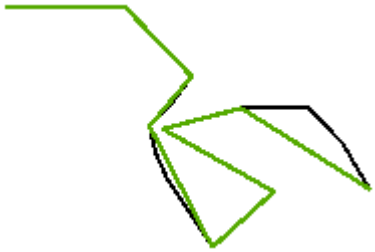
Obrázek 28: Generalizace implementovaným Douglas-Peuckerovým algoritmem (h = 0.15)



Obrázek 27: Generalizace implementovaným Langovým algoritmem (h = 0.3, p = 3)



Obrázek 30: Generalizace implementovaným Reumann-Witkamovým algoritmem (h = 0.3)



Obrázek 29: Generalizace implementovaným Whyattovým algoritmem (s = 0.2)

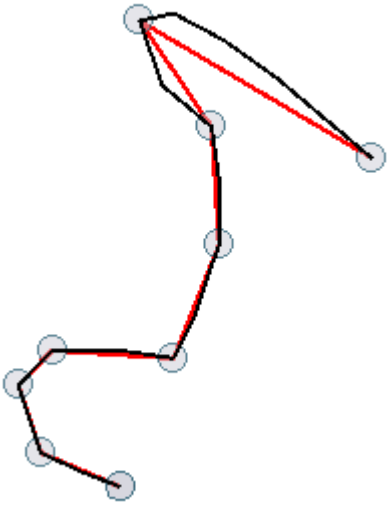
4.4.2 Porovnání algoritmů

Aby bylo možné jednotlivé algoritmy spravedlivě posoudit, byly nastaveny tak, aby generalizovaná linie měla vždy stejný počet bodů. Při daném počtu bodů lze říci, který výsledek původní linii nejlépe vystihuje. Pro tento test byla vybrána část řeky. Výchozí linii tvoří 17 bodů. Generalizovaná linie má bodů 9.

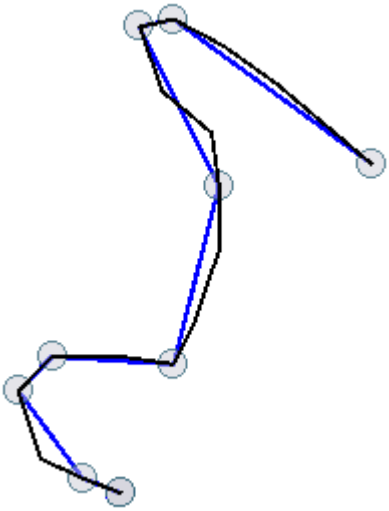
Všechny algoritmy si s generalizací poradily docela dobře. Na první pohled je vidět, že výsledky rozšířených lokálních algoritmů jsou si podobné a pro globální algoritmy to platí také. Generalizace zhoršuje přesnost původní linie. Míru přesnosti lze určit na základě porovnání polohy bodů původní a generalizované linie. Testována je plocha mezi původní a generalizovanou linií a kolmé vzdálenosti bodů výchozí linie od linie výsledné. Všechny hodnoty jsou přibližné a byly určeny pomocí měřících nástrojů v SW Quantum GIS. Obrázky výsledků generalizace a tabulka s naměřenými hodnotami se nachází níže.

Algoritmus	**Vzdálenost původního bodu od generalizované linie [mj]**																		**Plocha [mj^2]**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**∅**	
D - P	0	1	0	0	0	5	0	4	0	5	0	21	0	28	**31**	24	0	**7**	9200
R - W	0	0	26	0	0	5	0	4	0	26	**38**	0	0	0	9	10	0	**7**	8400
Whyattův	0	1	0	**36**	0	5	0	4	0	5	0	21	0	18	0	4	0	**6**	6300
Langův	0	0	**26**	0	0	5	0	11	19	0	20	10	0	0	9	10	0	**7**	8000

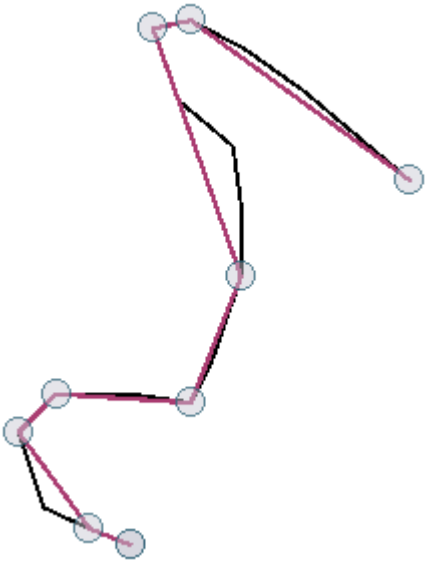
Tabulka 2: Porovnání generalizačních algoritmů



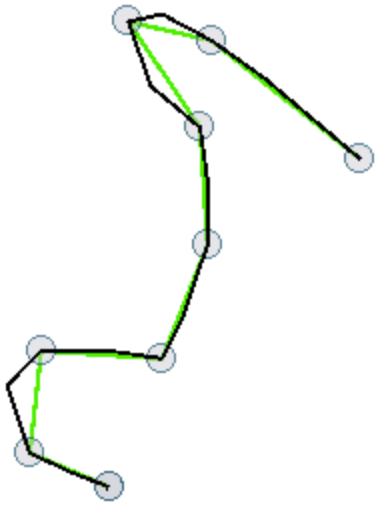
Obrázek 32: Douglas-Peuckerův algoritmus



Obrázek 31: Langův algoritmus



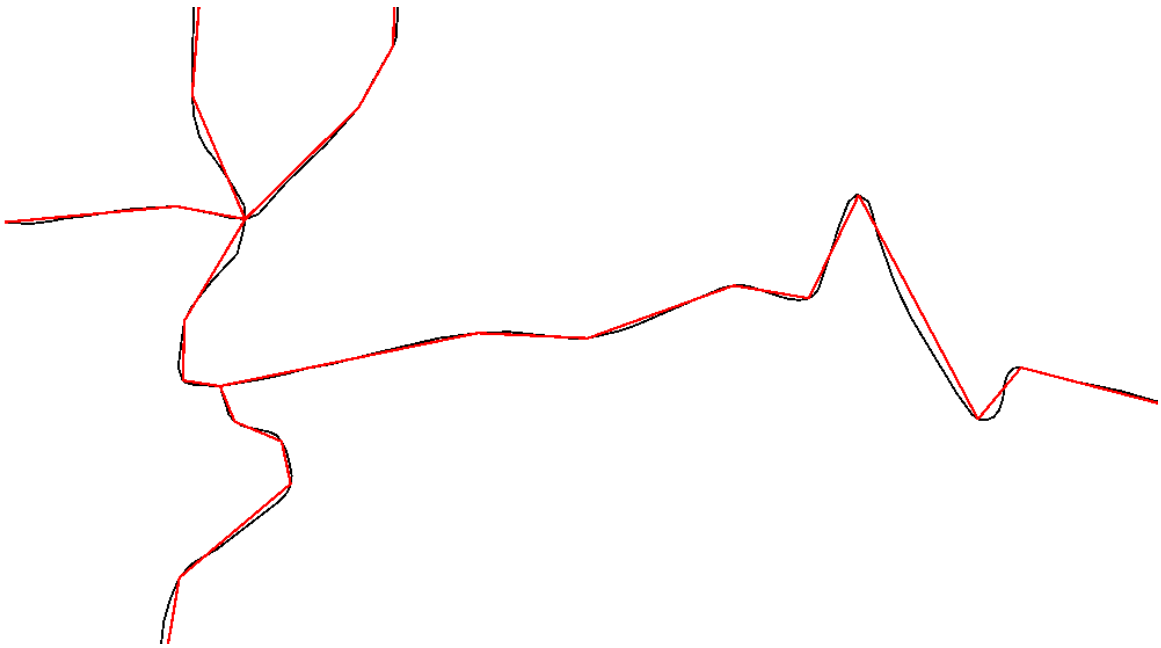
Obrázek 34: Reumann-Witkamův algoritmus



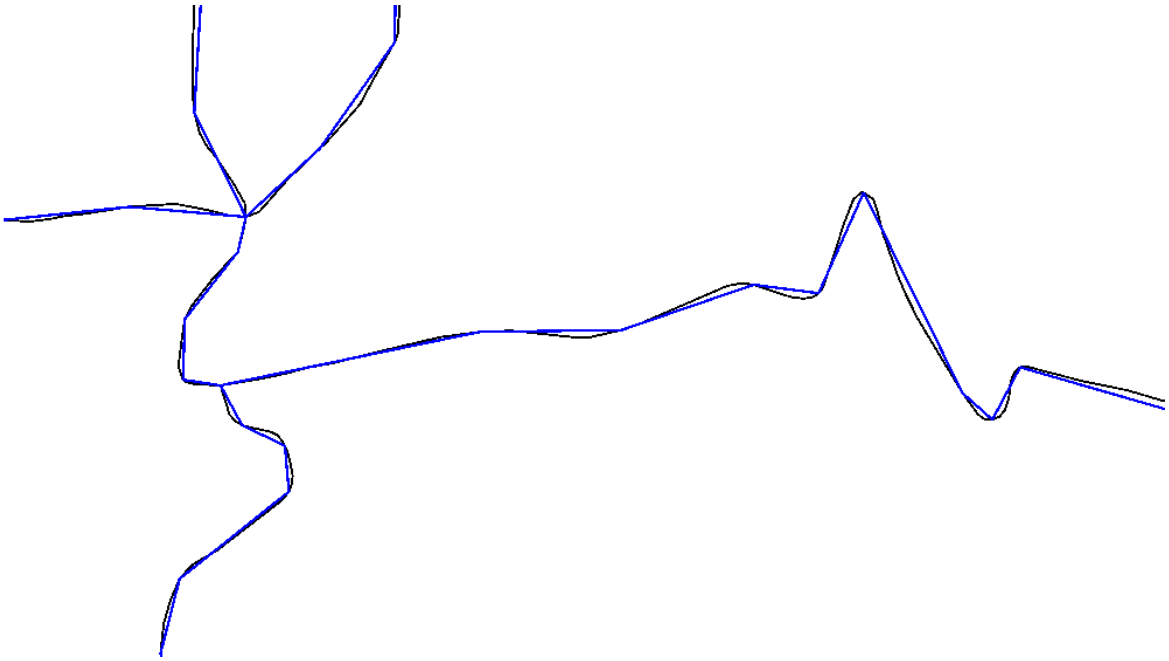
Obrázek 33: Whyattův algoritmus

Z uvedené tabulky vyplývá, že plocha, jež je ohraničena původní a generalizovanou linií je nejmenší v případě Whyatova algoritmu. Tento výsledek byl očekáván, protože princip algoritmu je založen právě na testování obsahu trojúhelníka, který vzniká pod vypouštěným bodem. Zajímavé je ale další pořadí. Douglas-Peuckerův algoritmus všeobecně považovaný za nejlepší z testovaných algoritmů se umístil na posledním místě. Největší rozdíl v poloze algoritmem vypuštěného bodu a generalizované linie se vyskytuje u generalizace Reumann-Witkamovým algoritmem. Tento algoritmus dává i po vizuální stránce při stanoveném počtu bodů nejhorší výsledek. Výsledky generalizace nelze posuzovat pouze na základě porovnávání geometrických parametrů, ale podle toho, jak generalizovaná linie vystihuje původní křivku. Výsledek Douglas-Peuckerova algoritmu kopíruje nejvěrněji původní křivku na nejdelší části linie.

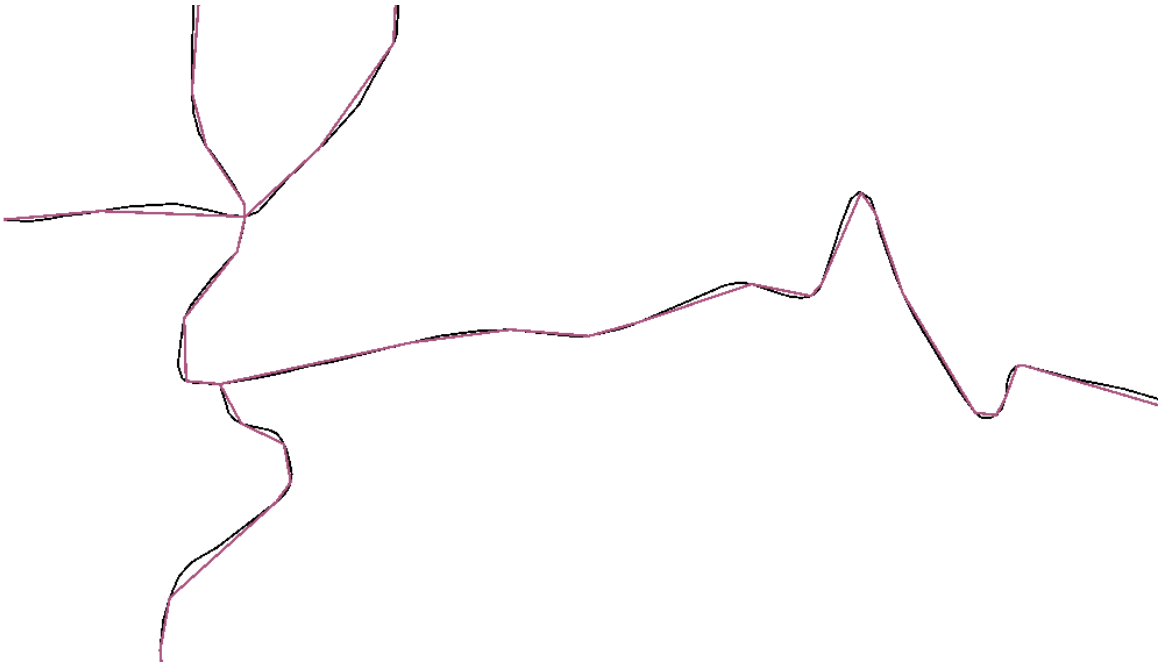
Pro další testování byla vybrána část silniční sítě v lokalitě Starého Města pod Sněžníkem. Tentokrát byla snaha o nastavení jednotlivých algoritmů tak, aby si výsledky generalizace byly co nejvíce podobné (viz obrázek 36-39). Vstupní data obsahovala 130 bodů. Uvedeného výsledku dosáhl Douglas-Peuckerův algoritmus s použitím 20 bodů. Langův algoritmus s 22 body, Whyattův algoritmus s 28 body a Reumann-Witkamův algoritmus potřeboval 32 bodů.



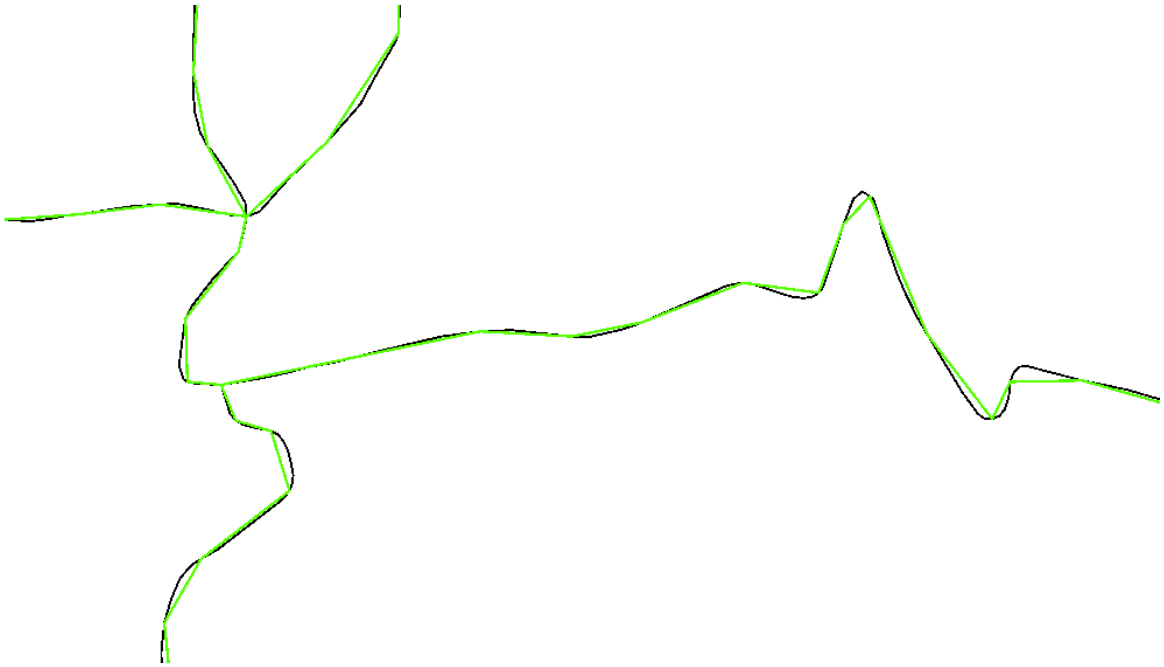
Obrázek 35: Douglas-Peuckerův algoritmus – silnice Staré Město pod Sněžníkem



Obrázek 36: Langův algoritmus – silnice Staré Město pod Sněžníkem



Obrázek 37: Reumann-Witkamův algoritmus – silnice Staré Město pod Sněžníkem



Obrázek 38: Whyatův algoritmus – silnice Staré Město pod Sněžníkem

4.4.3 Ukázka generalizace ploch

Porovnání jednotlivých algoritmů při aplikaci na plošné objekty znázorňuje obrázek 40. Nachází se na něm kraj Hlavní město Praha. Situace je zobrazena v měřítku 1 : 2 000 000. Nejvíce vlevo je původní hranice kraje. Následují hranice po použití Douglas-Peuckerova (h = 2 km), Langova (h = 3 km, p = 20), Reumann-Witkamova (h = 3 km) a nejvíce vpravo Whyattova (s = 2 km^2) algoritmu.

Obrázek 39: Generalizace kraje Hlavní město Praha

Po prohlédnutí jednotlivých výsledků generalizace, je zřejmé, že Douglas-Peuckerův algoritmus dává nejvýstižnější hranici. Použití nejmenšího počtu bodů pro lomenou čáru však také znamená nejméně vyhlazenou hranici.

4.4.4 Chování jednotlivých algoritmů

Po provedeném testování algoritmů následuje shrnutí získaných poznatků.

Douglas-Peuckerův algoritmus

Douglas-Peuckerův algoritmus je nejúčinnější při zachovávání tvaru čáry. Algoritmus nejlépe vystihuje původní linii za použití nejmenšího počtu bodů ze všech vybraných algoritmů. Protože se generalizace používá právě pro úsporu objemu dat, je tato vlastnost tedy klíčovou k úspěchu a používání tohoto algoritmu.

Reumann-Witkamův algoritmus

Reumann-Witkamův algoritmus vyniká svojí rychlostí. Potřebuje však nejvíce bodů k tomu, aby dosáhl srovnatelných výsledků s ostatními metodami. Jak vyplývá z popisu algoritmu (kapitola 2.1.3), změna směru čáry vyvolá vložení bodu do generalizované linie. V oblouku tedy může být v rámci nastavené přesnosti vloženo zbytečně mnoho bodů do nově vznikající křivky.

Langův algoritmus

Výsledky Langova algoritmu se dají ovlivnit dvěma parametry. Jedním je šířka koridoru a druhým počet bodů, se kterými se pracuje v jednom kroku generalizace. Algoritmus je podrobně popsán v kapitole 2.1.2. Šířkou koridoru je volena přesnost. Počet bodů vstupujících do jednoho kroku, reguluje počet bodů výsledné linie. Nemá smysl volit počet bodů roven jedné. Výsledkem by byla nezměněná data.

Whyattův algoritmus

Výhodou Whyattova algoritmu je jeho rychlost a snadná implementace. Princip algoritmu vychází z toho, že čím je testovaný bod linie $n+1$ významnější, tím je větší obsah trojúhelníku tvořeného třemi po sobě jdoucími body n, $n+1$, $n+2$. Pokud je testovaný bod hodně vzdálený, ale základna vzniklého trojúhelníka bude malá, bude i obsah malý a tento důležitý bod bude vypuštěn. I když algoritmus dosahuje dobrých

výsledků, je hodně závislý na charakteru vstupních dat, a proto se nepoužívá tolik jako Douglas-Peuckerův algoritmus.

5 Použitý software

Psaní kódu funkce probíhalo ve vývojovém prostředí Eclipse SDK 3.4. Používání Eclipse samozřejmě předcházela instalace Java JDK 1.5.0.

Pro správu databázového serveru PostgreSQL 8.2 bylo použito grafické rozhraní pgAdmin III verze 1.6.3. Součástí instalace PostgreSQL 8.2 bylo rozšíření o PostGIS 1.1.7.

Data, na kterých byla testována vytvořená funkce, byla k dispozici ve formátu shapefile. Součástí PostGIS jsou programy na převod dat ze shapefile do databáze a naopak. K importu slouží aplikace shp2pgsql, která byla využita během vypracovávání této diplomové práce.

Výstupy v grafické podobě byly prohlíženy pomocí Quantum GIS verze 0.11.0 - Metis, který umí připojit jako mapovou vrstvu tabulky z databáze PostGIS.

5.1 Eclipse SDK

Jedná se o multiplatformní open-source software napsaný v jazyce Java. Představuje grafické vývojové prostředí pro jazyk Java, které lze rozšířit instalováním různých zásuvných modulů. Poté může představovat vývojové prostředí pro jazyk C, C++, Python… Existují také moduly pro JavaScript, PHP, SQL, CVS, UML, HTML, XML. Standardní distribuce obsahuje vývojové prostředí (IDE), JDT modul pro programování Java aplikací a modul PDE pro vytváření nových zásuvných modulů do systému Eclipse. Poslední verzí je Eclipse SDK 3.4 (Ganymede).

Práce v tomto SW je velice příjemná. Editor zdrojových textů provádí barevné odlišení a zvýraznění syntaxe. Programování také zjednodušuje možnost automatického doplňování zdrojového kódu, doporučení řešení chyb, varování před potenciálním nebezpečím, barevné podtrhávání podezřelých částí kódu a propracovaná nápověda. Systém je nastaven tak, že zdrojové kódy automaticky kompiluje, takže je poté stačí jen spustit.

5.2 Java JDK

Java Development Kit (JDK) vyvíjí společnost Sun Microsystems. Součástí tohoto vývojového balíku je Java Runtime Environment (JRE) pro spouštění Java programů, kompilátor, debugger a knihovna základních tříd. S těmito nástroji lze pracovat z příkazové řádky. Zdrojový kód Javy je kompilován do bajtového kódu, který je interpretován virtuálním strojem Javy (JVM). Ten překládá bajtový kód do strojového jazyka, který běží na daném počítači.

5.3 Databázový server PostgreSQL

PostgreSQL je relační databázový systém s otevřeným zdrojovým kódem. Podporuje normy SQL92 a SQL99. Splňuje podmínky ACID, plně podporuje cizí klíče, operace JOIN, pohledy, spouště a uložené procedury. Systém může být doplněn o nové datové typy, funkce, operátory, agregační funkce a procedurální jazyky. Vlastní procedurální jazyk PL/SQL lze obohatit o C/C++, Perl, Python, Java. Databázový systém PostgreSQL lze rozšířit o nadstavbu PostGIS, která je popsána v kapitole 3.1. Nejnovější verzí je PostgreSQL 8.3.5.

5.4 pgAdmin III

Jde o volně šiřitelný multiplatformní nástroj pro snadnou správu databáze PostgreSQL. Zahrnuje grafické administrativní prostředí, nástroj pro psaní SQL dotazů se zvýrazňováním syntaxe, editor pro uložené procedury, možnost upravování konfiguračních souborů.

5.5 Quantum GIS

Quantum GIS (QGIS) je multiplatformní open-source GIS aplikace určená pro prohlížení a vytváření geografických dat. Podporuje vektorové, rastrové i databázové formáty. Umožňuje digitalizaci, tvorbu map, WMS, vytváření SQL dotazů nad atributovými daty a instalaci dalších zásuvných modulů. Mezi nejvýznamnější rozšíření patří možnost připojení vrstvy WFS, import ShapeFile do PostgreSQL a modul GRASS. Díky modulu GRASS se stává mocným nástrojem, který umí provádět analýzy prostorových dat, zpracování obrazových dat, prostorové modelování a vizualizaci. Aktuální verzí je QGIS 0.11.0 – Metis.

6 Závěr

Geografické informační systémy se rozšířily do mnoha odvětví. GIS lze uplatnit všude, kde je třeba vytvářet, editovat nebo kombinovat a analyzovat prostorově umístěná data. PostGIS patří mezi prostorové databáze, které umí takové údaje uchovávat. Data uložená v databázi mohou sloužit k různým úlohám. Dle účelu se také liší požadavky na objem a přesnost geodat. Jednou z možností, jak data přizpůsobit, je jejich generalizace.

Nejprve byly prostudovány dostupné prameny a proběhlo seznámení s metodami generalizace a různými generalizačními postupy. Rešerše literatury dává ucelený přehled o nalezených dokumentech, které se problematikou generalizace zabývají.

V rámci diplomové práce byla vytvořena funkce, která umožňuje generalizaci liniových a plošných objektů. Zdrojový kód je napsán v jazyce Java a z prostředí PostGIS je spouštěna pomocí PL/Java. Uživatel si může pro generalizaci vybrat jeden ze čtyř implementovaných algoritmů. Jedná se o Douglas-Peuckerův, Whyattův, Reumann-Witkamův a Langův algoritmus. Během vypracovávání této práce bylo zjištěno, že PostGIS již obsahuje jednu funkci, jež slouží ke generalizaci dat. Tato funkce provádí generalizaci Douglas-Peuckerovým algoritmem, ale neřeší na rozdíl od navržené funkce topologické úlohy týkající se vzniku nechtěných průsečíků či překrytu dat. Přínosem je také výběr z několika algoritmů, protože každý se chová trochu jinak a lze tedy použít ten nejvhodnější pro daná data či požadovaný výsledek. Vytvořená funkce může být použita nejen ke generalizaci, ale také k odhalení chyb v datech. Nepřesnosti, jež se vztahují například ke špatnému chytání na body během vektorizace, jsou zřejmé v generalizovaných datech, která tím ztrácí některé topologické vazby.

Vytvořená funkce bohužel není ideální. Existují situace, se kterými si neumí poradit (viz kapitola 4.3.4). Do budoucna by také bylo dobré přidat testování výměry generalizovaných oblastí a vypustit takové, které v rámci nastavených geometrických parametrů generalizace ztrácí význam. Napsaná funkce také nevyniká svojí rychlostí.

Tato práce sice nepřináší dokonalou univerzálně použitelnou funkci pro generalizaci geografických objektů, ale upozorňuje na problematiku generalizace a úskalí, která při ní vznikají. Dále přináší poznatky o využití jazyka Java v prostředí PostGIS. Cílem bylo seznámit se s generalizačními algoritmy a vyzkoušet je na reálných datech. Po závěrečném testování funkce lze konstatovat, že všechny vybrané algoritmy

se hodí ke generalizaci geografických objektů. Nejvýhodnější je použít Douglas-Peuckerův algoritmus, který nejlépe vystihuje průběh původní linie za použití nejmenšího počtu bodů ze všech testovaných algoritmů. Druhému místu odpovídá Langův algoritmus, protože jeho výsledek lze ovlivnit dvěma volitelnými parametry a dosáhnout tak optimálního výsledku.

Literatura

[1] BAYER, Tomáš. *Algoritmy v digitální kartografii*. 1. vyd. Praha : Karolinum, 2008. 251 s. ISBN 978-80-246-1499-1.

[2] VEVERKA, Bohuslav. *Tematická a topografická kartografie 10*. Praha : ČVUT v Praze , 2004. 220 s. ISBN 80-01-02381-8.

[3] ČERBA, Otakar. *Generalizace : Přednáška z předmětu Tematická kartografie* [online]. aktualizováno 12.12.2006 [cit. 2008-06-20]. Dostupný z WWW: <http://gis.zcu.cz/studium/tka/Slides/generalizace.pdf>.

[4] SRNKA, Erhart. *Matematicko-logické modelování procesu generalizace v kartografii*. Brno : s. n., 1979. 35 s. Vojenská akademie Antonína Zápotockého. Autoreferát dizertační práce.

[5] KOVAŘÍK, Pavel. *Automatizace některých pracovních postupů při využití matematických modelů generalizace*. Brno : s.n., 1980. 20 s. Vojenská akademie Antonína Zápotockého. Autoreferát dizertační práce.

[6] JANOŠEC, Josef. *Příspěvek k systémovému řešení metod automatizované kartografické generalizace*. Brno : s.n., 1984. 20 s. Vojenská akademie Antonína Zápotockého. Autoreferát dizertační práce.

[7] ŠTAMPACH, Radim. *Automatizovaná kartografická generalizace stavebních objektů z katastrálních map do map středních měřítek*. Brno, 2006. 97 s. Masarykova univerzita v Brně, Přírodovědecká fakulta. Vedoucí diplomové práce Karel Staněk. Dostupný z WWW: <http://is.muni.cz/th/63780/prif_m/>.

[8] SESTER, Monika. Generalization based on least squares adjustment. [online]. *International Archives of Photogrammetry and Remote Sensing*. 2000, vol. XXXIII, Part B4 [cit. 2008-06-20]. Dostupný z WWW: <http://www.ikg.uni-hannover.de/publikationen/publikationen/2000/Sester_Amsterdam.pdf>.

[9] BAYER, Tomáš. Kartografická generalizace stavebních objektů prostřednictvím 2D množinových operací. *Geodetický a kartografický obzor*. 2006, roč. 52, č. 12, s. 209-213.

[10] BRENNER, Claus, SESTER, Monika. *Cartographic generalization using primitives and constrains*. [online]. 2005 [cit. 2008-06-20]. Dostupný z WWW: <http://www.ikg.uni-hannover.de/publikationen/publikationen/2005/ICC_2005_Brenner.pdf>.

[11] MORAVEC, Dalibor. Geoinformatické modelování s topologickým generalizačním algoritmem. In *Sborník 14. kartografické konference : Úloha kartografie v geoinformační společnosti, Plzeň, 11.-13. Září 2001*. Sborník [online]. 2001 [cit. 2008-06-20]. ISBN 80-7082-781-5. Dostupný z WWW: <http://gis.zcu.cz/kartografie/konference2001/sbornik/moravec/moravec_referat.htm>.

[12] GOLD, Christopher, THIBAULT, David. *Map generalization by skeleton retraction*. [online]. 2004 [cit. 2008-06-20]. Dostupný z WWW: <http://www.skynet.ie/~sos/mapviewer/docs/Map_Generalization_by_Skeleton_Retraction.pdf>.

[13] *PostGIS manual* [online]. [2008] [cit. 2008-10-30]. Dostupný z WWW: <http://postgis.refractions.net/documentation/manual-1.3/>.

[14] DOLANSKÝ, Tomáš, BABICKÝ, Tomáš. *Základy kartografie : Kartografická generalizace* [online]. 2008 [cit. 2008-11-08]. Dostupný z WWW: <http://vrkoc.ujep.cz/engi/files/K_130_generalizace.pdf>.

[15] SUNDAY, Dan. *Intersections of Lines, Segments and Planes* [online]. c2001-2006 [cit. 2008-06-11]. Dostupný z WWW: <http://geometryalgorithms.com/algorithm_archive.htm>.

[16] PENÍZEK, Vít. *Generalizace*. 2007 [cit. 2008-09-20]. <Prezentace ve formátu pdf>.

Další prameny:

[17] *PostgreSQL* [online]. c1996-2008 [cit. 2008-11-30]. Dostupný z WWW: <http://www.postgresql.org/>.

[18] *PostGIS* [online]. [2008] [cit. 2008-11-30]. Dostupný z WWW: <http://www.postgis.org/>.

[19] *Eclipse* [online]. c2008 [cit. 2008-12-02]. Dostupný z WWW: <http://www.eclipse.org/org/>.

[20] *Java SE Downloads* [online]. c1994-2008 [cit. 2008-09-10]. Dostupný z WWW: <http://java.sun.com/javase/downloads/index.jsp>.

Seznam obrázků a tabulek

Seznam použitých zkratek

ACID	Atomicity, Consistency, Isolation, Durability
CVS	Concurrent Versions System
GIS	Geografický informační systém
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JAR	Java Archive
JDK	Java Development Kit
JDT	Java Development Tools
JRE	Java Runtime Environment
JVM	Java Virtual Machine
OGC	Open Geospatial Consortium
OS	Operační systém
PDE	Plug-in Development Environment
PL	Procedural Language
SDK	Software Development Kit
SQL	Structured Query Language
SRID	Spatial Reference IDentifier
SW	Software
UML	Unified Modeling Language
WFS	Web Feature Service
WKB	Well-Known Binary
WKT	Well-Known Text
WMS	Web Map Service
XML	eXtensible Markup Language

Přílohy

Příloha 1 – Zdrojový kód třídy `Rozdeleni`

Příloha 2 – Zdrojový kód třídy `GeneralizaceLinie`

Příloha 3 – Zdrojový kód třídy `GeneralizacePlochy`

Příloha 4 – Zdrojový kód třídy `Algoritmy`

Příloha 5 – Zdrojový kód třídy `Vektor`

Příloha 6 – Obsah přiloženého CD

Příloha 1 – Zdrojový kód třídy `Rozdeleni`

```
package cz.hrncirova.diplomka;

import java.util.*;
import org.postgis.*;

//Trida Rozdeleni
public class Rozdeleni {

//Funkce generalizace
public static Iterator generalizace(String [] g,String alg, double h,
                                    int p)
{
  //Seznam geografickych objektu, ktery je vracen
  ArrayList<LineString> out = new ArrayList<LineString>();
  ArrayList<Polygon> out2 = new ArrayList<Polygon>();

  try
  {
      //Seznam nactenych objektu LineString
      ArrayList<LineString> list = new ArrayList<LineString>();

      //Seznam nactenych objektu LinearRing
      ArrayList<LinearRing> listr = new ArrayList<LinearRing>();

      //Seznam všech nactenych bodu
      ArrayList<Point> vsechnybody = new ArrayList<Point>();

      //Seznam nactenych objektu Polygon
      ArrayList<Polygon> poly = new ArrayList<Polygon>();

      //Nacitani geografickych objektu z databaze
      for(String tmp : g)
      {
        PGgeometry geom = new PGgeometry(tmp);

        //Objekt MultiLineString
        if( geom.getGeoType() == 5 )
        {
            MultiLineString ml = (MultiLineString) geom.getGeometry();
                    for( int u = 0; u < ml.numLines(); u++ )
                    {
                      LineString ls = ml.getLine(u);

                      //ulozeni jednotlivych LineString do seznamu
                      list.add(ls);

                        for(int i=0; i<ls.numPoints();i++)
                        {
                          Point pt=ls.getPoint(i);

                          //ulozeni jednotlivych bodu do seznamu
                          vsechnybody.add(pt);

                        }
                    }
```

```
//Objekt LineString
}else if(geom.getGeoType() == 2)
{
    LineString ls = (LineString) geom.getGeometry();

    //ulozeni jednotlivych LineString do seznamu
    list.add(ls);

    for(int i=0; i<ls.numPoints();i++)
    {
      Point pt=ls.getPoint(i);

      //ulozeni jednotlivych bodu do seznamu
      vsechnybody.add(pt);
    }

//Objekt MultiPolygon
}else if(geom.getGeoType() == 6)
{
    MultiPolygon mp = (MultiPolygon)geom.getGeometry();

    for( int u = 0; u < mp.numPolygons(); u++ )
    {
       Polygon pl = mp.getPolygon(u);

       //vlozeni polygonu do seznamu
       poly.add(pl);

          for(int v=0; v <pl.numRings();v++ )
          {
            LinearRing rng = pl.getRing(v);

            //vlozeni kruznice do seznamu
            listr.add(rng);

            for(int i=0; i<rng.numPoints();i++)
            {
               Point pt=rng.getPoint(i);

               //vlozeni bodu do seznamu bodu
               vsechnybody.add(pt);
            }
          }
    }

//Objekt Polygon
}else if(geom.getGeoType() == 3)
{
    Polygon pl =(Polygon) geom.getGeometry();

    //vlozeni polygonu do seznamu
    poly.add(pl);

    for(int v=0; v <pl.numRings();v++ )
    {
       LinearRing rng = pl.getRing(v);

       //vlozeni kruznice do seznamu
       listr.add(rng);
```

```
                for(int i=0; i<rng.numPoints();i++)
                {
                   Point pt=rng.getPoint(i);

                   vsechnybody.add(pt);

                }
             }
          }
       }


       //Do generalzace vstupuji liniova data
       if(list.size()>0)
       {
          //Volani funkce pro generalizaci linii
          out = GeneralizaceLinie.linie(list, vsechnybody,alg,h,p);
       }


       //Do generalizace vstupuji plosna data
       if(poly.size()>0)
       {
         //Volani funkce pro generalizaci ploch
         out2 = GeneralizacePlochy.plochy(poly,listr,vsechnybody,alg,h,p);
       }

  }
  catch( Exception e )
  {
    e.printStackTrace();
  }

  if(out.size()>0)
  {
     return out.iterator();
  }else{
     return out2.iterator();
  }

}

}
```

Příloha 2 – Zdrojový kód třídy `GeneralizaceLinie`

```
package cz.hrncirova.diplomka;

import java.util.*;
import org.postgis.*;

public class GeneralizaceLinie
{
  public static ArrayList<LineString> linie(ArrayList<LineString> list,
              ArrayList<Point> vsechnybody, String alg, double h, int p)
  {
      //Seznam poctu vyskytu jednotlivych bodu
      ArrayList <Integer> b = new ArrayList <Integer>();

      //urceni kolikrat je ktery bod obsazen v datech
      for(int i=0;i<vsechnybody.size();i++)
      {
         int a=0;
         for(int j=0;j<list.size();j++)
         {
             if(Algoritmy.linetopoint(list.get(j)).contains
               (vsechnybody.get(i)))
            {
               a=a+1;
            }
         }
         b.add(a);
      }

      //Pomocna hodnota k urceni pozice v seznamu b
      int n=0;

      //Seznam linii k testovani vzniku pruseciku a zaroven seznam
      //aktuálně generalizovanych linii
      ArrayList<LineString> testvsel=new ArrayList<LineString>();

      //Seznam konvexních obalek
      ArrayList<LineString> obalky=new ArrayList<LineString>();

      //Pruchod vsech linii za ucelem generalizace kazde z nich
      for(LineString l : list)
      {
        //Seznam bodu linie
        ArrayList<Point> body;

        //Seznam poradi uzlu v ramci linie
        ArrayList <Integer> nodesj = new ArrayList <Integer>();

        //Prevedeni linie na seznam bodu
        body=Algoritmy.linetopoint(l);

        //Body generalizovane linie
        ArrayList <Point> genlin=new ArrayList<Point>();

        //Hledani uzlovach bodu
        int j=0;
        nodesj.add(j);
```

```
for( j=1;j<body.size()-1;j++)
{
    if(b.get(j+n)>1 ) //Pokud existuje bod vice nez jednou
    {
       nodesj.add(j);
    }
}

j=body.size()-1;
nodesj.add(j);

//Cyklus zpracovava všechny useky linie
for(int m=0; m<nodesj.size();m++)
{

  //Body useku
  List <Point> nodes = new ArrayList <Point>();

  //Body useku po generalizaci
  ArrayList<Point> cara = new ArrayList<Point>();


  if(m<nodesj.size()-1)
  {
     nodes=body.subList(nodesj.get(m),nodesj.get(m+1)+1);
  }

  if(m==nodesj.size()-1){
    nodes=body.subList(nodesj.get(m),body.size());
  }

  //Vytvoreni konvexni obalky kolem linie
  obalky.add(Algoritmy.hull(nodes));

  //Generalizace dle vybraneho algoritmu
  if(alg.equalsIgnoreCase("dp"))
  {
    SortedMap<Integer,Point> nove = new TreeMap<Integer,Point>();
    nove = Algoritmy.generalizacedp(nodes,nove,h,0,nodes.size()-
                                    1,obalky,testvsel);
    ArrayList<Point> nov = new ArrayList<Point>(nove.values());
    cara=nov;

  }else if(alg.equalsIgnoreCase("lang"))
  {
    cara = Algoritmy.generalizacel(nodes, h, p,obalky,testvsel);

  }else if(alg.equalsIgnoreCase("whyatt"))
  {
    cara = Algoritmy.generalizacew(nodes, h,obalky,testvsel);

  }else if(alg.equalsIgnoreCase("reumann"))
  {
    cara = Algoritmy.generalizacer(nodes, h,obalky,testvsel);
  }

  //Pridani bodu generalizovaného useku do linie
  genlin.addAll(cara);
```

```
            if(m!=nodesj.size()-1)
            {
              //aby nebyl uzlovy bod vlozen dvakrat
              genlin.remove(genlin.size()-1);
            }
          }
          n=n+j+1;

          //Prevedeni seznamu bodu na objekt typu LineString
          Point[] c = new Point[genlin.size()];
          genlin.toArray(c);

          LineString genlinie = new LineString(c);
          testvsel.add(genlinie);
        }

        return testvsel;
    }

}
```

Příloha 3 – Zdrojový kód třídy `GeneralizacePlochy`

```
package cz.hrncirova.diplomka;

import java.util.*;
import org.postgis.*;

public class GeneralizacePlochy
{
    public static ArrayList<Polygon> plochy(ArrayList<Polygon> poly,
                                            ArrayList<LinearRing> list,
                                            ArrayList<Point> vsechnybody,
                                            String alg,double h, int p)
    {

      //Seznam poctu vyskytu jednotlivych bodu
      ArrayList <Integer> b = new ArrayList <Integer>();

      //urceni kolikrat je ktery bod obsazen v datech
      for(int i=0;i<vsechnybody.size();i++)
      {
        int a=0;
        for(int j=0;j<list.size();j++)
        {
            if(Algoritmy.linetopoint(list.get(j)).contains(
                                                vsechnybody.get(i)))
            {
               a=a+1;
            }
        }
        b.add(a);
      }

      //Pomocna hodnota k urceni pozice v seznamu b
      int n=0;

      //Obsahuje klic, ktery identifikuje puvod linie, hodnotami jsou
      //všechny linie na které se rozpadnou polygony
      Map<ArrayList<Integer>,List<Point>> vse=new HashMap<ArrayList<
                                            Integer>,List<Point>>();


      //Obsahuje seznam obalek linii
      ArrayList<LineString> test=new ArrayList<LineString>();

      //Obsahuje seznam linii k testovani pruseciku (aktualizac po
      //generalizaci)
      ArrayList<LineString> test2=new ArrayList<LineString>();

      //Obsahuje seznam oblasti, ktere s zadnou jinou nesousedi
      ArrayList<ArrayList<Integer>> ostrov=new ArrayList<ArrayList<
                                                    Integer>>();

      //Pruchod vsech polygonu za ucelem rozlozeni na linie
      Integer pid=0;  //poradi polygonu
      Integer nid=0;  //poradi useku mezi uzly

          for(Polygon pol : poly)
          {
              Integer lid=0;  //poradi kruznice v ramci polygonu
```

```
for(int l=0; l< pol.numRings(); l++)
{

 //Hledani uzlu

 //Seznam poradi uzlu v ramci linie
 ArrayList <Integer> nodesj = new ArrayList <Integer>();

 //Seznam bodu linie
 ArrayList<Point> body = new ArrayList<Point>();

 body=Algoritmy.linetopoint(pol.getRing(l));
 int j=0;

    if(b.get(j+n)>1 && b.get(j+n)>b.get(j+1+n))
    {
       nodesj.add(j);
    }

    for(j=1;j<body.size()-1;j++)
    {
       if(b.get(j+n)==2 && (b.get(j+n)>b.get(j+1+n)|
                            b.get(j+n)>b.get(j-1+n)))
       {
            nodesj.add(j);
       }

       if(b.get(j+n)>2 && b.get(j+n)>b.get(j+1+n) &&
                            b.get(j+n)>b.get(j-1+n))
       {
            nodesj.add(j);
       }
    }


    j=body.size()-1;

    if(b.get(j+n)>1 && b.get(j+n)>b.get(j-1+n))
    {
       nodesj.add(j);
    }

    int o=0;  // hodnota 0 nebo 2 - není soused

    if(nodesj.size()==1)     //existuje jeden uzel
    {
       //vzdalenosti
       ArrayList<Double> d=new ArrayList<Double>();

       for(int u=0;u<body.size();u++)
       {
          d.add(Algoritmy.delka(body.get(nodesj.get(0)),
                                      body.get(u)));
       }

       ArrayList<Integer> dm=new ArrayList<Integer>(
                        Algoritmy.maximum(d).values());
       nodesj.add(dm.get(dm.size()-1));
```

```
}

if(nodesj.size()==0)  //pokud neexistuje zadny uzel
{
   o=2;

   //vzestupne souradnice x a poradi bodu
   SortedMap<Double, Integer> dlex=new TreeMap<
                                    Double, Integer>();
   //vzestupne soradnice y a poradi bodu
   SortedMap<Double, Integer> dley=new TreeMap<
                                    Double, Integer>();

   for(int u=0;u<body.size();u++)
   {
       dlex.put(body.get(u).getX(),u);
       dley.put(body.get(u).getY(),u);
   }


   //Hodnoty x
   ArrayList<Double> x= new ArrayList<Double>(
                                        dlex.keySet());

   //Hodnoty y
   ArrayList<Double> y= new ArrayList<Double>(
                                        dley.keySet());

   //Serazeni dle umisteni v polygonu
   SortedMap<Integer,Integer> t=new TreeMap<
                                    Integer,Integer>();

   //zajima nas pouze klicova hodnota
   t.put(dlex.get(x.get(0)), 0);
   t.put(dlex.get(x.get(x.size()-1)), 0);
   t.put(dley.get(y.get(0)), 0);
   t.put(dley.get(y.get(y.size()-1)),0);

   nodesj.addAll(t.keySet());
}

int pom=0;

if((body.get(nodesj.get(0))).equals(body.get(
                     nodesj.get(nodesj.size()-1))))
{
   pom=1; //pocatecni uzel se rovna koncovemu
}

//Cyklus prochazi vsechny linie daneho polygonu a
//uklada je pod jedinecnym klicem do kontejneru
for(int m=0; m<nodesj.size();m++)
{
   //Body linie
   List <Point> nodes = new ArrayList <Point>();

   //Klic
   ArrayList<Integer> k=new ArrayList<Integer>();

   if(m<nodesj.size()-1)
   {
```

```
   nodes=body.subList(nodesj.get(m),
                                 nodesj.get(m+1)+1);
}

if(m==nodesj.size()-1 && pom==0)
{
   nodes.addAll(body.subList(nodesj.get(m),
                                      body.size()));
    nodes.addAll(body.subList(1,nodesj.get(0)+1));

}

// Linie v kontejneru jiz je
if(vse.containsValue(nodes))
{
      k=getKey(vse,nodes);
      vse.remove(k); //odstraneni puvodnich udaju
      k.add(pid);    //doplneni klice o udaje z
                     //druheho polygonu
      k.add(lid);
      k.add(nid);
      k.add(0);      //udaj, ze se nejedna opacny
                      //smer bodu

      if(vse.containsKey(k)| k.size()>7)
      {
      }else{
          vse.put(k, nodes); //vlozeni noveho
                             //klice a seznamu bodu
      }

// Linie v kontejneru jiz je, ale v opacnem smeru
}else if(vse.containsValue(reverse(nodes)))
{
      k=getKey(vse,reverse(nodes));
      vse.remove(k); //odstraneni puvodnich udaju
      k.add(pid);    //doplneni udaju z druheho
                     //polygonu
      k.add(lid);
      k.add(nid);
      k.add(1);      // jedna se o opacny smer
                     //bodu

      if(vse.containsKey(k)| k.size()>7){
      }else{
            //vlozeni noveho klice a seznamu bodu
            vse.put(k, reverse(nodes));
      }
//Linie jeste neni vlozena do kontejneru
}else{
    // prvni polozna klice znaci poradi polygonu
    k.add(pid);

    // druha polozna klice znaci poradi kruznice
    //v polygonu
    k.add(lid);

    // treti polozna klice znaci poradi useku
    //mezi uzly
    k.add(nid);
```

```
                    // vlozeni klice a seznamu bodu linie
                    vse.put(k, nodes);

                    if(o==2)
                    {
                      //vlozeni klice linie oblasti, ktera
                      //nesousedi s jinou
                      ostrov.add(k);
                    }
                }
                nid++;
            }
            n=n+j+1;
            lid++;
        }
        pid++;
}

Iterator<List<Point>> idv=vse.values().iterator();

//Tvorba obalek linii
while(idv.hasNext()
{
    List<Point> i=new ArrayList<Point>();
    i=idv.next();
    test.add(Algoritmy.hull(i));//vlozeni obalky do seznamu

    //Prevod seznamu bodu linie na LineString
    Point[] c= new Point[i.size()];
    i.toArray(c);
    LineString lstr=new LineString(c);
    test2.add(lstr);
}

Iterator<ArrayList<Integer>> id=vse.keySet().iterator();
int t=0;

//Cyklus vola generalizacni algoritmus
while(id.hasNext())
{
  ArrayList<Integer> i=new ArrayList<Integer>();
  i= id.next();

  if(vse.get(i).size()>0){

     //Body genealizovane linie
     ArrayList<Point> nove = new ArrayList<Point>();

     if(alg.equalsIgnoreCase("dp"))
     {
       SortedMap<Integer,Point> cara = new TreeMap<Integer,
                                                   Point>();
       cara = Algoritmy.generalizacedp(vse.get(i),cara,h,0,
                              vse.get(i).size()-1,test,test2);
       ArrayList<Point> nov = new ArrayList<Point>(
                                              cara.values());
       nove=nov;
     }else if(alg.equalsIgnoreCase("lang"))
     {
       nove = Algoritmy.generalizacel(vse.get(i), h, p,test,
                                                    test2);
```

```
      }else if(alg.equalsIgnoreCase("whyatt"))
      {
        nove = Algoritmy.generalizacew(vse.get(i), h, test,
                                                        test2);
      }else if(alg.equalsIgnoreCase("reumann"))
      {
        nove = Algoritmy.generalizacer(vse.get(i), h, test,
                                                        test2);
      }

      vse.put(i,nove); // vlozeni bodu nove linie do kontejneru

      test2.remove(t); // odstranni puvodni linie
      Point[] c= new Point[nove.size()];
      nove.toArray(c);
      LineString lstr=new LineString(c);
      test2.add(t,lstr); // vlozeni generalizovane linie
    }
    t++;
}


//Nasleduje testovani prekrytu ostrovu s jinymi objekty
ArrayList<ArrayList<Integer>>it=new ArrayList<ArrayList<
                                        Integer>>(vse.keySet());

//Seznam ostrovnich oblasi, které protínají jinou oblast
ArrayList<Integer> por=new ArrayList<Integer>();

for(int f=0;f<it.size();f++)
{
  ArrayList<Integer> k=it.get(f);
  List<Point> po=vse.get(k);

  boolean pr=false;

  //V seznamu linii ostrovnich oblasti je linie s klicem k
  if(ostrov.contains(k))
  {
      for(int j=0;j<po.size()-1;j++)
      {
          pr= Algoritmy.prusecik(po.get(j), po.get(j+1),test2);

          //Existuje prusecik s ostrovni oblasti
          if(pr==true)
          {
              por.add(k.get(0));
              break;
          }
      }



      if(por.contains(k.get(0)))
      {
           vse.remove(k); //odstraneni prekryvajiciho prvku
      }

  }

}
```

```
 //Seznam generalizovaných oblasti
 ArrayList<Polygon> out=new ArrayList<Polygon>();

//Poskladani linii zpet do polygonu
for(Integer u=0;u<poly.size();u++){
    int nr=poly.get(u).numRings();
    LinearRing [] lr= new LinearRing[nr];
    int pom=0;

   for(int j=0;j<nr;j++)
   {
      SortedMap<Integer,List<Point>> linie = new TreeMap<
                                          Integer,List<Point>>();
      Iterator<ArrayList<Integer>> id2=vse.keySet().iterator();

     /Prochazi se vsechny  linie
     while(id2.hasNext())
     {
         ArrayList<Integer> i= id2.next();

        //Vsechny linie ze steneho polygonu a kruznice
        if(i.get(0).equals(u) && i.get(1).equals(j))
        {
              // klicem je poradi linie
              linie.put(i.get(2), vse.get(i));
              //linie se srovnaji za sebe ve spravnem poradi
        }

        if(i.size()>3)  //linie je soucasti dvou polygonu
        {

           //Vsechny linie ze steneho polygonu a kruznice
           if(i.get(3).equals(u)&& i.get(4).equals(j))
           {
              if(i.get(6)==0)
              {
                linie.put(i.get(5), vse.get(i));
                //umisteni linie dle klice
              }else  //prevraceny smer
              {
                linie.put(i.get(5), reverse(vse.get(i)));
                //umisteni linie dle klice
              }
           }

        }
     }

     Iterator<List<Point>> iter3=linie.values().iterator();

     //Seznam bodu všech linii z jednoho polygonu
     ArrayList<Point> ring=new ArrayList<Point>();

     while(iter3.hasNext())
     {
           List<Point> temp=iter3.next();
           ring.addAll(temp);
     }

     if(ring.isEmpty())
     {
```

```
            pom=1; // zadne body z oblasti
      }else
      {
            //Prevedeni bodu na LinearRing
            Point[] c = new Point[ring.size()+1];
            ring.toArray(c);
            c [c.length-1]=c [0];
            lr [j]=new LinearRing(c);
      }
    }

    if(pom==0)
    {
            //Prevedeni kruznic na polygony
            Polygon genpol=new Polygon(lr);

            if(genpol.numPoints()>2){
              out.add(genpol);
            }
    }
}

//Test zda polygon nelezi uvnitr jineho polygonu
ArrayList<ArrayList<Point>> mbox=new ArrayList<ArrayList<
                                                    Point>>();
//Seznam nevyhovujících oblasti
ArrayList<Integer> vyrad=new ArrayList<Integer>();

//Tvorba opsaného obdélníka

for(int g=0;g<out.size();g++)
{
  mbox.add(Algoritmy.minmax(out.get(g)));
}

for(int f=0;f<mbox.size();f++)
{
  ArrayList<Point> box1=mbox.get(f);

  for(int z=0;z<mbox.size();z++)
  {
     ArrayList<Point> box2=mbox.get(z);

     //Souradnice minmax boxu
     Double v1=box1.get(0).getX();
     Double v2=box1.get(1).getX();
     Double v3=box1.get(0).getY();
     Double v4=box1.get(2).getY();

     Double u1=box2.get(0).getX();
     Double u2=box2.get(1).getX();
     Double u3=box2.get(0).getY();
     Double u4=box2.get(2).getY();

     //Pokud minmax box uvnitr jineho
     if(u1>v1 && u2<v2 && u3>v3 && u4<v4)
     {
        //Pocet pruseciku s okolni oblasti
        int pocet= Algoritmy.prusecik(out.get(z).getPoint(0),
                              out.get(z).getPoint(2),out.get(f));
        if(pocet%2!=0) //existuje sudy pocet pruseciku
```

```
                    {
                       vyrad.add(z); //vyrad oblast z
                    }
                }

            }
          }

          for(int v=0;v<out.size();v++)
          {
            if(vyrad.contains(v))
            {
                  // odstraneni polygonu lezicich uvnitr jineho
                  out.remove(v);
            }
          }

          return out;
    }


    //Funkce, ktera vrati klic z kontejneru na zaklade zadani hodnoty
    public static ArrayList<Integer> getKey(Map<ArrayList<Integer>,
                                            List<Point>> m,List<Point> l)
    {
         Iterator<ArrayList<Integer>> iter=m.keySet().iterator();
         ArrayList<Integer> out=new ArrayList<Integer>();

         while(iter.hasNext())
         {
             ArrayList<Integer> c=iter.next();
             if(m.get(c).equals(l))
             {
                  out=c;
             }
         }
         return out;
    }

    //Funkce otoci poradi bodu seznamu
    public static List<Point> reverse(List<Point> l)
    {
        List<Point> out= new ArrayList<Point>();
        for(int i=l.size()-1; i>=0;i--)
        {
           out.add(l.get(i));
        }
        return out;
    }

}
```

Příloha 4 – Zdrojový kód třídy `Algoritmy`

```
package cz.hrncirova.diplomka;

import org.postgis.*;
import java.util.*;

public class Algoritmy {

//Funkce provadi generalizaci Douglas-Peuckerovym algoritmem
public static SortedMap<Integer,Point> generalizacedp(List<Point> body,
                                        SortedMap<Integer,Point> nove,
                                        double h, int s, int k,
                                        ArrayList<LineString> obalky,
                                        ArrayList<LineString> testvse)
{

      ArrayList<Double> delkyk = new ArrayList<Double>();
      int u = 0;

      //Pokud je poradi pocatecniho bodu nizsi nez koncoveho nasleduje
      //rekurzivni volani funkce
      if(s < k-1 )
      {

         //Hledani nejvzdalenejsiho bodu od usecky predstavovane s, k:

         //Vypocet delek jednotlivych bodu mezi s a k
         for(int i = s+1; i < k ; i++)
         {

            delkyk.add(kolmavzd(body.get(s),body.get(k),body.get(i))
         }

         //Vzestupne serazeni urcenych delek
         Map<Double, Integer> max = maximum(delkyk);

         //seznam delek
         ArrayList<Double> dk=new ArrayList<Double>(max.keySet());

         //puvodni poradi k novemu seznamu
         ArrayList<Integer> dki=new ArrayList<Integer>(max.values());

         //Pokud je nejvetsi delka vetsi nez delkove kriterium pokracuje
         //nasledujici blok
         if(dk.get(dk.size()-1)>=h)
         {
            // pomocna promenna, pokud je rovna nule existuje prusecik
            int pom=0;

            //Cyklus probiha v pridade, ze existuje nechteny prusecik.
            // Dalsi podminkou je existence nezkoumane vzdalenosti ze
            //seznamu vzdalenosti bodu
            while(pom == 0 && dki.size()>0)
            {
                  int i = dki.get(dki.size()-1);
                  u = s+i+1; // poradi bodu v ramci cele linie

                  //bod, ktery ma byt vlozen do linie
                  Point tmp = body.get(u);
```

```
                    //Testovani vzniku pruseciku
                    boolean prus = prusecik(body.get(s),tmp,new ArrayList<
                                    Point>(nove.values()),obalky,testvse);
                    boolean prus2 = prusecik(tmp,body.get(k),new ArrayList<
                                    Point>(nove.values()),obalky,testvse);

                    //prusecik neexistuje
                    if(prus == false && prus2 == false)
                    {
                          nove.put(u ,tmp); //bod je vlozen do nove linie
                          pom=1;
                    }
                    else  //prusecik existuje
                    {
                          dki.remove(dki.size()-1);
                          //vyrazeni vzdalenosti ze seznamu
                    }
               }


               //Opakovane volani funkce na dve nove vznikle usecky
               nove.putAll(generalizacedp(body,nove,h,s,u,obalky,testvse));
               nove.putAll(generalizacedp(body,nove,h,u,k,obalky,testvse));
            }

         }

      //Vlozeni prvniho a posledniho bodu linie
      nove.put(0, body.get(0));
      nove.put(body.size()-1, body.get(body.size()-1));

      return nove;
}

//Funkce provadi generalizaci Reumann-Witkamovym algoritmem
public static ArrayList<Point> generalizacer(List<Point> bodyl, double h,
                                             ArrayList<LineString> obalky,
                                            ArrayList<LineString> testvse)
{
      ArrayList<Point> body = new ArrayList<Point>(bodyl);
      int s = 0;                //pocatecni bod linie
      int k = body.size()-1;    //koncovy bod linie

      //seznam bodu k odstraneni
      ArrayList<Integer> odstranit = new ArrayList<Integer>();

      //Dokud je pocatecni bod hrany, ktera je osou koridoru mensi nez
      //koncovy bod linie...
      while(s < k)
      {

         //Seznam obsahujici pocatecni body hran, ktere protina koridor
         ArrayList<Integer> strana = new ArrayList<Integer>();

         //Cyklus prochazi body vsech hran od hrany tvorici koridor do
         //konce linie
         for(int i = s+2; i <= k; i++ )
         {
            double v = kolmavzd(body.get(s),body.get(s+1),body.get(i-1));
            double w = kolmavzd(body.get(s),body.get(s+1),body.get(i));
```

```
            //Pokud je vzdalenost pocatecniho vrcholu od osy koridoru
            //mensi nez sirka koridoru a vzdalenost koncoveho bodu hrany
            //vetsi nebo naopak, koridor hranu protina
            if((v < (h/2) && w > (h/2)) |(v > (h/2) && w < (h/2)))
            {
                  strana.add(i-1);
            }
         }

         //Existuji protnute hrany
         if(strana.size() > 0)
         {
            for(int i = s+1; i < strana.get(0);i++)
            {
                  odstranit.add(i); //poradi bodu mezi hranou tvorici
                                    //koridor a prvni protnutou hranou se
                                    // uklada, pozdeji jsou tyto body
                                    //odstraneny
            }

            //novym pocatkem osy koridoru je pocatecni bod protnute hrany
            s = strana.get(0);
         } else //Neexistji protnute hrany
         {
            for(int i = s+1; i < k;i++)
            {
              odstranit.add(i);
              //poradi bodu mezi hranou tvorici koridor a poslednim bodem
              //linie se uklada, pozdeji jsou tyto body odstraneny
            }

            s = k;  //za pocatek je dosazen koncovy bod a vypocet konci
         }

      }

      //Proces odstraneni bodu vyrazenych algoritmem
      for(int i = odstranit.size()-1; i>=0;i--)
      {
            int t = odstranit.get(i);
            Point p = body.get(t);
            body.remove(t);  //odstraneni bodu

            //Vypocet pruseciku
            boolean prus = prusecik(body.get(t),body.get(t-1),body,
                                                      obalky,testvse);

            //Pokud existuje prusecik, bod je zpet vlozen do linie
            if(prus == true)
            {
                body.set(t,p);
            }
      }

      return body;
}
```

```
//Funkce provadi generalizaci Langovym algoritmem
public static ArrayList<Point> generalizacel(List<Point> bodyl, double h,
                                        int p,ArrayList<LineString> obalky,
                                        ArrayList<LineString> testvse)
{
    ArrayList<Point> body = new ArrayList<Point>(bodyl);

    //Try-catch blok zachytava vyjimku, kterou zpusobi p vetsi nez pocet
    //bodu linie
    try{
         ArrayList <Integer> odstranit = new ArrayList <Integer>();
         int s = 0;  //poradi pocatecniho bodu daneho kroku generalizace
         int k = 0;

         //Tento cyklus postupne projde vsechny body linie
         while(s < body.size()-1)
         {
            ArrayList <Integer> vne = new ArrayList <Integer>();
            ArrayList <Integer> uvnitr = new ArrayList <Integer>();

            //Jeden krok generalizace predstavuje nasledujici cyklus
            for(int i = s+1; i<s+p-k; i++)
            {

              //Vypocet vzdalenosti testovaneho bodu od spojnice prvniho
              //a posledniho bodu kroku
              double d = kolmavzd(body.get(s),body.get(s+p-k),
                                                        body.get(i));

              if(d >= (h/2))
              {
                  vne.add(i);       //Bod lezi vne koridoru
              }
              else{
                  uvnitr.add(i);    //Bod lezi v koridoru
              }
            }

            //Pokud lezi nejaky bod vne, zmeni se koncovy bod kroku
            if(vne.size()> 0)
            {
               k = k+1;
            } else   //Zadny pod nelezi vne koridoru
            {
               //ulozi se body,ktere budou odstraneny
               odstranit.addAll(uvnitr);

            //nove nastaveni pocatecniho a koncoveho bodu pro dalsi krok
               s = s+p-k;
               k = 0;

            }

            //Kdyz je p vetsi nez zbyvajici pocet bodu v linii
            if(s > body.size()-1-p)
            {
               //misto p se pouzije zbyvajici pocet bodu
               p = body.size()-1-s;
            }

         }
```

```
            //Proces odstraneni bodu vyrazenych algoritmem
            for(int i = odstranit.size()-1; i>=0;i--)
            {
               int t = odstranit.get(i);
               Point po = body.get(t);
               body.remove(t); //odstraneni bodu

               //Vypocet pruseciku
               boolean prus = prusecik(body.get(t),body.get(t-1),body,
                                                        obalky,testvse);

               //Pokud existuje prusecik, bod je zpet vlozen do linie
               if(prus == true)
               {
                  body.set(t,po);
               }
            }

         }
         catch(Exception e){
               System.out.println("Zadej p mensi nez "+(body.size()));
         }

         return body;
}

//Funkce provadi generalizaci Whyattovym algoritmem
public static ArrayList<Point> generalizacew(List<Point> bodyl, double
                                        smin,ArrayList<LineString> obalky,
                                            ArrayList<LineString> testvse)
{
    ArrayList<Integer> odstranit = new ArrayList<Integer>();
    ArrayList<Point> body = new ArrayList<Point>(bodyl);

    //Zatimco budou v seznamu body k odstraneni probiha tento cyklus
    Do
    {
         odstranit.clear();

         //Z kazdych trech po sobe jdoucich bodu je pocitan obsah
         for(int i = 0; i < body.size()-2;i=i+2)
         {
            double o;
            o = obsah(body.get(i),body.get(i+1),body.get(i+2));

            if(o < smin)
            {
              //pokud obsah nevyhovuje kriteriu, bod bude vyrazen
              odstranit.add(i+1);
            }
         }

         //Proces odstraneni bodu vyrazenych algoritmem
            for(int i = odstranit.size()-1; i >= 0; i-- )
            {
                  int t = odstranit.get(i);
                  Point p = body.get(t);
                  body.remove(t);  //odstraneni bodu
```

```
                    //Vypocet pruseciku
                    boolean prus = prusecik(body.get(t),body.get(t-1),
                                                    body,obalky,testvse);


                    //Pokud existuje prusecik, bod je zpet vlozen do linie
                    if(prus == true)
                    {
                          body.set(t,p);
                    }
              }

       }while(odstranit.size() > 0);

      return body;
}

//Funkce prevadi objekt typu LinearRing na seznam bodu
public static ArrayList<Point> linetopoint(LinearRing ls)
{
      ArrayList<Point> body = new ArrayList<Point>();

      for( int p = 0; p < ls.numPoints(); p++ )
      {
         Point pt = ls.getPoint(p);
         body.add(pt);
      }

      return body;
}

//Funkce prevadi objekt typu LineString na seznam bodu
public static ArrayList<Point> linetopoint(LineString ls)
{
      ArrayList<Point> body = new ArrayList<Point>();

      for(int p = 0; p < ls.numPoints(); p++)
       {
         Point pt = ls.getPoint(p);
         body.add(pt);
       }

      return body;
}

//Funkce pro vypocet vzdalenostin dvou bodu
public static double delka(Point a, Point b)
{
      return Math.sqrt((a.getX()-b.getX())*(a.getX()-b.getX())+
                  (a.getY()-b.getY())*(a.getY()-b.getY()));
}

//Funkce urcuje kolmou vzdalenost bodu od spojnice dvou bodu
public static double kolmavzd(Point a, Point b, Point c)
{
      double s;
      double obsah;
      double v;

      //Vychazi z Heronova vzorce
      s =(delka(a,b)+ delka(b,c)+ delka(c,a))/2;
```

```
        obsah = Math.sqrt(s*(s-delka(a,b))*(s-delka(b,c))*
                                                    (s-(delka(c,a))));
        v = 2*obsah/delka(a,b);

        return v;
}

//Funkce seradi vzestupne hodnoty typu double
public static SortedMap<Double,Integer> maximum (ArrayList <Double> a)
{
        //Klicem je vstupni hodnota.
        //Druhym udajem je poradi ze vstupniho seznam
        SortedMap<Double,Integer> m = new TreeMap<Double,Integer>();

        for(int i = 0 ; i < a.size(); i++ )
        {
              m.put(a.get(i),i);
        }
        return m;
}

//Funkce pro urcene obsahu trojuhelnika urceneho tremi body
public static double obsah(Point a, Point b, Point c)
{
        double s;
        double obsah;

        //Heronuv vzorec
        s = (delka(a,b)+ delka(b,c)+ delka(c,a))/2;
        obsah = Math.sqrt(s*(s-delka(a,b))*(s-delka(b,c))*
                                                    (s-(delka(c,a))));
        return obsah;
}

//Funkce urcuje souradnice minimalniho opsaneho obdelnika
public static ArrayList<Point> minmax(Polygon poly)
{
        ArrayList<Point> box = new ArrayList<Point>();

        //Udaje ukladane do maxx a maxy budu razeny vzestupne dle klice
        SortedMap<Double, Point> maxx=new TreeMap<Double, Point>();
        SortedMap<Double, Point> maxy=new TreeMap<Double, Point>();

        //Cyklus prochazi vsechny body polygonu
        for(int j = 0;j < poly.numPoints();j++)
        {
            Point p = poly.getPoint(j);
            maxx.put(p.getX(),p);
            //klicem je souradnice x, hodnotou odpovidajici bod

            maxy.put(p.getY(),p);
            //klicem je souradnice y, hodnotou odpovidajici bod
        }

        //Serazene seznamy x-ovych a y-ovych hodnot
        ArrayList<Double> x = new ArrayList<Double>(maxx.keySet());
        ArrayList<Double> y = new ArrayList<Double>(maxy.keySet());

        box.add(new Point(x.get(0),y.get(0)));
        //do seznamu je ulozen bod o minimalnich souradnicich x a y
```

```
      box.add(new Point(x.get(x.size()-1),y.get(0)));
      //maximalni x a mniimalni y

      box.add(new Point(x.get(x.size()-1),y.get(y.size()-1)));
      //maximalni x a maximalni y

      box.add(new Point(x.get(0),y.get(y.size()-1)));
      //minimalni x a maximalni y

      box.add(new Point(x.get(0),y.get(0)));
      //posledni bod shodny s prvnim

      return box;
}

//Funkce testuje vznik pruseciku
//Funkce pro urceni nechteneho pruseciku behem generalizacniho algoritmu

public static boolean prusecik (Point p0, Point p1, ArrayList<Point>
                                test, ArrayList<LineString> obalky,
                                ArrayList<LineString> testvse)
{
    // nastavena hodnota odpovida neexistenci pruseciku
    boolean i = false;

    //Vektor testovane hrany
    Vektor u = new Vektor(p1.getX()-p0.getX(),p1.getY()-p0.getY());

    //Testovani se vsemi hranami dane linie
    for(int j = 0;j < test.size()-1;j++)
    {
       Vektor v = new Vektor(test.get(j+1).getX()-test.get(j).getX(),
                             test.get(j+1).getY()-test.get(j).getY());

       Vektor w = new Vektor(p0.getX()-test.get(j).getX(),
                                      p0.getY()-test.get(j).getY());

       //Nema smysl resit prusecik sousednich hran
       if(p0.equals(test.get(j))|p0.equals(test.get(j+1))|p1.equals(
                               test.get(j))|p1.equals(test.get(j+1)))
       {
       }else{
            double s;
            double t;

            s = (v.getU2()*w.getU1()-v.getU1()*w.getU2())/(v.getU1()*
                                      u.getU2()-v.getU2()*u.getU1());
            t = (u.getU1()*w.getU2()-u.getU2()*w.getU1())/(u.getU1()*
                                      v.getU2()-u.getU2()*v.getU1());

            //Pokud prusecik existuje
            if((s > 0 && s < 1) &&(t > 0 && t < 1))
            {
                  i = true;
                  break;
            }
       }
     }
```

```
      //Pokud prusecik neexistuje dochazi k dalsimu testovani s ostatními
      //objekty
      if(i == false)
      {
         for(int k = 0; k < obalky.size();k++){
             LineString body = obalky.get(k);

             for(int j = 0;j < body.numPoints()-1;j++)
             {
                 Vektor v = new Vektor(body.getPoint(j+1).getX()-
                         body.getPoint(j).getX(),body.getPoint(j+1).getY()
                                                 -body.getPoint(j).getY());
                 Vektor w = new Vektor(p0.getX()-body.getPoint(j).getX(),
                                       p0.getY()-body.getPoint(j).getY());

             //Nema smysl resit prusecik sousednich hran

             if(p0.equals(body.getPoint(j))|p0.equals(body.getPoint(j+1))|
                p1.equals(body.getPoint(j))|p1.equals(body.getPoint(j+1)))
             {
             }else{
                   double s;
                   double t;

                   s = (v.getU2()*w.getU1()-v.getU1()*w.getU2())/
                               (v.getU1()*u.getU2()-v.getU2()*u.getU1());
                   t = (u.getU1()*w.getU2()u.getU2()*w.getU1())/
                               (u.getU1()*v.getU2()-u.getU2()*v.getU1());

                   //Pokud existuje prusecik s obalkou
                   if((s > 0 && s < 1) && (t > 0 && t < 1))
                   {
                   //volani funkce pro vypocet pruseciku se vsemi liniemi
                     i = prusecik(p0,p1,testvse);
                         if(i == true)
                         {
                           //prusecik existuje a vypocet je prerusen
                           break;

                         }
                   }
             }
         }

         if(i == true)break;
       }

     }

     return i;
}

//Funkce testuje vznik pruseciku
//Funkce pocita prusecik mezi hranou a seznamem objektu typu LineString
public static boolean prusecik (Point p0, Point p1, ArrayList<LineString>
                                                                  testvse)
{
      boolean i = false;    // nastavena hodnota odpovida neexistenci
                            //pruseciku
```

```
        //Vektor testovane hrany
        Vektor u = new Vektor(p1.getX()-p0.getX(),p1.getY()-p0.getY());


        //Testovani se vsemi hranami dane liniemi
        for(int k = 0; k < testvse.size();k++)
        {
             LineString body = testvse.get(k);

             for(int j = 0;j < body.numPoints()-1;j++)
             {

                 Vektor v = new Vektor(body.getPoint(j+1).getX()-
                     body.getPoint(j).getX(),body.getPoint(j+1).getY()-
                                                 body.getPoint(j).getY());

                 Vektor w = new Vektor(p0.getX()-body.getPoint(j).getX(),
                                       p0.getY()-body.getPoint(j).getY());

             //Nema smysl resit prusecik sousednich hran

               if(p0.equals(body.getPoint(j))|p0.equals(body.getPoint(j+1))|
                  p1.equals(body.getPoint(j))|p1.equals(body.getPoint(j+1)))
               {
               }else{
                        double s;
                        double t;

                        s = (v.getU2()*w.getU1()-v.getU1()*w.getU2())/
                                  (v.getU1()*u.getU2()-v.getU2()*u.getU1());

                        t = (u.getU1()*w.getU2()-u.getU2()*w.getU1())/
                                  (u.getU1()*v.getU2()-u.getU2()*v.getU1());

                        //Pokud prusecik existuje
                        if((s > 0 && s < 1) &&(t > 0 && t < 1))
                        {
                             i = true;
                             break;
                        }
               }
             }

             if(i == true) break;
        }

        return i;
}

//Funkce vraci pocet pruseciku s polygonem
//Funkce na urceni poctu pruseciku s polygonem
public static int prusecik (Point p0, Point p1, Polygon testvse)
{
    int pocet = 0;

    //Vektor poloprimky z testovaneho bodu
    Vektor u = new Vektor(p1.getX()-p0.getX(),p1.getY()-p0.getY());
```

```
    //Testovani se vsemi hranami daneho polygonu
    for(int k = 0; k < testvse.numRings();k++)
    {
        LinearRing l = testvse.getRing(k);

        for(int j = 0;j < l.numPoints()-1;j++)
        {

           Vektor v = new Vektor(l.getPoint(j+1).getX()-
                           l.getPoint(j).getX(),l.getPoint(j+1).getY()-
                                                   l.getPoint(j).getY());

           Vektor w = new Vektor(p0.getX()-l.getPoint(j).getX(),
                                        p0.getY()-l.getPoint(j).getY());

                double s;
                double t;

                s = (v.getU2()*w.getU1()-v.getU1()*w.getU2())/
                               (v.getU1()*u.getU2()-v.getU2()*u.getU1());
                t = (u.getU1()*w.getU2()-u.getU2()*w.getU1())/
                               (u.getU1()*v.getU2()-u.getU2()*v.getU1());

                if(s <= 0 && (t >= 0 && t < 1))
                {
                    pocet++; //Pokud prusecik existuje zvysy se pocet o 1
                }

         }
     }

     return pocet;
}

//Tvorba konvexni obalky bodu linie
//Funkce vraci body obalove zony kolem dane linie jako typ LineString
public static LineString hull(List<Point> body)
{
    ArrayList<Point> obal = new ArrayList<Point>();
    LineString l = new LineString();

    //Serazeni bodu polygonu podle souradnice x
    if(body.size() > 0)
    {

        //Klicem je souradnive x a hodnotou dany bod
        SortedMap<Double,Point> bodyx = new TreeMap<Double, Point>();

        for(int i = 0; i < body.size();i++)
        {
            bodyx.put(body.get(i).getX(), body.get(i));
        }



        //Vypocet horni casti obalu z bodu  nad delici primkou
        ArrayList<Point> body2 = new ArrayList<Point>(bodyx.values());
        ArrayList<Point>horni = new ArrayList<Point>();
        List<Point>dolni = new ArrayList<Point>();

        int n = body2.size()-1; //koncovy bod delici usecky
```

```
obal.add(body2.get(0));
int i = 1;

//Delici usecka
Point p0 = body2.get(0);
Point p1 = body2.get(n);
double dy = p1.getY()-p0.getY(); //souradnicove rozdily
double dx = p1.getX()-p0.getX();
double ws = Math.atan2(dy,dx);   //vypocet smerniku

//Zbyle body mnoziny
while(i < n)
{
    double dpy;
    double dpx;
    double wp;
    Point p = body2.get(i);

    dpy = p.getY()-p0.getY(); //souradnicove rozdily
    dpx = p.getX()-p0.getX();
    wp = Math.atan2(dpy,dpx); //vypocet smerniku

    //Test zda bude bod pouzit pro vypocet horni nebo dolni
    //obalky
    if((wp-ws) > 0)
    {
        horni.add(body2.get(i));  //bod lezi vlevo
    }else{
        dolni.add(body2.get(i));  //bod lezi vpravo
    }

    i++;
}

//Tvorba horniho konvexniho obalu
int j = 0;
while(j < horni.size())
{
    //Aktualne posledni hrana obalu
    if(j == 0)
    {
        p0 = body2.get(0);
        p1 = body2.get(n);
     }else{
          p0 = obal.get(obal.size()-2);
          p1 = obal.get(obal.size()-1);
     }

     dy = p1.getY()-p0.getY();  //souradnicove rozdily
     dx = p1.getX()-p0.getX();
     ws = Math.atan2(dy,dx);       //vypocet smerniku

     SortedMap<Double,Integer> bl = new TreeMap<
                                                Double,Integer>();
     SortedMap<Double,Integer> bl2 = new TreeMap<
                                                Double,Integer>();

     for(int k = j;k < horni.size();k++)
     {
          //Testovany bod
```

```
            Point p = horni.get(k);
            double dpy = p.getY()-p0.getY();
            double dpx = p.getX()-p0.getX();
            double wp = Math.atan2(dpy,dpx);

            if((wp-ws) > 0)
            {
                bl.put((wp-ws),k); //bod lezi vlevo
            }else{
                  bl2.put((wp-ws),k); //bod lezi vpravo
            }
        }

        ArrayList<Integer> cl = new ArrayList<Integer>(bl.values());
        ArrayList<Integer> cl2 = new ArrayList<Integer>(
                                                        bl2.values());

        int v;
        if(cl.size() > 0)
        {
            v = cl.get(cl.size()-1);   // oznaci bod nejvice vlevo
        }else{
            v = cl2.get(cl2.size()-1); //oznaci bod nejmene vpravo
        }

        obal.add(horni.get(v)); //vlozeni oznaceneho bodu do obalu
        j = v+1;

    }

    obal.add(body2.get(n));
    dolni = GeneralizacePlochy.reverse(dolni);
    dolni.add(body2.get(0));

    //Tvorba dolniho konvexniho obalu
    int u = 0;
    while(u < dolni.size())
    {
         //Aktualne posledni hrana obalu
        if(u == 0)
        {
            p0 = body2.get(n);
            p1 = body2.get(0);
        }else{
            p0 = obal.get(obal.size()-2);
            p1 = obal.get(obal.size()-1);
        }

        dy = p1.getY()-p0.getY();
        dx = Math.abs(p1.getX()-p0.getX());
        ws = Math.atan2(dy,dx);

        SortedMap<Double,Integer> bl = new TreeMap<Double,Integer>();
        SortedMap<Double,Integer> bl2 = new TreeMap<
                                                    Double,Integer>();

        for(int k = u;k < dolni.size();k++)
        {
            //Testovany bod
            Point p = dolni.get(k);
            double dpy = p.getY()-p0.getY();
```

```
                    double dpx = Math.abs(p.getX()-p0.getX());
                    double wp = Math.atan2(dpy,dpx);

                    if((wp-ws) < 0)
                    {
                         bl.put((wp-ws),k);  //bod lezi vlevo
                    }else{
                         bl2.put((wp-ws),k);  //bod lezi vpravo
                    }
               }

               ArrayList<Integer> cl = new ArrayList<Integer>(bl.values());
               ArrayList<Integer> cl2 = new ArrayList<Integer>(
                                                              bl2.values());

               int v;
               if(cl.size() > 0)
               {
                   v = cl.get(0);  // oznaci bod nejvice vlevo
               }else{
                   v= cl2.get(0); // oznaci bod nejmene vpravo
               }

               obal.add(dolni.get(v)); //vlozeni oznaceneho bodu do obalu
               u = v+1;
            }

            //Prevedeni seznamu bod; na objekt typu LineString
            Point[] c = new Point[obal.size()];
            obal.toArray(c);
            l = new LineString(c);
         }

         return l;
      }

}
```

Příloha 5 – Zdrojový kód třídy `Vektor`

```
package cz.hrncirova.diplomka;

public class Vektor
{
	private double u1;
	private double u2;

	//Konstruktor
	public Vektor(double u1, double u2)
	{
		this.u1 = u1;
		this.u2 = u2;
	}


	//Metody
	public double getU1()
	{
		return u1;
	}

	public void setU1(double u1)
	{
		this.u1 = u1;
	}

	public double getU2()
	{
		return u2;
	}

	public void setU2(double u2)
	{
		this.u2 = u2;
	}
}
```

Příloha 6 – Obsah přiloženého CD

- **Diplomová práce**
  - *Diplomova_prace.pdf*
- **Zdrojové kódy**
  - *generalizace.jar*
  - *generalizace.zip*