Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Simplifikace vektorových geodat pro účely progresivní vizualizace

Plzeň, 2014 Vladislav Razým

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2014

Vladislav Razým

Poděkování

Rád bych poděkoval Ing. Janu Ježkovi, Ph.D. za odborné vedení diplomové práce a za poskytnutí cenných informací z oboru geografických informačních systémů. Dále bych chtěl poděkovat svým rodičům za podporu v průběhu studia.

Abstract

The aim of this thesis is to propose and implement an efficient algorithm of progressive transmission of vector data, particularly linear features. The issue of progressive transmission of linear features is closely related to their simplification. As a result of simplification, topology between the original and simplified linear features may not be preserved, which is undesirable in GIS. Therefore, it is necessary to maintain consistent topology during progressive transmission of linear features. The basis of the proposed method is the BLG tree data structure that stores the results of the Douglas-Peucker simplification algorithm. The BLG tree is further used to check topology and determine transmission order of the individual points of linear features. One part of the proposed method is the special lossless compression algorithm of linear features that serves to increase efficiency of progressive transmission. Experiments on real geographic data (river network in the Czech Republic, ∼225 000 points) show that computation time of all algorithm steps (BLG tree creation, topology checking, encoding) is of the order of tenths of a second. The results of experiments indicate that the proposed method could be used as part of real-time applications. The proposed method can be used not only to progressive transmission and visualization of linear features, but also to their topologically consistent simplification and their compression.

Abstrakt

Cílem práce je návrh a implementace efektivního algoritmu progresivního přenosu vektorových dat, konkrétně liniových prvků. S problematikou progresivního přenosu liniových prvků úzce souvisí problematika jejich simplifikace. V důsledku simplifikace může dojít k porušení topologie mezi původními a zjednodušenými liniovými prvky, což je v oblasti GIS posuzováno jako nežádoucí jev. Proto je nutné, aby byla v průběhu progresivního přenosu liniových prvků topologie zachována. Těžištěm navržené metody je datová struktura BLG strom, která ukládá výsledky simplifikačního algoritmu Douglas-Peucker. BLG strom je dále využit pro kontrolu topologie a určení pořadí přenosu jednotlivých bodů liniových prvků. Pro větší efektivitu progresivního přenosu je součástí navržené metody také speciální algoritmus bezztrátové komprese liniových prvků. V průběhu testování algoritmu na reálných geografických datech (vodní toky ČR, ∼225 000 bodů) bylo zjištěno, že se doba trvání všech kroků algoritmu (vytvoření BLG stromu, kontrola topologie, kódování) pohybovala řádově v desetinách sekundy. Výsledky testování naznačují, že by navržená metoda mohla být vhodná pro aplikace pracující v reálném čase. Navrženou metodu lze využít nejen k progresivnímu přenosu a vizualizaci liniových prvků, ale také k jejich topologicky konzistentní simplifikaci a kompresi.

Obsah

Úvod

Geografická data[1] neboli geodata se díky internetu stávají pro běžného uživatele jednoduše dostupná a mají velké praktické využití (webové mapy, mobilní mapy, virtuální glóby aj.). Geodata se skládají z geoobjektů[2], jejichž geometrie je v případě vektorových geodat popsána pomocí bodů a liniových prvků. Je zřejmé, že množství vektorových geodat může být velké, což přináší nové výzvy zejména v oblasti efektivity jejich přenosu.

Tato práce se zabývá simplifikací a progresivním přenosem liniových prvků, což jsou procesy, které jsou spolu velice úzce spjaty. Simplifikace neboli zjednodušení liniových prvků je jedna z metod jejich generalizace[3]. Progresivní přenos dat je typ přenosu, kdy je nejprve přenesena jejich hrubá aproximace, která je s dále přenesenými daty postupně zpřesňována. V případě, že je aktuální aproximace dat pro uživatele dostatečná, není třeba další data přenášet. Toho lze využít k radikálnímu snížení objemu přenášených, zejména v případě progresivního přenosu liniových prvků.

V průběhu progresivního přenosu liniových prvků je nutné jednotlivé body přenášet ve vhodném pořadí, aby každý přenesený bod co nejvíce zpřesnil aktuální aproximaci liniového prvku. K určení pořadí přenosu jednotlivých bodů lze použít vhodné simplifikační algoritmy, např. algoritmus Douglas-Peucker [6] nebo algoritmus Visvalingam-Whyatt [13]. V důsledku simplifikace liniových prvků však může dojít k porušení jejich vzájemných vztahů neboli topologie, což je v oblasti GIS nežádoucí jev. Za porušení topologie lze považovat například situaci, kdy dojde vlivem zjednodušení k průniku dvou původně se neprotínajících liniových prvků. V průběhu progresivního přenosu je proto nutné zaručit, aby byla zachována topologie mezi původními a dosud přenesenými (tzn. zjednodušenými) liniovými prvky.

Organizace textu v této práci je následující: V kapitole 1 je uveden současný stav dané problematiky a popis základních pojmů nezbytných pro popis navržené metody progresivního přenosu liniových prvků. V kapitole 2 jsou definovány podmínky zachování topologie v důsledku simplifikace liniových prvků. Popisu navržené metody progresivního přenosu liniových prvků se věnuje kapitola 3. Kapitola 4 popisuje implementační detaily navržené metody a kapitola 5 uvádí výsledky jejího použití v praxi.

[1] Data o poloze, tvaru a vztazích mezi jevy reálného světa, vyjádřená zpravidla ve formě souřadnic a topologie.

[2] Část modelované reality, kterou je možno na dané úrovni generalizace (na daném měřítku) modelovat jako jeden objekt.

[3] Proces výběru, zjednodušení a zevšeobecnění geografických dat.

Kapitola 1

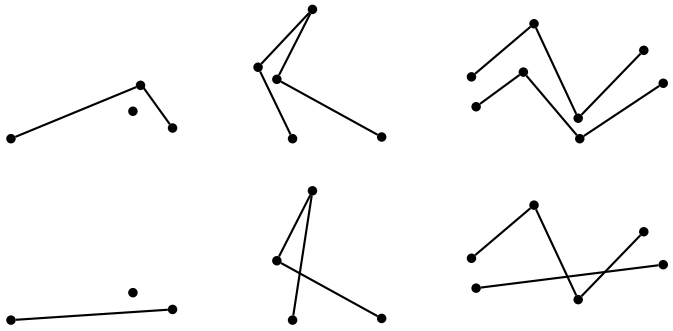
Současný stav

V této kapitole bude uveden současný stav problematiky progresivního přenosu geografických dat a základní pojmy nezbytné pro popis navržené metody. Jak bylo naznačeno již v úvodu, s problematikou progresivního přenosu geografických dat úzce souvisí proces jejich generalizace. Hlavním důvodem použití generalizace geografických dat je zlepšení grafické stránky jejich vizualizace. Přestože je generalizace do značné míry uměním zkušeného kartografa, existují algoritmy automatické generalizace, což je netriviální a časově náročný proces, který je z velké části založen na algoritmech výpočetní geometrie. Tato práce se však bude zabývat pouze simplifikací liniových prvků, což je jedna z metod generalizace geografických dat. Díky simplifikaci liniových prvků je totiž možné realizovat jejich progresivní přenos.

Zásadní úlohou v průběhu simplifikace liniových prvků je zachování vzájemných vztahů (topologie) mezi původními a zjednodušenými liniovými prvky. Vlivem simplifikace totiž může dojít ke změně relativní polohy bodu vůči liniovému prvku nebo může liniový prvek protínat jiný liniový prvek, ale také sám sebe (viz obr. 1.1). Porušení topologie je v oblasti GIS posuzováno jako nežádoucí jev, protože jeho důsledkem může dojít k nesprávnému výsledku prostorových dotazů, ale také ke zhoršení grafické stránky v průběhu vizualizace geografických dat. Problematika zachování topologie v průběhu simplifikace je v [15] považována za hlavní příčinu relativně pomalého rozvoje v oblasti progresivního přenosu vektorových dat. Tvorbou simplifikačních algoritmů liniových prvků zachovávajících topologii se zabývají např. v [4, 5, 8, 11].

S mapovými aplikacemi pracujícími v reálném čase je spjat termín on-the-fly generalizace, který může být definován jako vytvoření mapy v reálném čase v závislosti na požadovaném měřítku a účelu. Existují dva přístupy, jak může být on-the-fly generalizace realizována. První přístup vytvoří z původní mapy v průběhu předzpracování několik generalizovaných map s různou úrovní detailu (různých měřítek). V závislosti na požadavku uživatele je pak přenesena mapa z vybrané úrovně. Tento přístup může být v závislosti na způsobu uložení generalizovaných map realizován dvěma způsoby:

1. Generalizované mapy jsou uloženy nezávisle na sobě. V tomto případě dochází k redundanci dat jak při jejich uložení, tak při přenosu, protože každá generalizovaná mapa musí být vždy uložena, resp. přenesena celá.



Obrázek 1.1: Změna topologie v důsledku zjednodušení liniových prvků. (a) změna relativní polohy bodu vůči linii, (b) linie protíná sama sebe, (c) linie protíná jinou linii.

2. Generalizovaná mapa je uložena pouze pro nejnižší úroveň detailu, zatímco mapy s vyšší úrovní detailu ukládají pouze informace o zpřesnění úrovně předchozí. Tímto způsobem dochází k minimální redundanci dat jak při jejich uložení, tak při přenosu.

Ve druhém případě je přenos progresivní, protože mapa s nejnižší úrovní detailu (nejhrubší aproximace) je postupně zpřesňována daty uloženými ve vyšších úrovních detailu. Tato metoda je použita např. v [3, 14]. Nevýhodou přístupu s předzpracováním je jeho malá pružnost, která se projeví např. při změně původní mapy, kdy musí být znovu vytvořeny jednotlivé generalizované mapy. Další nevýhodou je existence omezeného počtu úrovní detailu, která na straně klienta neumožňuje „spojitý zoom“.

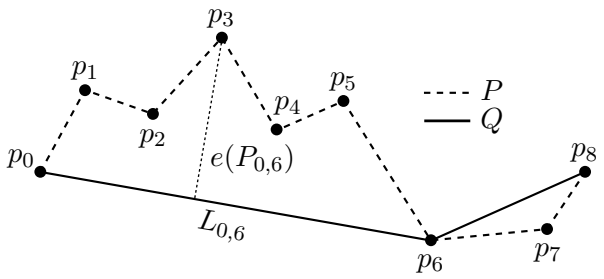
Druhý přístup on-the-fly generalizace je generalizovat původní mapu v reálném čase a bez předzpracování. Toto řešení je sice o mnoho pružnější, ovšem časově velice náročné. Doba generalizace původní mapy často nesplňuje požadavky aplikací pracujících v reálném čase, a proto praktické řešení on-the-fly generalizace bez předzpracování s největší pravděpodobností zatím neexistuje.

Poznámka V závislosti na uvedených informacích o on-the-fly generalizaci je možné přesněji definovat cíl této práce, který lze popsat jako on-the-fly simplifikaci liniových prvků bez předzpracování a jejich efektivní přenos. Jedná se tedy o úlohu, která je pouze podmnožinou úlohy on-the-fly generalizace geografických dat.

1.1 Základní pojmy

V této sekci bude definováno několik pojmů a zavedeno značení, které bude používáno po zbytek textu. Definice a značení základních pojmů z velké části vychází z [1, 10]. Pro snadnější pochopení definic poslouží obrázky 1.2 a 1.3.

Lomená čára P je posloupnost bodů $(p_k)_{k=0}^{n-1}$, přičemž každé dva po sobě jdoucí body jsou spojeny úsečkou, kterou budeme dále nazývat *hranou* lomené



Obrázek 1.2: Grafické znázornění definovaných pojmů.

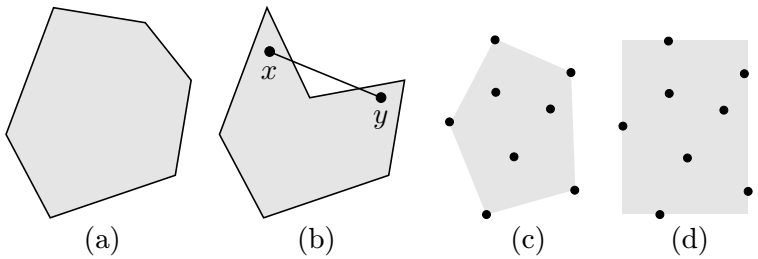
čáry. Lomená čára je *uzavřená*, jsou-li její koncové body spojeny hranou. Lomená čára je *jednoduchá*, pokud se neprotínají její nesousední hrany. *Segment* P_{ij} lomené čáry P je lomená čára $(p_k)_{k=i}^{j}$. Segment je tedy obecně část lomené čáry, dle definice je však segmentem i celá lomená čára. Jako *hlavní úsečku* L_{ij} segmentu P_{ij} budeme nazývat úsečku $p_i p_j$, tedy úsečku spojující koncové body segmentu P_{ij}. Označme $(k_i)_{i=0}^{m-1}$ rostoucí posloupnost nezáporných celých čísel, pro kterou platí $k_0 = 0$, $k_{m-1} = n - 1$. *Zjednodušená* lomená čára Q je lomená čára $(p_{k_i})_{i=0}^{m-1}$. Označme $d(\cdot, \cdot)$ eukleidovskou vzdálenost mezi dvěma body v prostoru libovolné dimenze. *Chyba segmentu* $e(P_{ij})$ je hodnota

$$e(P_{ij}) = \max_{i \leq k \leq j} d(p_k, L_{ij}),$$

kde $d(p_k, L_{ij})$ je eukleidovská (nejkratší) vzdálenost mezi bodem p_k a úsečkou L_{ij}. Chyba segmentu tedy představuje vzdálenost bodu segmentu nejvzdálenějšího od jeho hlavní úsečky. *Chyba zjednodušené lomené čáry* nebo *chyba simplifikace* $e(Q)$ je hodnota

$$e(Q) = \max_{0 \leq i < m-1} e(P_{k_i k_{i+1}}).$$

Řekneme, že zjednodušená lomená čára Q splňuje *toleranci zjednodušení* $\varepsilon \geq 0$, jestliže platí $e(Q) \leq \varepsilon$.



Obrázek 1.3: Grafické znázornění definovaných pojmů (a) konvexní polygon, (b) nekonvexní polygon, (c) konvexní obálka, (d) obdélníková obálka.

Polygon $S(P)$ je část roviny, která je ohraničená uzavřenou lomenou čarou P. Polygon S je *konvexní*, jestliže pro všechna $x \in S$ a $y \in S$ platí $xy \subseteq S$. *Konvexní obálka* $\mathcal{H}(A)$ množiny A bodů v rovině je nejmenší konvexní polygon

obsahující všechny body množiny A. *Obdélníková obálka* $\mathcal{B}(A)$ množiny bodů A v rovině je nejmenší konvexní polygon $S(P)$ obsahující všechny body množiny A, přičemž každá hrana P je rovnoběžná s jednou ze souřadnicových os.

1.2 Simplifikace liniových prvků

Pojem liniový prvek, který byl dosud používán a kterým jsou myšleny lomené čáry a hranice polygonů, je pro další popis příliš obecný, a proto se dále budeme omezovat pouze na lomené čáry. Při popisu však nedochází k újmě na obecnosti, neboť hranice polygonu je uzavřenou lomenou čarou.

Simplifikace neboli zjednodušení lomené čáry je proces, v průběhu kterého dochází k redukci počtu jejích bodů v závislosti na libovolném kritériu. Většina simplifikačních algoritmů je založena na detekci tzv. kritických bodů, což jsou body, které mají největší vliv na tvar lomené čáry. Způsob, jakým jsou tyto body hledány, má vliv na kvalitu (počet bodů, přesnost) zjednodušené lomené čáry, ale také na výpočetní složitost. Ta se u simplifikačních algoritmů pohybuje od $\mathcal{O}(n)$ po $\mathcal{O}(n^2)$, kde n je počet bodů lomené čáry. Velice podrobný popis problematiky simplifikace a různých simplifikačních algoritmů je popsán v knize [9]. Dle [2] je nejpoužívanějším algoritmem pro zjednodušení liniových prvků algoritmus Douglas-Peucker, který bude podrobně popsán v sekci 1.2.1.

Nepříjemností většiny simplifikačních algoritmů je, že pokud je vstupní lomená čára jednoduchá, nemusí platit, že zjednodušená lomená čára bude také jednoduchá. Jinými slovy je možné, že důsledkem zjednodušení dojde k „samoprůniku“ zjednodušené lomené čáry (viz obr. 1.1). Tato problematika bude podrobněji popsána v sekci 1.2.3 a kapitole 2.

1.2.1 Algoritmus Douglas-Peucker

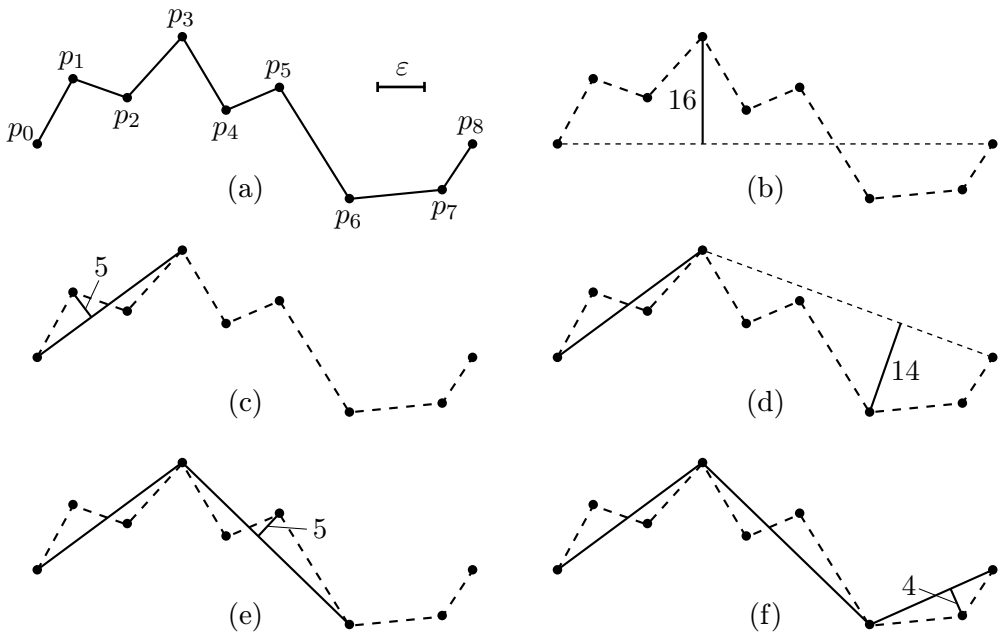
Algoritmus Douglas-Peucker [6] je simplifikační algoritmus, jehož vstupem je lomená čára P a tolerance zjednodušení ε a výstupem zjednodušená lomená čára Q splňující zadanou toleranci. Algoritmus je popsán rekurzivní funkcí `SimplifyDP` (viz algoritmus 1.1) a graficky znázorněn na obrázku 1.4. Vstupem funkce je segment P_{ij}, který má být zjednodušen. Funkce `SimplifyDP` nejprve nalezne bod p_k segmentu P_{ij}, který je nejvzdálenější od jeho hlavní úsečky L_{ij}. Je-li vzdálenost tohoto bodu větší než tolerance ε, pak je segment P_{ij} v tomto bodě rozdělen na dva segmenty P_{ik} a P_{kj}, které se po řadě stanou vstupem funkce `SimplifyDP`. V opačném případě budou koncové body segmentu P_{ij} součástí zjednodušené lomené čáry.

Výpočetní složitost algoritmu Douglas-Peucker je závislá na způsobu, jakým je hledán bod segmentu nejvzdálenější od jeho hlavní úsečky, a také na počtu bodů zjednodušené lomené čáry (algoritmus je tedy citlivý na velikost výstupu). Pokud je nejvzdálenější bod hledán sekvenčně (postupně se vypočítá vzdálenost každého bodu daného segmentu), je výpočetní složitost algoritmu $\mathcal{O}(n^2)$, přesněji $\mathcal{O}(mn)$, kde m, resp. n je počet bodů Q, resp. P. V [7] bylo dokázáno, že výpočetní složitost algoritmus Douglas-Peucker je $\mathcal{O}(n \log n)$ (hodnota složitosti pro danou velikost výstupu není v [7] uvedena). Důkaz je založen na principu, že bod segmentu nejvzdálenější od jeho hlavní úsečky je součástí jeho konvexní

Input : lomená čára P_{ij}, tolerance zjednodušení $\varepsilon \geq 0$
Output: zjednodušená lomená čára Q

`SimplifyDP(`P_{ij}`)`
  $p_k \leftarrow$ bod segmentu P_{ij} nejvzdálenější od jeho hlavní úsečky L_{ij}
  if $d(p_k, L_{ij}) > \varepsilon$ **then**
    `SimplifyDP(`P_{ik}`)`
    `SimplifyDP(`P_{kj}`)`
  else
    koncové body segmentu P_{ij} jsou součástí Q

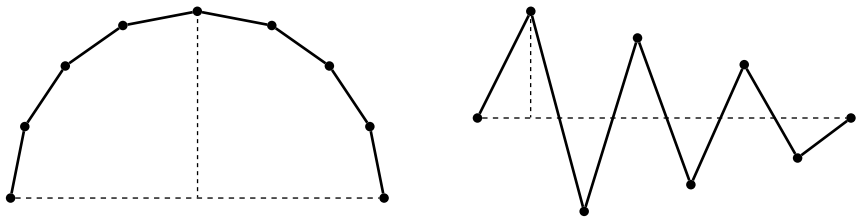
Algoritmus 1.1: Algoritmus Douglas-Peucker.



Obrázek 1.4: Grafické znázornění algoritmu Douglas-Peucker pro $\varepsilon = 7$.

obálky (nejvzdálenější bod segmentu tedy není hledán sekvenčně). Při testování na reálných datech však byla doba výpočtu pomocí $\mathcal{O}(n \log n)$ algoritmu kratší pouze pro velká n, protože algoritmus zpomaluje manipulace s datovými strukturami sloužícími pro výpočet a údržbu konvexní obálky.

V případě sekvenčního hledání nejvzdálenějšího bodu je doba výpočtu algoritmu Douglas-Peucker závislá na tom, kolikrát bude počítána vzdálenost bodu od hlavní úsečky. To závisí na tvaru lomené čáry, přesněji na poloze nejvzdálenějšího bodu, ve kterém bude segment rozdělen. Z hlediska doby výpočtu je optimální, pokud je vstupní segment rozdělen vždy v polovině, např. jsou-li body lomené čáry rovnoměrně rozmístěny na půlkružnici. Naopak nejhorší případ nastane, pokud je v každém kroku vstupní segment rozdělen na dva segmenty, přičemž jeden z nich tvoří pouze dva body, např. jsou-li body lomené čáry rozmístěny „cik-cak“ (viz obr. 1.5). Za předpokladu, že nejvzdálenější bod segmentu leží přibližně v jeho polovině, pak je průměrná (očekávaná) výpočetní složitost algoritmu Douglas-Peucker $\mathcal{O}(n \log n)$.



(a) (b)

Obrázek 1.5: Extrémní případy tvaru vstupní lomené čáry zjednodušované algoritmem Douglas-Peucker.

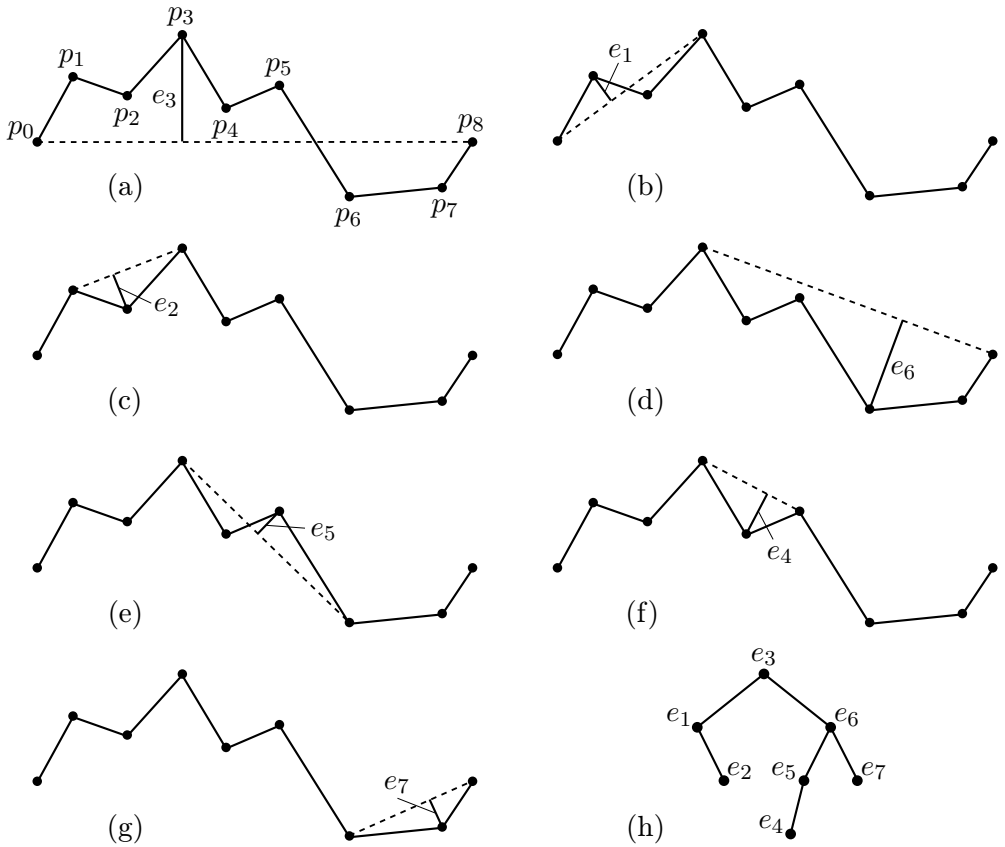
1.2.2 BLG strom

BLG (Binary Line Generalization) strom [12] je binární strom, který slouží k uložení výsledků algoritmu Douglas-Peucker. Vytvoření BLG stromu lze považovat za předzpracování s cílem následného rychlého zjednodušení lomené čáry. V BLG stromu jsou uloženy všechny body lomené čáry vyjma bodů koncových. V každém jeho uzlu je uložen bod lomené čáry společně s příslušnou chybou segmentu, který daný uzel reprezentuje. Algoritmus vytvoření BLG stromu je popsán rekurzivní funkcí `CreateBLGTree` (viz algoritmus 1.2) a graficky znázorněn na obrázku 1.6. Vstupem funkce je segment P_{ij} a výstupem kořen BLG stromu reprezentující tento segment. Funkce `CreateBLGTree` nejprve nalezne bod p_k segmentu P_{ij}, který je nejvzdálenější od jeho hlavní úsečky L_{ij} a společně s jeho vzdáleností ho uloží do kořene BLG stromu. Tato vzdálenost tedy představuje chybu simplifikace segmentu P_{ij}, pokud bychom za zjednodušenou lomenou čáru považovali jeho hlavní úsečku L_{ij}. Dále je segment P_{ij} v bodě p_k rozdělen na dva segmenty P_{ik} a P_{kj}, které se po řadě stanou vstupem funkce `CreateBLGTree`, jejíž výstupem je levý, resp. pravý potomek kořene BLG stromu. Tento proces se rekurzivně opakuje, dokud má vstupní segment alespoň tři body.

Input : lomená čára P_{ij}
Output: kořen BLG stromu

`CreateBLGTree(`P_{ij}`)`
- **if** $j - i < 2$ **then**
  - **return** null
- $p_k \leftarrow$ bod segmentu P_{ij} nejvzdálenější od jeho hlavní úsečky L_{ij}
- `node` $\leftarrow$ `CreateNode() // Segment(node)` $\leftarrow P_{ij}$
- `Point(node)` $\leftarrow p_k$
- `Error(node)` $\leftarrow d(p_k, L_{ij})$
- `Left(node)` $\leftarrow$ `CreateBLGTree(`P_{ik}`)`
- `Right(node)` $\leftarrow$ `CreateBLGTree(`P_{kj}`)`
- **return** node

Algoritmus 1.2: Algoritmus vytvoření BLG stromu.



Obrázek 1.6: Grafické znázornění vytvoření BLG stromu (h).

Výpočetní složitost vytvoření BLG stromu je závislá, stejně jako výpočetní složitost algoritmu Douglas-Peucker, na způsobu hledání bodu segmentu nejvzdálenějšího od jeho hlavní úsečky. Pokud je nejvzdálenější bod hledán sekvenčně (postupně se vypočítá vzdálenost každého bodu daného segmentu), je výpočetní složitost algoritmu $\mathcal{O}(n^2)$, kde n je počet bodů P. Pokud je hledán způsobem uvedeným v [7], pak je výpočetní složitost $\mathcal{O}(n \log n)$.

Hloubka BLG stromu je závislá na tvaru lomené čáry, přesněji na poloze nejvzdálenějšího bodu v segmentu. Je-li vstupní segment rozdělen vždy v polovině, např. jsou-li body lomené čáry rovnoměrně rozmístěny na půlkružnici, pak bude hloubka BLG stromu nejmenší možná, a BLG strom bude vyvážený. Naopak nejhorší případ nastane, pokud je v každém kroku vstupní segment rozdělen na dva segmenty, přičemž jeden z nich tvoří pouze dva body, např. jsou-li body lomené čáry rozmístěny „cik-cak“ (viz obr. 1.5). V tomto případě bude hloubka stromu největší možná a BLG strom bude degenerovaný na lineární seznam. Hloubka BLG stromu je proto obecně $\mathcal{O}(n)$, kde n je počet bodů vstupní lomené čáry. Budeme-li předpokládat, že nejvzdálenější bod segmentu leží přibližně v jeho polovině, pak je průměrná (očekávaná) hloubka BLG stromu $\mathcal{O}(\log n)$.

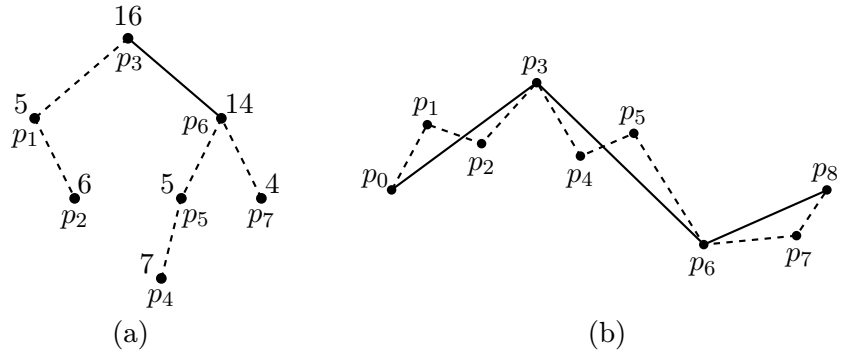
Simplifikace pomocí BLG stromu Proces zjednodušení lomené čáry P pomocí BLG stromu je popsán algoritmem 1.3 a zobrazen na obrázku 1.7. Vstupem algoritmu jsou koncové body lomené čáry P (nejsou uloženy v BLG stromu), kořen BLG stromu a tolerance zjednodušení. Dokud je chyba v uzlu větší než tolerance zjednodušení, prochází algoritmus uzly BLG stromu metodou in-order a bod v navštíveném uzlu přidá na konec zjednodušené lomené čáry. Výpočetní složitost algoritmu zjednodušení pomocí BLG stromu je $\mathcal{O}(m)$, kde m je počet bodů zjednodušené lomené čáry.

Input : p_0, p_{n-1}, kořen BLG stromu `root`, tolerance zjednodušení ε
Output: zjednodušená lomená čára $Q = (q_i)_{i=0}^{m-1}$

begin
  $q_0 \leftarrow p_0$
  `Simplify(root)`
  $q_{m-1} \leftarrow p_{n-1}$

`Simplify(node)`
  if `node` $\neq$ `null` **and** `Error(node)` $> \varepsilon$ **then**
    `Simplify(Left(node))`
    přidej `Point(node)` na konec lomené čáry Q
    `Simplify(Right(node))`

Algoritmus 1.3: Simplifikace pomocí BLG stromu.



Obrázek 1.7: Grafické znázornění simplifikace pomocí BLG stromu pro $\varepsilon = 7$. Plnou čarou je zobrazen (a) průchod BLG stromu a (b) odpovídající zjednodušená lomená čára.

1.2.3 Zachování topologie

V důsledku simplifikace lomených čar může dojít k porušení jejich topologie (viz obr. 1.1). Porušení topologie je v oblasti GIS nežádoucí, a proto vzniklo několik speciálních simplifikačních algoritmů, které se snaží tomuto jevu zamezit. Algoritmy zachovávající topologii se označují jako topologicky konzistentní simplifikační algoritmy a jejich princip spočívá v modifikaci některého z běž-

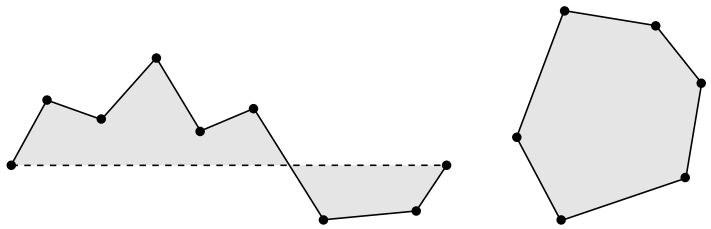
ných simplifikačních algoritmů. Tato modifikace je založená na dodatečných podmínkách simplifikace, jejichž splnění zaručuje správnost topologie zjednodušené lomené čáry. V praxi je tímto způsobem často modifikován algoritmus Douglas-Peucker [4, 8, 11].

Nevýhodou topologicky konzistentních simplifikačních algoritmů oproti obyčejným simplifikačním algoritmům je jejich vyšší časová náročnost. Doba kontroly topologie totiž mnohdy několikanásobně překračuje dobu výpočtu samotné simplifikace, a proto je způsob kontroly topologie hlavním aspektem, který určuje časovou náročnost topologicky konzistentních simplifikačních algoritmů.

Kapitola 2

Podmínky zachování topologie

V této krátké kapitole budou stanoveny podmínky, které musí být splněny, aby byla simplifikace lomených čar topologicky konzistentní. Aby však mohly být tyto podmínky stanoveny, je třeba přesněji definovat, co je za porušení topologie považováno. K tomuto účelu je nutno nejprve uvést, jaké mohou nastat případy vzájemné polohy bodu x a lomené čáry P, přičemž budeme předpokládat, že $x \notin P$. Protože lomená čára nedělí rovinu na dvě poloroviny (na rozdíl od přímky), nelze vzájemnou polohu bodu a lomené čáry popsat ve smyslu orientace (napravo, nalevo). Pro účely zachování topologie proto budeme vzájemnou polohu bodu a lomené čáry vyjadřovat jako vzájemnou polohu bodu a polygonu tvořeného touto lomenou čarou po spojení jejích koncových bodů hranou (viz obr. 2.1).



Obrázek 2.1: Body ležící uvnitř polygonu tvořeného (a) lomenou čarou, (b) uzavřenou lomenou čarou.

Uvažujme množinu bodů a jednoduchých lomených čar v rovině, z nichž žádné dvě se neprotínají. Aby důsledkem zjednodušení lomených čar z této množiny nedošlo k porušení topologie, musí být splněny tyto podmínky:

1. Musí být zachována vzájemná poloha mezi body a zjednodušenými lomenými čarami.

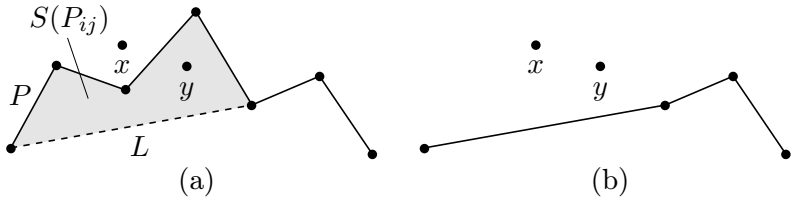
2. Musí být zachována vzájemná poloha mezi zjednodušenými lomenými čarami.

3. Všechny zjednodušené lomené čáry musí být jednoduché.

Za jakých okolností je možné zjednodušit lomenou čáru, aby byly splněny tyto podmínky, objasňují tvrzení 1, 2, 3, která vychází z [4, 11].

Tvrzení 1. Je dán bod x a jednoduchá lomená čára P. Nechť je libovolný segment P_{ij} zjednodušen jeho hlavní úsečkou. Označme $S(P_{ij})$ polygon tvořený segmentem P_{ij}. Jestliže platí $x \notin S(P_{ij})$, pak nedojde ke změně vzájemné polohy bodu x a zjednodušené lomené čáry (viz obr. 2.2).

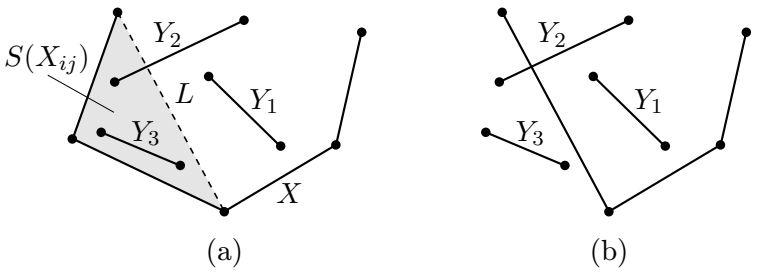
Důkaz. Označme L hlavní úsečku segmentu P_{ij}. Protože platí $L \subseteq S(P_{ij})$, pak pro bod $x \notin S(P_{ij})$ platí také $x \notin L$, a proto nedošlo ke změně vzájemné polohy bodu lomené čáry. □



Obrázek 2.2: (a) Původní lomená čára P, (b) zjednodušená lomená čára P. V důsledku zjednodušení lomené čáry P došlo ke změně vzájemné polohy mezi bodem y a zjednodušenou lomenou čarou P.

Tvrzení 2. Jsou dány vzájemně se neprotínající jednoduché lomené čáry X a Y. Nechť je libovolný segment $X_{ij} \subseteq X$ zjednodušen jeho hlavní úsečkou. Označme $S(X_{ij})$ polygon tvořený segmentem X_{ij}. Jestliže platí $Y \cap S(X_{ij}) = \emptyset$, pak se zjednodušená lomená čára X a lomená čára Y neprotínají a je zachována vzájemná poloha mezi nimi (viz obr. 2.3).

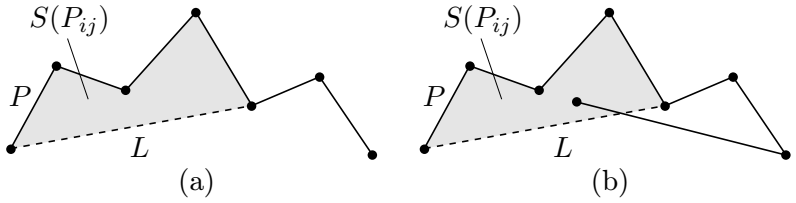
Důkaz. Označme L hlavní úsečku segmentu X_{ij}. Platí, že $L \subseteq S(X_{ij})$. Jestliže $Y \cap S(X_{ij}) = \emptyset$, pak také $Y \cap L = \emptyset$, a proto se zjednodušená lomená čára X a lomená čára Y nemohou protínat ani nemohou měnit svojí vzájemnou polohu. □



Obrázek 2.3: (a) Původní lomená čára X, (b) zjednodušená lomená čára X. V důsledku zjednodušení došlo ke změně vzájemné polohy mezi lomenými čarami Y_2, Y_3 a zjednodušenou lomenou čarou X.

Tvrzení 3. Je dána jednoduchá lomená čára P. Nechť je libovolný segment P_{ij} zjednodušen jeho hlavní úsečkou. Označme $P^{\mathsf{c}} = P \setminus P_{ij}$ a $S(X_{ij})$ polygon tvořený segmentem P_{ij}. Jestliže platí $P^{\mathsf{c}} \cap S(P_{ij}) = \emptyset$, pak je zjednodušená lomená čára jednoduchá (viz obr. 2.4).

Důkaz. Jestliže platí $P^{\mathsf{c}} \cap S(P_{ij}) = \emptyset$ a $L \subseteq S(P_{ij})$, pak platí $P^{\mathsf{c}} \cap L = \emptyset$. Protože P^{c} neprotíná L, bude zjednodušená lomená čára P jednoduchá. □



Obrázek 2.4: (a) Zjednodušená lomená čára P bude jednoduchá, (b) zjednodušená lomená čára P nebude jednoduchá.

Kapitola 3

Navržená metoda

Jak bylo zmíněno již v úvodu, cílem této práce je vytvořit efektivní algoritmus progresivního přenosu liniových prvků (lomených čar), v průběhu kterého je zachována jejich původní topologie. Podrobněji bude problematika vysvětlena na následujícím příkladu.

Uživatel (klient) chce zobrazit na svém počítači říční síť celé ČR. Geometrie říční sítě je popsána pomocí lomených čar a je s přesností na 1 m na zemském povrchu uložena na serveru v prostorové databázi. Uvažujme, že množství dat popisujících geometrii sítě je tak velké, že nelze přenést v reálném čase. Přenášet kompletní geometrii však není nutné, protože pro zobrazení říční sítě celé ČR na běžném monitoru není třeba mít k dispozici její geometrii s přesností na 1 m. Geometrii je proto možné zjednodušit odstraněním nepotřebných bodů z lomených čar, které ji popisují. V závislosti na měřítku, ve kterém má být říční síť u klienta zobrazena, je možné stanovit toleranci zjednodušení (viz sekce 1.2) tak, aby vliv odstraněných bodů měl na přesnost zobrazení minimální nebo žádný vliv. Na straně serveru se tedy geometrie s požadovanou tolerancí zjednoduší. Důsledkem zjednodušení však může dojít k porušení topologie říční sítě, a proto je třeba v jeho průběhu správnost topologie zajistit. Zjednodušená, topologicky konzistentní geometrie říční sítě je poté odeslána klientovi a zobrazena na jeho monitoru.

3.1 Topologicky konzistentní simplifikace

Základem topologicky konzistentního simplifikačního algoritmu je datová struktura BLG strom (viz sekce 1.2.2). Pro účely zachování topologie v průběhu simplifikace bude každý uzel BLG stromu ukládat informaci o tom, zda je možné zjednodušit segment daného uzlu jeho hlavní úsečkou, aniž by došlo k porušení topologie. Tato informace bude v uzlu `node` uložena jako pravdivostní hodnota (true, false) a bude označována jako `Required(node)`. Bod `Point(node)` budeme označovat jako *potřebný*, pokud `Required(node)` = true. Postup topologicky konzistentní simplifikace lomené čáry pomocí BLG stromu je popsán algoritmem 3.1. Od algoritmu „nekonzistentní“ simplifikace (viz algoritmus 1.3) se liší pouze v tom, že průchod stromu (zjednodušení) probíhá nejen, dokud není splněna požadovaná tolerance zjednodušení, ale také, dokud není možné zjednodušit příslušný segment jeho hlavní úsečkou.

Input : p_0, p_{n-1}, kořen BLG stromu root, tolerance zjednodušení ε
Output: zjednodušená lomená čára $Q = (q_i)_{i=0}^{m-1}$

```
begin
    q_0 ← p_0
    Simplify(root)
    q_{m-1} ← p_{n-1}

Simplify(node)
    if node ≠ null then
        if Error(node) > ε or Required(node) = true then
            Simplify(Left(node))
            přidej Point(node) na konec lomené čáry Q
            Simplify(Right(node))
```

Algoritmus 3.1: Topologicky konzistentní simplifikace pomocí BLG stromu.

Návod na určení, zda je bod potřebný nebo nepotřebný, dávají tvrzení 1, 2, 3 z kapitoly 2. Dále bude popsán nejjednodušší případ, kdy je potřebnost bodů zjištěna v závislosti na tvrzení 1, tedy tak, aby v důsledku zjednodušení nedošlo ke změně vzájemné polohy bodu a zjednodušené lomené čáry. Budeme předpokládat, že před začátkem zjišťování potřebnosti bodů bude pro všechny uzly BLG stromu platit `Required` = false. Postup zjištění potřebnosti bodů popisuje algoritmus 3.2.

Input : node - uzel BLG stromu
Global: x - bod, vůči kterému je testována topologie

```
TestTopology(node)
    if node ≠ null then
        if x ∈ S(Segment(node)) then
            Required(node) ← true
        TestTopology(Left(node))
        TestTopology(Right(node))
```

Algoritmus 3.2: Algoritmus zjišťující potřebnost bodů dle tvrzení 1.

Algoritmus je popsán rekurzivní funkcí `TestTopology`, jejíž vstupem je uzel BLG stromu `node` a bod x. Princip algoritmu je velice jednoduchý. Pokud bod x leží uvnitř polygonu tvořeného segmentem uzlu `node`, je bod v tomto uzlu označen za potřebný. Postup se poté rekurzivně opakuje pro levého a pravého potomka uzlu `node`. Přestože je popis algoritmu jednoduchý, jeho výpočetní složitost je vysoká. Algoritmus musí projít celý strom a v každém uzlu provést test vzájemné polohy bodu a segmentu daného uzlu. Vzhledem k tomu, že v praxi může být počet testovaných bodů x velký, nebyl by tento postup příliš efektivní, a je proto nutné najít rychlejší způsob.

Jedním ze způsobů urychlení algoritmu 3.2 je použít místo testu vzájemné polohy bodu a segmentu, test vzájemné polohy bodu a konvexní obálky seg-

mentu, jak popisuje algoritmus 3.3. Zásadní změnou oproti algoritmu 3.2, která vede k jeho urychlení, je ukončení průchodu stromu, pokud bod x není prvkem konvexní obálky segmentu aktuálního uzlu. Potomci tohoto uzlu nemusí být dále testováni, protože je zaručeno, že bod x není prvkem konvexní obálky segmentu žádného jeho potomka.

Důkaz. Označme

$$A = \texttt{Segment(node)}$$
$$B = \texttt{Segment(Left(node))}$$
$$C = \texttt{Segment(Right(node))}$$

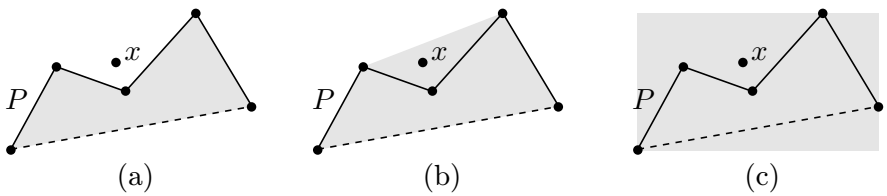
Protože $A = B \cup C$, platí $B \subseteq A$, a proto $\mathcal{H}(B) \subseteq \mathcal{H}(A)$. Jestliže $x \notin \mathcal{H}(A)$, pak musí platit $x \notin \mathcal{H}(B)$. Obdobně platí pro C. □

Input : node - uzel BLG stromu
Global: x - bod, vůči kterému je testována topologie

```
TestTopology(node)
    if node ≠ null then
        if x ∈ H(Segment(node)) then
            Required(node) ← true
            TestTopology(Left(node))
            TestTopology(Right(node))
```

Algoritmus 3.3: Algoritmus zjišťující potřebnost bodů dle tvrzení 1.

Algoritmus 3.3 má oproti algoritmu 3.2 jednu nevýhodu. Pomocí algoritmu 3.3 totiž může být za potřebný označen bod, který by potřebný být nemusel, a proto zjednodušená lomená čára může obsahovat více bodů, než by bylo třeba (viz obr. 3.1).



Obrázek 3.1: (a) Bod x není prvkem polygonu tvořeného segmentem P, ale je prvkem (b) konvexní obálky a (c) obdélníkové obálky segmentu P.

Dalším problémem algoritmu 3.3 je výpočet samotné konvexní obálky každého testovaného segmentu. Algoritmus však lze dále urychlit nahrazením testu vzájemné polohy bodu a konvexní obálky segmentu testem vzájemné polohy bodu a obdélníkové obálky segmentu. Při použití obdélníkové obálky se sice ještě více prohlubuje problém s nesprávným označováním bodů za potřebné (viz obr. 3.1), nicméně z hlediska paměťové a výpočetní náročnosti má použití obdélníkové obálky tyto výhody:

– Obdélníkovou obálku lze reprezentovat pomocí dvou bodů, a to bodem s minimálními a bodem s maximálními souřadnicemi obálky. Paměťové nároky na její uložení jsou tedy konstantní.

– V průběhu vytvoření BLG stromu lze vypočítat obdélníkovou obálku libovolného uzlu s konstantní složitostí.

– Vzájemnou polohu bodu a obdélníkové obálky i vzájemnou polohu dvou obdélníkových obálek lze zjistit s konstantní složitostí.

Protože jsou paměťové nároky na uložení obdélníkové obálky velmi malé, bude každý uzel BLG stromu ukládat obdélníkovou obálku jeho segmentu, která bude vypočítána v průběhu vytvoření BLG stromu. Dále budou popsány algoritmy zjištění potřebnosti bodů v závislosti na tvrzeních 1, 2, 3 ze sekce 2. Princip těchto algoritmů velice dobře popisují samotná tvrzení a vyplývá z nich, že jejich jádrem je test vzájemné polohy bodu a segmentu. Z výše zmíněných důvodů je však u nich test vzájemné polohy bodu a segmentu nahrazen testem vzájemné polohy bodu a obdélníkové obálky segmentu.

Ještě před tím, než budou popsány jednotlivé algoritmy testu topologie, je nutné zdůraznit důležitou věc. Uvažujme obdélníkovou obálku $\mathcal{B}(P)$, kde P je uzavřená lomená čára, která tvoří hranici $\mathcal{B}$. Za obdélníkovou obálku $\mathcal{B}$ budeme v popisu algoritmů testu topologie považovat pouze množinu $\mathcal{B} \setminus P$, tedy obdélníkovou obálku bez její hranice.

3.1.1 Test topologie „lomená čára - bod“

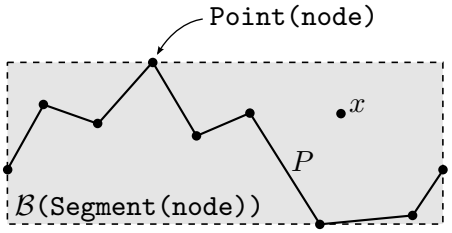
Algoritmus 3.4 popisuje způsob zjištění potřebnosti bodů v závislosti na tvrzení 1. Jeho princip je stejný jako princip algoritmu 3.3 s tím rozdílem, že je test vzájemné polohy bodu a konvexní obálky nahrazen testem vzájemné polohy bodu a obdélníkové obálky. Význam jednotlivých hodnot použitých v algoritmu 3.4 je znázorněn na obrázku 3.2.

Input : node - uzel BLG stromu lomené čáry P
Global: x - bod, vůči kterému je testována topologie

```
TestTopology(node)
    if node ≠ null then
        if x ∈ B(Segment(node)) then
            Required(node) ← true
            TestTopology(Left(node))
            TestTopology(Right(node))
```

Algoritmus 3.4: Algoritmus zjišťující potřebnost bodů dle tvrzení 1.

Výpočetní složitost testu topologie „lomená čára - bod“ je závislá na vzájemné poloze bodu a lomené čáry. Protože je výpočetní složitost testu vzájemné polohy bodu a obdélníkové obálky konstantní, je výpočetní složitost testu topologie „lomená čára - bod“ v nejlepším případě také $\mathcal{O}(1)$. Tato situace nastane, pokud testovaný bod leží mimo obdélníkovou obálku kořene BLG stromu. Obecně je však výpočetní složitost tohoto testu $\mathcal{O}(n)$, kde n je počet bodů lomené čáry P.



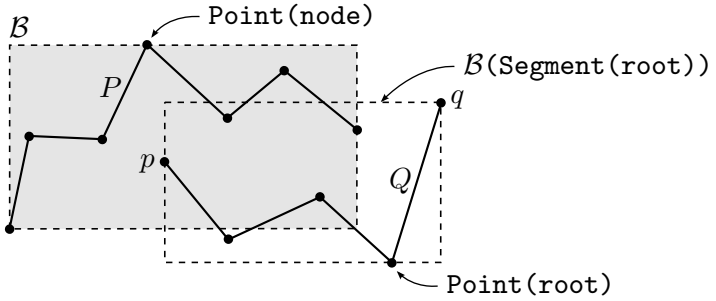
Obrázek 3.2: Obrázek k algoritmu 3.4.

3.1.2 Test topologie „lomená čára - lomená čára“

Algoritmus 3.5 zjišťuje potřebnost bodů v závislosti na tvrzení 2, tedy tak, aby v důsledku simplifikace nedošlo ke změně vzájemné polohy mezi zjednodušenou lomenou čarou a jinou lomenou čarou. Význam jednotlivých hodnot použitých v algoritmu 3.5 je znázorněn na obrázku 3.3. Algoritmus je popsán rekurzivní funkcí `TestTopology`. Uzel `node` je označen jako potřebný, jestliže v obdélníkové obálce $\mathcal{B}$ jeho segmentu leží libovolný bod lomené čáry Q. Nejprve se zjistí, zda v $\mathcal{B}$ neleží některý z koncových bodů Q (nejsou uloženy v BLG stromu) a poté, pomocí funkce `Contain`, zda v $\mathcal{B}$ neleží některý z ostatních („nekoncových“) bodů Q. Funkce `Contain` efektivně využívá hierarchické struktury obdélníkových obálek v BLG stromu. Hlavní urychlení spočívá v testu na prázdný průnik obdélníkových obálek:

$$\mathcal{B} \cap \mathcal{B}(\texttt{Segment(root)}) \stackrel{?}{=} \emptyset$$

Jestliže je tato podmínka splněna, pak je zaručeno, že žádný bod segmentu `Segment(root)` neleží v $\mathcal{B}$, a proto nemusí být tyto body dále testovány. V opačném případě se testuje, zda bod `Point(root)` leží uvnitř $\mathcal{B}$. Jestliže je tato podmínka splněna, je bod uzlu `node` označen jako potřebný. V opačném případě je funkce `Contain` rekurzivně zavolána pro levého a pravého potomka uzlu `root`.



Obrázek 3.3: Obrázek k algoritmu 3.5.

Výpočetní složitost testu topologie „lomená čára - lomená čára“ je závislá na vzájemné poloze lomených čar. Protože je výpočetní složitost testu vzájemné polohy dvou obdélníkových obálek konstantní, je výpočetní složitost testu topologie „lomená čára - lomená čára“ v nejlepším případě také $\mathcal{O}(1)$. Tato situace

Input : node - uzel BLG stromu lomené čáry P
Global: (p, q, root) - koncové body, resp. kořen BLG stromu lomené čáry Q, vůči které je testována topologie

```
TestTopology(node)
    if node ≠ null then
        B ← B(Segment(node))
        if p ∈ B or q ∈ B or Contain(B, root) then
            Required(node) ← true
            TestTopology(Left(node))
            TestTopology(Right(node))

Contain(B, root)
    if root = null or B ∩ B(Segment(root)) = ∅ then
        return false
    if Point(root) ∈ B then
        return true
    return Contain(B, Left(root)) or Contain(B, Right(root))
```

Algoritmus 3.5: Algoritmus zjišťující potřebnost bodů dle tvrzení 2.

nastane, pokud je průnik obdélníkové obálky kořene BLG stromu lomené čáry P a obdélníkové obálky kořene BLG stromu lomené čáry Q prázdný. V obecném případě nebude výpočetní složitost horší než $\mathcal{O}(mn)$, kde m, resp. n je počet bodů lomené čáry P, resp. Q.

3.1.3 Test topologie „lomená čára“

Algoritmus 3.6 zjišťuje potřebnost bodů v závislosti na tvrzení 3, tedy tak, aby nedocházelo k „samoprůnikům“ zjednodušené lomené čáry. Algoritmus je velice podobný algoritmu 3.5. Význam jednotlivých hodnot použitých v algoritmu 3.6 je znázorněn na obrázku 3.4. Algoritmus je popsán rekurzivní funkcí `TestTopology`. Uzel `node` je označen jako potřebný, jestliže v obdélníkové obálce $\mathcal{B}$ jeho segmentu leží libovolný bod P, který není prvkem tohoto segmentu. Nejprve se zjistí, zda v $\mathcal{B}$ neleží některý z koncových bodů P (nejsou uloženy v BLG stromu) a poté, pomocí funkce `Contain`, zda v $\mathcal{B}$ neleží některý z ostatních („nekoncových“) bodů P. Funkce `Contain` efektivně využívá hierarchické struktury obdélníkových obálek v BLG stromu. Podmínka `node` $\stackrel{?}{=}$ `root` slouží k tomu, aby nebyly testovány body P, které jsou prvkem testovaného segmentu `Segment(node)`. Hlavní urychlení spočívá v testu na prázdný průnik obdélníkových obálek:

$$\mathcal{B} \cap \mathcal{B}(\texttt{Segment(root)}) \stackrel{?}{=} \emptyset$$

Jestliže je tato podmínka splněna, pak je zaručeno, že žádný bod segmentu `Segment(root)` neleží v $\mathcal{B}$, a proto nemusí být tyto body dále testovány. V opačném případě se testuje, zda bod `Point(root)` není koncovým bodem testovaného segmentu a zda leží uvnitř $\mathcal{B}$. Jestliže je tato podmínka splněna, je

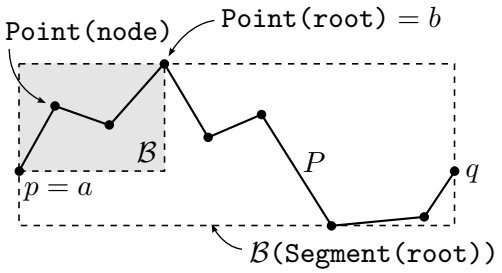
bod uzlu `node` označen jako potřebný. V opačném případě je funkce `Contain` rekurzivně zavolána pro levého a pravého potomka uzlu `root`.

Input : node - uzel BLG stromu lomené čáry P
Global: (p, q, root) - koncové body, resp. kořen BLG stromu lomené čáry P

```
TestTopology(node)
    if node ≠ null then
        (a, b) ← koncové body segmentu Segment(node)
        B ← B(Segment(node))
        if (a ≠ p and p ∈ B) or (b ≠ q and q ∈ B) or Contain(node, root)
        then
            Required(node) ← true
        TestTopology(Left(node))
        TestTopology(Right(node))

Contain(node, root)
    (a, b) ← koncové body segmentu Segment(node)
    B ← B(Segment(node))
    if root = null or node = root or B ∩ B(Segment(root)) = ∅ then
        return false
    if Point(root) ≠ a and Point(root) ≠ b and Point(root) ∈ B then
        return true
    return Contain(node, Left(root)) or Contain(node, Right(root))
```

Algoritmus 3.6: Algoritmus zjišťující potřebnost bodů dle tvrzení 3.



Obrázek 3.4: Obrázek k algoritmu 3.6.

Na rozdíl od předchozích dvou testů nemá tento test konstantní výpočetní složitost v nejlepším případě. Funkce `TestTopology` totiž musí vždy rekurzivně projít celý BLG strom. V nejlepším případě má tento test výpočetní složitost $\mathcal{O}(n \log n)$, kde n je počet bodů lomené čáry. Tento případ nastane, pokud je BLG strom vyvážený a průnik obdélníkových obálek potomků každého uzlu je prázdný. Obecně je však výpočetní složitost tohoto testu $\mathcal{O}(n^2)$.

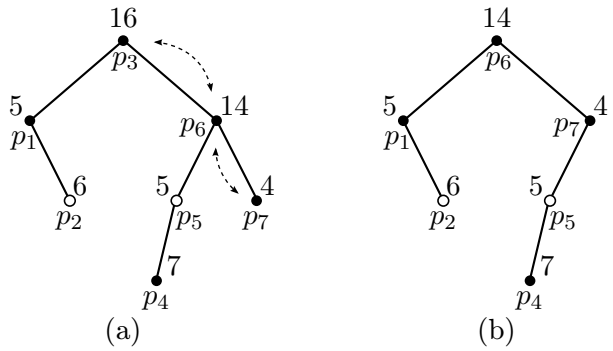
3.2 Progresivní přenos vektorových dat

V této sekci bude popsána problematika progresivního přenosu vektorových dat (bodů, lomených čar a polygonů). Na úvod bude popsán způsob progresivního přenosu jednotlivých bodů lomené čáry, který je základem progresivního přenosu vektorových dat. Poté bude popsán postup jejich kódování, dekódování a ukládání na straně klienta.

3.2.1 Progresivní přenos lomené čáry

Základem progresivního přenosu lomené čáry je opět datová struktura BLG strom. Progresivní přenos lomené čáry je založen na postupném přenosu jejích bodů tak, aby každý přenesený bod co možná nejvíce zpřesnil zjednodušenou lomenou čáru tvořenou již přenesenými body. Princip progresivního přenosu je relativně jednoduchý. Nejprve jsou přeneseny koncové body lomené čáry, které lze považovat za její nejhrubší aproximaci a následný přenos probíhá v krocích: (1) přenes bod uložený v kořeni BLG stromu, (2) odstraň kořen BLG stromu. Přenesené body jsou na straně klienta postupně ukládány také do BLG stromu.

Proces odstranění kořene BLG stromu je ukázán na obrázku 3.5. Odstranění probíhá tak, že se kořen stromu vymění s jeho „důležitějším“ potomkem a tento proces obnovy se pro původní kořen stromu rekurzivně opakuje, dokud se nestane listem, kdy je ze stromu odstraněn. Otázkou zůstává, jakým způsobem zvolit důležitějšího potomka daného uzlu. Má-li uzel pouze jednoho potomka, pak je tento důležitější. Má-li uzel potomky dva, pak se výběr důležitějšího z nich řídí tímto pravidlem: Jestliže je jeden z potomků potřebný a druhý nepotřebný, je zvolen potomek potřebný, jinak je zvolen potomek s větší chybou.



Obrázek 3.5: (a) Původní BLG strom, kde šipky naznačují proces výměny uzlů po odstranění kořene. (b) BLG strom po odstranění kořene z původního stromu. Černé uzly znázorňují potřebné uzly, bílé nepotřebné. Čísla u uzlů představují jejich chybu.

Výpočetní složitost odstranění kořene BLG stromu hloubky h je $\mathcal{O}(h)$ (kvůli „prohazování“ uzlů). Výpočetní složitost progresivního přenosu m bodů je proto $\mathcal{O}(mh)$. Uvažujme lomenou čáru P, která je tvořena n body. Výpočetní složitost progresivního přenosu kompletní lomené čáry je $\mathcal{O}(n^2)$, protože hloubka BLG stromu je obecně $\mathcal{O}(n)$.

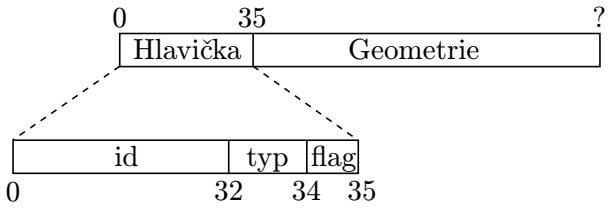
Výše popsaným progresivním přenosem lomené čáry je možné zabránit redundanci v přenášených datech, což bude vysvětleno na příkladu z úvodu této kapitoly: Uvažujme situaci, kdy si klient vyžádal kompletní říční síť ČR s přesností na 10 m. Protože je geometrie této sítě na serveru uložena s přesností na 1 m, dojde nejprve k jejímu zjednodušení, v jehož průběhu se pro každou lomenou čáru popisující geometrii vytvoří BLG strom. Poté budou ke klientovi výše popsaným způsobem progresivně přeneseny pouze takové body, aby byla splněna požadovaná přesnost a aby byla zachována topologie říční sítě. Na straně klienta budou přijaté body jednotlivých lomených čar ukládány do příslušných BLG stromů a po skončení přenosu vykresleny. Redundance při přenosu bude snížena ve chvíli, kdy klient požaduje geometrii říční sítě s jinou přesností, než má k dispozici. Bude-li klient požadovat geometrii sítě s menší přesností než 10 m, má k tomu všechna potřebná data k dispozici (v BLG stromech), a žádný přenos tedy probíhat nemusí. Jestliže klient žádá geometrii sítě s větší přesností než 10 m, pak se ze serveru přenesou jen data potřebná k požadovanému zpřesnění geometrie. Ta jsou uložená v již dříve vytvořených BLG stromech, a proto lze jednoduše pokračovat v progresivním přenosu jednotlivých bodů, dokud nebude splněna požadovaná přesnost.

3.2.2 Kódování

V oblasti kódování vektorových dat do binárního datového proudu je v praxi často používán standardizovaný formát WKB (well-known binary), pomocí kterého jsou vektorová data běžně přenášena a ukládána v prostorových databázích. Formát WKB však kóduje geometrii vektorových dat sekvenčně (bod po bodu), a proto není vhodný pro realizaci jejich progresivního přenosu. V této sekci proto bude popsán způsob progresivního kódování vektorových dat (bodů, lomených čar a polygonů), který je však oproti formátu WKB značně minimalistický.

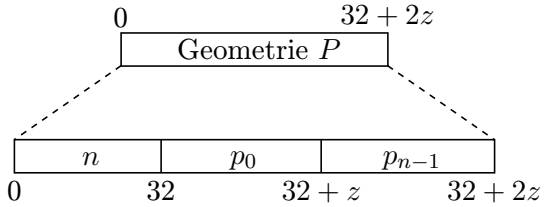
Pro názornou představu o problematice progresivního kódování je uveden následující příklad: Uvažujme dva objekty (lomenou čáru a polygon), jejichž geometrie bude přenášena ze serveru ke klientovi. Předpokládejme, že klient neví, kolik objektů bude přenášeno ani jakého jsou typu (zda se jedná o bod, lomenou čáru nebo polygon). Společně s geometrií je proto třeba zaslat klientovi také informaci o typu objektu. Uvažujme, že je nejprve přenesena zjednodušená geometrie obou objektů, která je na straně klienta dekódována a uložena do příslušných BLG stromů. V případě, že by klient požádal o zpřesnění geometrie, je třeba určit, do kterého BLG stromu mají být přenesené body uloženy. Proto je nutné každému objektu přiřadit hodnotu, která bude daný objekt jednoznačně identifikovat (id), a společně s geometrií a informací o typu objektu ji zaslat klientovi.

Geometrie jednotlivých objektů je kódována ve formátu, který je znázorněn na obrázku 3.6. V hlavičce je kódována informace o id a typu objektu. Význam atributu „flag“ bude vysvětlen dále. Způsob kódování geometrie je závislý na typu objektu. Pokud je objektem bod, pak položka geometrie obsahuje pouze souřadnice tohoto bodu. Jestliže je objektem lomená čára nebo polygon, je situace složitější. Jestliže je hodnota atributu flag 0, je dále zakódován počet

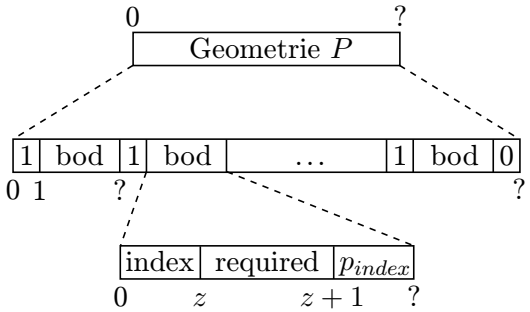


Obrázek 3.6: Formát kódování bodů, lomených čar a polygonů.

bodů daného objektu a souřadnice jeho koncových bodů (viz obr. 3.7). Jestliže je hodnota atributu flag 1, pak jsou zakódovány pouze body, které zpřesňují geometrii daného objektu (viz obr. 3.8). Tyto body jsou kódovány v pořadí, v jakém byly vybírány z BLG stromu (viz výše). Aby bylo možné jednotlivé body dekódovat (klient neví, kolik jich bylo zakódováno), je před každým bodem zakódován příznakový bit, který má hodnotu 0, pokud další bod nenásleduje. V rámci každého bodu je zakódována informace o jeho indexu, potřebnosti a souřadnicích.



Obrázek 3.7: Formát kódování geometrie lomené čáry $P = (p_k)_{k=0}^{n-1}$, pro flag $= 0$. Hodnota z představuje počet bitů potřebných pro kódování bodu lomené čáry, závisí tedy na použitém datovém typu jeho souřadnic.



Obrázek 3.8: Formát kódování geometrie lomené čáry $P = (p_k)_{k=0}^{n-1}$, pro flag $= 1$. Hodnota $z = \lceil \log_2 n \rceil$ určuje počet bitů na kódování indexu.

Největší množství zakódovaných dat je tvořeno souřadnicemi bodů zpřesňujících geometrii objektu. Aby byl přenos zakódovaných dat co nejefektivnější, jsou souřadnice těchto bodů komprimovány.

Komprese souřadnic

V této části bude popsán bezztrátový kompresní algoritmus, jehož cílem je snížit množství dat potřebných ke kódování souřadnic bodů zpřesňujících geometrii objektu. Algoritmus je bezztrátový, aby nedocházelo ke změně souřadnic dekódovaných bodů. V případě, že by byl algoritmus ztrátový, a tím pádem docházelo ke změně souřadnic dekódovaných bodů, docházelo by také ke zvětšování chyby zjednodušení, což by mohlo vést k překročení tolerance zjednodušení. Další požadavek na tento algoritmus je, aby byl rychlý. Komprimovat se v tomto případě vyplatí pouze za předpokladu, že je čas komprese a přenosu komprimovaných dat kratší než čas přenosu nekomprimovaných dat.

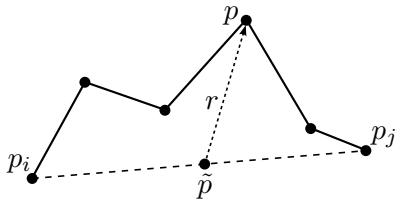
Cílem každého bezztrátového kompresního algoritmu je nalezení a odstranění redundance ve vstupních datech. Popsaný algoritmus snižuje datovou redundanci tím způsobem, že z již zakódovaných bodů odhaduje hodnotu souřadnic aktuálně kódovaného bodu a následně kóduje pouze chybu tohoto odhadu. Chybu odhadu budeme dále nazývat *residuum*. Předpokládá se, že počet bitů potřebný pro kódování residua bude nižší než počet bitů potřebný pro kódování souřadnic. Cílem je odhadnout hodnotu souřadnic kódovaného bodu co nejpřesněji, aby absolutní hodnota residua byla co nejmenší. Čím menší totiž bude absolutní hodnota residua, tím méně bude potřeba bitů pro jeho zakódování, a tím větší komprese bude dosaženo.

Výpočet residuí Jádrem kompresního algoritmu je opět BLG strom. V každém jeho uzlu je uložena hodnota residua, kterou lze vypočítat v průběhu vytvoření BLG stromu. Uvažujme, že p_i a p_j jsou koncové body segmentu daného uzlu. Odhad souřadnic bodu p daného uzlu bude značen $\tilde{p}$ a platí pro něj

$$\tilde{p} = \frac{p_i + p_j}{2}. \tag{3.1}$$

Odhad bodu p je tedy střed hlavní úsečky segmentu (viz obr. 3.9). Residuum budeme dále značit r a platí pro něj

$$r = p - \tilde{p}. \tag{3.2}$$



Obrázek 3.9: Grafické znázornění odhadu souřadnic bodu p a velikosti residua.

Je třeba si uvědomit, že residuum je vektor stejné dimenze jako body lomené čáry. Pro zdůraznění konkrétní souřadnice residua budeme dále používat dolní index, např. $r = (r_1, r_2)$.

Kódování residuí Dále budeme předpokládat, že jsou souřadnice všech bodů i residuí celočíselné. Jak bylo zmíněno výše, kompresní algoritmus je založen na myšlence, že počet bitů potřebný pro kódování residua bude nižší než počet bitů potřebný pro kódování souřadnic. V průběhu kódování residuí je třeba brát v potaz, že při dekódování musí být znám počet bitů zakódovaného residua. Je například možné stanovit počet bitů konstantní, ale to není z hlediska dosažené úrovně komprese efektivní volba. Způsob kódování residuí proto vychází z předpokladu, že se v průběhu progresivního přenosu lomené čáry velikost residuí postupně zmenšuje, a proto se postupně zmenšuje i počet bitů potřebný k jejich zakódování. Nejedná se však o pravidlo, ale pouze o trend.

Proces kódování residua popisuje algoritmus 3.7. Vstupem algoritmu je residuum r, naposledy kódované residuum s a proud bitů, do kterého bude residuum r zakódováno. Kódování residua probíhá pro každou jeho souřadnici zvlášť. Do proudu bitů je nejprve zapsán bit reprezentující znaménko r_i a poté následuje kódování absolutní hodnoty r_i. Nejprve se zjistí hodnota n_r, resp. n_s, která představuje počet bitů $|r_i|$, resp. $|s_i|$. Pokud platí $n_r \leq n_s$, pak se do proudu bitů zapíše příznakový bit 0 signalizující, že je hodnota $|r_i|$ zakódována pomocí n_s bitů. Tento případ je ve shodě s trendem, že se velikost residua postupně snižuje. V opačném případě, tedy pokud platí $n_r > n_s$, se do proudu bitů nejprve zapíše sekvence $n_r - n_s$ bitů s hodnotou 1 (toto je potřebné pro správné dekódování) a až poté příznakový bit 0 signalizující konec této sekvence. Do proudu bitů je poté zapsána hodnota $|r_i|$. V případě prvního kódovaného residua, kdy naposledy zakódované residuum neexistuje, je hodnota n_s stanovena jako počet bitů datového typu souřadnic bodů. Podmínkou je, že stejná hodnota musí být použita při dekódování prvního residua.

Input: $r = (r_1, r_2)$ - residuum, $s = (s_1, s_2)$ - naposledy zakódované residuum, stream - proud bitů

```
begin
    for i ← 1 to 2 do // pro každou souřadnici
        if r_i < 0 then // zapiš znaménkový bit
            WriteBit(1, stream)
        else
            WriteBit(0, stream)
        n_r ← ⌈log2(|r_i| + 1)⌉ // počet bitů |r_i|
        n_s ← ⌈log2(|s_i| + 1)⌉ // počet bitů |s_i|
        while n_r > n_s do
            WriteBit(1, stream)
            n_s ← n_s + 1
        WriteBit(0, stream)
        WriteBits(|r_i|, n_s, stream) // zapiš n_s bitů |r_i|
```

Algoritmus 3.7: Kódování residua.

Kódování neceločíselných souřadnic Doposud jsme předpokládali, že jsou souřadnice bodů celočíselné. V praxi se však většinou setkáme s případy, kdy jsou souřadnice reálná čísla (např. zeměpisné souřadnice). Reálná čísla jsou v počítači reprezentována čísly s pohyblivou řádovou čárkou, většinou pomocí 32bitového typu float, resp. 64bitového typu double definovaného standardem IEEE-754.

Souřadnice reprezentované datovým type float nebo double musí být nejprve převedeny na celá čísla, a poté mohou být kódovány stejným způsobem, jako bylo popsáno výše. Převod na celočíselné souřadnice však není transformací, ale pouze jinou interpretací datového typu float, resp. double.

Příklad kódování souřadnic Postup kódování neceločíselných souřadnic bude ukázán na příkladu. Uvažujme, že chceme zakódovat zeměpisné souřadnice bodu

$$p = (49.7238794, 13.3512331)$$

reprezentované 64bitovým datovým typem double. Dále uvažujme, že bod p zpřesňuje geometrii objektu, a proto budeme jeho souřadnice komprimovat (kódovat pouze residuum). Odhad hodnoty $\tilde{p}$ bodu p určíme dle vztahu 3.1 z koncových bodů p_i, p_j příslušného segmentu:

$$\begin{aligned} p_i &= (49.7247061, 13.3510072) \\ p_j &= (49.7239636, 13.3523958) \\ \tilde{p} = \frac{p_i + p_j}{2} &= (49.7243348, 13.3517015). \end{aligned}$$

Aby bylo kódované residuum celočíselné, je třeba souřadnice bodu p a jeho odhadu $\tilde{p}$ interpretovat jako celočíselné souřadnice, proto

$$\begin{aligned} p &= (4632194831207538626, 4623705694106520479) \\ \tilde{p} &= (4632194895306427697, 4623705957792278662). \end{aligned}$$

Residuum r vypočítáme dle vztahu 3.2 jako

$$r = p - \tilde{p} = (-64098889071, -263685758183)$$

Je vidět, že velikost souřadnic residua je menší než velikost celočíselných souřadnic bodu p. Proto je možné pro zakódování residua použít méně bitů něž pro zakódování celočíselných souřadnic bodu p, a tím dosáhnout komprese.

3.2.3 Dekódování

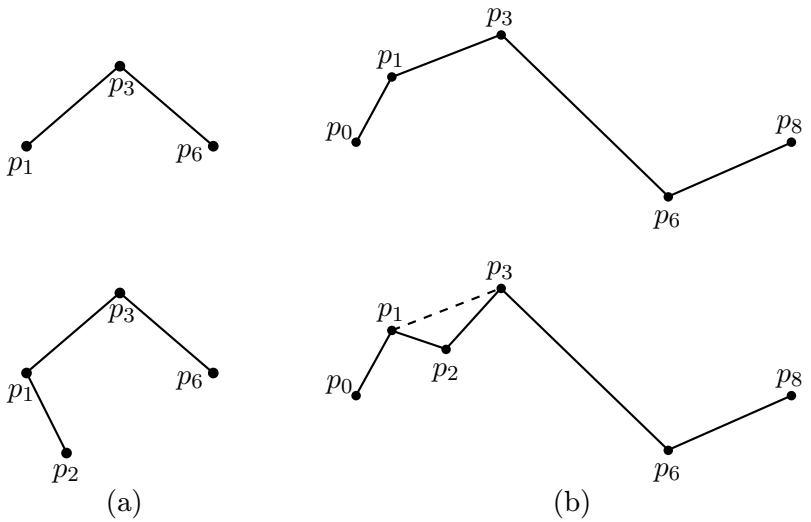
V této části bude popsán proces dekódování bodů zpřesňujících geometrii objektů a proces jejich ukládání do příslušných BLG stromů. Způsob, jakým jsou body zpřesňující geometrii kódovány, je popsán v sekci 3.2.2 a znázorněn na obrázku 3.8. Proces dekódování a uložení bodu do BLG stromu je popsán algoritmem 3.8 a zobrazen na obrázku 3.10.

```
Input: root - kořen BLG stromu, p_i, p_j - koncové body lomené čáry, stream
       - proud bitů obsahující zakódovaná data
begin
    node ← CreateNode()
    Index(node) ← dekóduj index ze stream
    if root = null then // BLG strom je prázdný
        root ← node
    while root ≠ node do
        if Index(node) > Index(root) then
            p_i ← Point(node)
            if Right(root) = null then
                Right(root) ← node
            root ← Right(root)
        else
            p_j ← Point(node)
            if Left(root) = null then
                Left(root) ← node
            root ← Left(root)
    Required(node) ← ReadBit(stream)
    r ← dekóduj residuum ze stream // viz algoritmus 3.9
    p̃ ← (p_i + p_j)/2 // odhad souřadnic
    Point(node) ← r + p̃ // rekonstrukce souřadnic
    Error(node) ← d(Point(node), p_i p_j) // výpočet chyby
```

Algoritmus 3.8: Algoritmus dekódování bodu a jeho uložení do BLG stromu.

Vstupem algoritmu je kořen BLG stromu, do kterého bude dekódovaný bod uložen, koncové body lomené čáry (ty jsou již dekódovány) a proud bitů obsahující zakódovaná data. Nejprve se dekóduje index bodu, který poslouží k nalezení místa v BLG stromu, kam má být dekódovaný bod uložen. Jestliže je strom prázdný, dekódovaný bod bude uložen do jeho kořene. V opačném případě je nalezeno místo v BLG stromu tak, aby indexy v jeho uzlech byly při průchodu stromu metodou in-order vzestupně seřazeny. V průběhu hledání se aktualizují koncové body p_i, p_j segmentu aktuálního uzlu, které poslouží k dekódování souřadnic a výpočtu chyby segmentu (proto chyba segmentu nemusí být na straně serveru kódována a přenášena). Protože jsou souřadnice bodů komprimovány, je nutné nejprve dekódovat residuum.

Postup dekódování residua je popsán algoritmem 3.9. Vstupem algoritmu je naposledy dekódované residuum $s = (s_1, s_2)$ a proud bitů, ze kterého bude residuum $r = (r_1, r_2)$ dekódováno. Nejprve se dekóduje znaménko residua r a poté se zjistí počet bitů naposledy dekódovaného residua n_s. Počet bitů pro dekódování hodnoty $|r_i|$ se určí jako součet hodnoty n_s a délky sekvence jedničkových bitů v proudu. Poté se dekóduje hodnota $|r_i|$ a v závislosti na hodnotě znaménka se určí hodnota r_i. V případě prvního dekódovaného residua, kdy naposledy dekódované residuum neexistuje, je hodnota n_s stanovena jako počet bitů datového typu souřadnic bodů.



Obrázek 3.10: Grafické znázornění uložení bodu p_2 do BLG stromu. (a) BLG strom, (b) lomená čára reprezentovaná BLG stromem.

Výpočetní složitost vložení bodu do BLG stromu je $\mathcal{O}(h)$, kde h je hloubka BLG stromu. Výpočetní složitost vložení m bodů do prázdného BLG stromu je $\mathcal{O}(mh)$, kde h je hloubka výsledného BLG stromu. Uvažujme lomenou čáru P, která je tvořena n body. Výpočetní složitost dekódování všech bodů lomené čáry P je $\mathcal{O}(n^2)$, protože hloubka výsledného BLG stromu je obecně $\mathcal{O}(n)$.

Příklad rekonstrukce souřadnic Jak probíhá rekonstrukce souřadnic bodů v praxi bude opět ukázáno na příkladu. Označme p souřadnice dekódovaného bodu, které neznáme. Nejprve dekódujeme hodnotu residua

$$r = (-64098889071, -263685758183).$$

Ze vztahu 3.2 vyplývá, že $p = r + \tilde{p}$, kde $\tilde{p}$ je odhad bodu p, který dle vztahu 3.1 určíme z koncových bodů p_i, p_j příslušného segmentu:

$$\begin{aligned} p_i &= (49.7247061, 13.3510072) \\ p_j &= (49.7239636, 13.3523958) \\ \tilde{p} = \frac{p_i + p_j}{2} &= (49.7243348, 13.3517015). \end{aligned}$$

Víme, že souřadnice bodu $\tilde{p}$ jsou reprezentovány 64bitovým datovým typem double, a proto je třeba je interpretovat jako celočíselné souřadnice

$$\tilde{p} = (4632194895306427697, 4623705957792278662).$$

Celočíselné souřadnice bodu p tedy mají hodnotu

$$p = r + \tilde{p} = (4632194831207538626, 4623705694106520479).$$

Protože jsou souřadnice bodů lomené čáry reprezentovány 64bitovým datovým typem double, je nutné takto interpretovat i celočíselné souřadnice bodu p, tedy

$$p = (49.7238794, 13.3512331).$$

Input : $s = (s_1, s_2)$ - naposledy dekódované residuum, stream - proud bitů
Output: $r = (r_1, r_2)$ - dekódované residuum

begin
  for $i \leftarrow 1$ **to** 2 **do** // pro každou souřadnici
    $sign \leftarrow$ ReadBit(stream) // přečti znaménkový bit
    $n_s \leftarrow \lceil \log_2(|s_i| + 1) \rceil$ // počet bitů $|s_i|$
    while ReadBit(stream) $= 1$ **do**
      $n_s \leftarrow n_s + 1$
    $r_i \leftarrow$ ReadBits(n_s, stream) // přečti n_s bitů
    if $sign = 1$ **then**
      $r_i \leftarrow -r_i$

Algoritmus 3.9: Dekódování residua.

3.3 Výpočetní složitost

V této krátké sekci bude shrnuta problematika výpočetní složitosti jednotlivých kroků navržené metody. Výpočetní složitost všech kroků navržené metody je závislá na hloubce BLG stromu, která je obecně $\mathcal{O}(n)$, kde n je počet jeho uzlů. Průměrná (a také nejmenší možná) hloubka BLG stromu je však $\mathcal{O}(\log n)$ (viz sekce 1.2.2), proto i průměrná (očekávaná) výpočetní složitost jednotlivých kroků navržené metody bude lepší než jejich výpočetní složitost v nejhorším případě (viz tabulka 3.1).

Operace	Výpočetní složitost		
	Nejlepší	Průměrná	Nejhorší
Vytvoření BLG stromu	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Test „lomená čára - bod“	$\mathcal{O}(1)$	?	$\mathcal{O}(n)$
Test „lomená čára - lomená čára“	$\mathcal{O}(1)$	?	$\mathcal{O}(nm)$
Test „lomená čára“	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Kódování	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Dekódování	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$

Tabulka 3.1: Nejlepší, průměrná a nejhorší výpočetní složitost jednotlivých kroků navržené metody.

Z tabulky 3.1 je patrné, jak se průměrná hloubka BLG stromu odráží v průměrné výpočetní složitosti jednotlivých kroků navržené metody. U testů „lomená čára - bod“ a „lomená čára - lomená čára“ není průměrná složitost uvedena, protože závisí na vzájemné poloze těchto objektů.

Kapitola 4

Implementace

Programová část diplomové práce byla realizována v jazyce C++ (dle standardu C++98) ve formě „header-only“ generické knihovny napsané ve stylu STL (Standard Template Library). Knihovnu lze využít pro

- topologicky konzistentní progresivní přenos vektorových dat,
- topologicky konzistentní simplifikaci vektorových dat,
- kompresi vektorových dat.

Knihovnu tvoří pouze dva soubory. Soubor `BLGTree.h` obsahuje třídy (struktury) reprezentující BLG stromy, jejich uzly a data v těchto uzlech uložená. Soubor `util.h` obsahuje funkce a třídy, které jsou využívány v souboru `BLGTree.h` a také pomocné funkce pro kódování a dekódování. Oba soubory společně s kompletní programovou dokumentací jsou uloženy na CD, které je přiloženo k této práci.

Na úvod je nutné popsat, jakým způsobem je lomená čára reprezentována v paměti počítače. Souřadnice bodů lomené čáry musí být uloženy v libovolné datové struktuře, která splňuje koncept dopředného iterátoru[1] (ideálně pole nebo vektor). V této struktuře musí být body i jejich souřadnice uloženy v pořadí znázorněném na obrázku 4.1.

x_0	y_0	x_1	y_1	...	x_{n-1}	y_{n-1}

Obrázek 4.1: Uložení bodů lomené čáry $P = (p_k)_{k=0}^{n-1}$, $p_k = (x_k, y_k)$.

4.1 Implementace BLG stromu

Základní třídou reprezentující BLG strom je `BLGTree` (viz kód 4.1). Třída `BLGTree` je šablonovaná a její parametry mají následující význam: `DIM`, resp. `T` je dimenze, resp. datový typ souřadnic bodů vstupní lomené čáry a `Data` představuje typ dat uložených v uzlech BLG stromu, které jsou typu `Node`. Třída

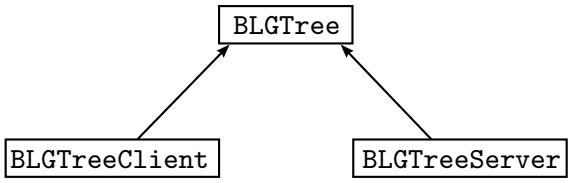
[1] forward iterator

mimo jiné ukládá ukazatel na kořen BLG stromu a koncové body vstupní lomené čáry (nejsou uloženy v BLG stromu).

```
template <unsigned DIM, class T, class Data, class Node>
struct BLGTree {
protected:
  ...
  Node *root;
  T firstPoint[DIM];
  T lastPoint[DIM];
};
```

Kód 4.1: Třída `BLGTree`.

Hlavní funkčností této třídy je pouze zjednodušení lomené čáry reprezentované BLG stromem. Tato třída totiž slouží pouze jako bázová třída speciálních tříd `BLGTreeClient` a `BLGTreeServer` (viz obr. 4.2), které budou dále popsány. Zásadní rozdíl mezi těmito dvěma třídami (vyjma definovaných operací) je v odlišné správě paměti jejich uzlů.



Obrázek 4.2: Diagram tříd.

4.1.1 BLGTreeClient

Třída `BLGTreeClient` je jednodušší než její serverový protějšek `BLGTreeServer`. Hlavní funkčností této třídy (vyjma zděděných operací) je dekódování bodů lomené čáry a jejich vkládání do BLG stromu. Při implementaci této třídy byl kladen důraz na efektivní hospodaření s paměťovými prostředky, aby mohlo být na straně klienta uloženo co nejvíce potřebných dat. Dále se popis veškeré funkčnosti omezí prakticky výhradně na záležitosti související se správou paměti.

```
template <unsigned DIM, class T, class Data = BLGData<DIM, T> >
struct BLGTreeClient
  : public BLGTree <DIM, T, Data, BLGNodeClient<Data> >
{
  ...
  ~BLGTreeClient() {
    Clear();
  }

  void Clear() {
    delete this->root;
    this->root = NULL;
  }
};
```

Kód 4.2: Třída `BLGTreeClient`.

Při pohledu na kód 4.2 se zdá, že destruktor uvolní paměť pouze kořene BLG stromu. Ten je však speciálního typu `BLGNodeClient` (viz kód 4.3), a proto zavolání destruktoru kořene stromu způsobí rekurzivní uvolnění paměti všech uzlů stromu.

```
template <class Data>
struct BLGNodeClient {
  Data *data;
  BLGNodeClient *left;
  BLGNodeClient *right;

  BLGNodeClient() : left(NULL), right(NULL) {
    data = new Data();
  }

  ~BLGNodeClient() {
    delete data; data = NULL;
    delete left; left = NULL;
    delete right; right = NULL;
  }
};
```

Kód 4.3: Třída `BLGNodeClient`.

Data uložená v BLG stromu na straně klienta jsou implicitně typu `BLGData`, jehož definice je následující:

```
template <unsigned DIM, class T>
struct BLGData {
  T point[DIM];
  unsigned index;
  float e2;
  bool required;

  BLGData() : required(false) {}
};
```

Kód 4.4: Třída `BLGData`.

V poli `point` jsou uloženy souřadnice bodu, proměnná `index` určuje pořadí bodu v rámci lomené čáry, proměnná `e2` ukládá druhou mocninu chyby segmentu daného uzlu a proměnná `required` určuje potřebnost daného bodu. Důvodem uložení druhé mocniny chyby je pouze efektivita, která spočívá ve vyhnutí se operaci odmocnění při jejím výpočtu. Mohlo by se zdát vhodné použít pro proměnnou `e2` stejný datový typ, jako mají souřadnice bodů lomené čáry, tedy `T`. Tato úvaha je správná, pokud by byl typ `T` neceločíselný. V případě celočíselného typu by však při výpočtu `e2` mohlo dojít k relativně velké chybě způsobené zaokrouhlením na celé číslo nebo k velice snadnému přetečení rozsahu celočíselného typu. Datový typ hodnoty `e2` je proto neceločíselný, a tedy nezávislý na datovém typu souřadnic bodů.

4.1.2 BLGTreeServer

Třída `BLGTreeServer` reprezentuje BLG strom na straně serveru. Hlavní funkčností této třídy (vyjma zděděných operací) je vytvoření BLG stromu ze zadané lomené čáry, kontrola topologie lomené čáry a kódování její geometrie. Při implementaci této třídy byl kladen důraz na rychlost těchto operací, která je do značné míry ovlivněna způsobem uložení jednotlivých uzlů stromu.

```
template <unsigned DIM, class T,
  class Data = BLGDataServer<DIM, T> >
struct BLGTreeServer
  : public BLGTree <DIM, T, Data, BLGNodeServer<Data> >
{
  ...
  ~BLGTreeServer() {
    Clear();
  }
protected:
  ...
  Data *nodesData;
  Node *nodes;

  void Clear() {
    delete[] nodesData; nodesData = NULL;
    delete[] nodes; nodes = NULL;
  }
};
```

Kód 4.5: Třída `BLGTreeServer`.

Dále bude popsán zásadní rozdíl správy paměti oproti BLG stromu na straně klienta. K vytvoření BLG stromu na straně serveru slouží jeho funkce

```
template <class ForwardIterator>
Create(ForwardIterator first, ForwardIterator last);
```

kde rozsah [`first`, `last`) obsahuje souřadnice bodů lomené čáry zadané ve formátu znázorněném na obrázku 4.1. Protože je znám přesný počet uzlů BLG stromu, jsou z důvodu efektivity v této funkci jednorázově alokovány dva souvislé bloky paměti (pole `nodes` a `nodesData`) pro uložení všech uzlů, resp. jejich dat. Dealokace paměti obou polí je opět jednorázová, a proto není nutné jednotlivé uzly stromu procházet a jeden po druhém odstraňovat z paměti. Z toho důvodu musí třída `BLGTreeServer` použít speciální typ uzlu `BLGNodeServer` (viz kód 4.6), jehož konstruktor i destruktor je prázdný.

```
template <class Data>
struct BLGNodeServer {
  Data *data;
  BLGNodeServer *left;
  BLGNodeServer *right;

  BLGNodeServer() : data(NULL), left(NULL), right(NULL) {}

  ~BLGNodeServer() {}
};
```

Kód 4.6: Třída `BLGNodeServer`.

Data uložená v tomto BLG stromu jsou implicitně typu `BLGDataServer`, jehož definice je následující:

```
template <unsigned DIM, class T>
struct BLGDataServer : public BLGData<DIM, T> {
  typename type_traits::make_signed_integral<T>::type
    residuum[DIM];
  util::bounding_box<DIM, T> bbox;
};
```

Kód 4.7: Třída `BLGDataServer`.

Data každého uzlu tedy dědí atributy ze třídy `BLGData` (viz kód 4.4), navíc však ukládají informaci o residuu a obdélníkové obálce segmentu daného uzlu. Na první pohled není zcela zřejmý datový typ residua. Jak bylo zmíněno v předchozí kapitole, souřadnice residua musí být celočíselné, a to i v případě, že je datový typ `T` souřadnic bodů neceločíselný. Proto pokud je `T` neceločíselný typ, je třeba změnit interpretaci tohoto typu na vhodný celočíselný typ, který bude typem residua. Konkrétně, jestliže `T` bude typu float, resp. double, pak typ residua bude 32bitový, resp. 64bitový znaménkový celočíselný typ. Bude-li `T` znaménkový nebo neznaménkový celočíselný typ, pak bude typ residua znaménkový celočíselný typ stejné velikosti.

4.2 Použití knihovny

Jak bylo zmíněno výše, knihovna je „header-only“ a využívá pouze knihovnu STL. Proto stačí soubory `BLGTree.h` a `util.h` jednoduše přidat k souborům projektu, ve kterém budou funkce knihovny využívány. Příklad použití knihovny (zdrojový kód) je uveden v příloze B.

Jestliže je knihovna použita pouze k topologicky konzistentní simplifikaci vektorových dat, pak není třeba věnovat speciální pozornost způsobu překladu zdrojového kódu. Opačná situace nastává ve chvíli, kdy je knihovna použita ke kompresi nebo progresivnímu přenosu vektorových dat, tedy v situacích, kdy jsou vektorová data kódována do výstupního proudu. V tomto případě je třeba brát v potaz možné problémy, které mohou nastat v průběhu jejich dekódování na jiném počítači, tedy problémy s přenositelností.

Zakódovaná data by měla být přenositelná, pokud jsou v průběhu kódování i dekódování splněny tyto podmínky:

1. Je použito stejné pořadí bajtů při ukládání datových typů do operační paměti (stejný endian).

2. V případě neceločíselných souřadnic vektorových dat je použita aritmetika definovaná standardem IEEE-754.

Obě podmínky jsou zejména díky velkému rozšíření procesorů s architekturou x86 (Intel, AMD) v drtivé většině případů splněny. V ostatních případech nelze zakódovaná data považovat za přenositelná.

Pokud je použit neceločíselný typ souřadnic vektorových dat, může důsledkem nevhodného překladu zdrojového kódu dojít k problému v průběhu je-

jich kódování. Tato situace nastane za předpokladu, že výsledný program využívá pro operace s neceločíselnými datovými typy instrukční sadu x87. Ta totiž v průběhu operací s čísly typu float nebo double tato čísla interně (v registrech) reprezentuje pomocí 80bitového typu s „rozšířenou přesností“. Tato vlastnost může způsobit, že jednotlivé bity uložené v datovém typu float nebo double nemusí odpovídat jeho skutečné hodnotě, která může být dočasně uložena v registrech, a proto může být do výstupního proudu zakódována nesprávná hodnota souřadnic. Spolehlivým řešením tohoto problému je použít místo instrukční sady x87 novější instrukční sadu SSE, která rozšířenou přesnost nepodporuje. Použití instrukční sady SSE lze nastavit jako parametr při kompilaci zdrojového kódu, pro architekturu x86-64 by však měla být tato instrukční sada použita při překladu jako výchozí.

Kapitola 5

Testování

V této kapitole bude popsán proces testování a jeho výsledky. Testování bylo provedeno na reálných geografických datech, přičemž byly sledovány dvě veličiny, a to doba výpočtu jednotlivých kroků navržené metody a velikost komprese.

Měření doby výpočtu Pro měření doby výpočtu byla využita knihovna Boost[1]. Měřen byl pouze čas, který byl v průběhu výpočtu strávený na procesoru (CPU time), nikoliv reálný čas doby výpočtu (wall-clock time). V případě, že doba výpočtu byla příliš krátká na to, aby mohla být přesně změřena[2], byl tentýž výpočet opakován ve smyčce a výsledný čas byl vydělen počtem opakování smyčky.

Měření velikosti komprese Pro určení velikosti komprese byly sledovány tři veličiny, a to počet bitů na bod BPP (bits per point), počet bitů na souřadnice bodu BPC (bits per coordinates) a kompresní poměr CR (compression ratio). Jak je definována veličina BPP popisuje následující vztah

$$\mathrm{BPP} = \frac{\text{počet bitů zapsaných do výstupního proudu}}{\text{počet bodů zapsaných do výstupního proudu}}.$$

Veličina BPP však neřekne mnoho o vlastnostech v této práci popsaného kompresního algoritmu souřadnic, protože kromě samotných souřadnic jsou do výstupního proudu kódovány i jiné informace (viz sekce 3.2.2). V průběhu testování proto byla sledována také veličina BPC, která je definována jako

$$\mathrm{BPC} = \frac{\text{počet bitů kódujících souřadnice zapsaných do výstupního proudu}}{\text{počet bodů zapsaných do výstupního proudu}}.$$

Veličina CR slouží jako hlavní ukazatel velikosti komprese, neboť zcela objektivně vyjadřuje kvalitu kompresního algoritmu. Veličina CR je definována jako poměr

$$\mathrm{CR} = \frac{\text{dimenze bodu} \times \text{počet bitů datového typu souřadnic}}{\mathrm{BPC}}.$$

[1] http://www.boost.org

[2] Doba výpočtu by měla být alepoň 100x delší, než je rozlišení časovače procesoru, které se na OS Windows pohybuje okolo 15 ms.

Pro názorné vysvětlení veličin BPP, BPC a CR poslouží následující krátký příklad: Do souboru o velikosti 1400 bitů bylo zapsáno 10 bodů, jejichž dimenze je 2 (body v rovině) a jejichž souřadnice jsou reprezentovány 64bitovým typem double. Ze 1400 zakódovaných bitů jich bylo 900 použito na zakódování souřadnic bodů. Jednotlivé veličiny budou mít hodnotu:

$$\mathrm{BPP} = \frac{1400}{10} = 140, \qquad \mathrm{BPC} = \frac{900}{10} = 90, \qquad \mathrm{CP} = \frac{2 \times 64}{90} \approx 1.42.$$

Parametry testování Testování bylo provedeno na PC s operačním systémem Windows 7 a procesorem Intel Core 2 Duo T7200. Zdrojový kód testovacího programu byl přeložen překladačem Mingw-w64 (GCC verze 4.9.0) s parametrem optimalizace `-O3`.

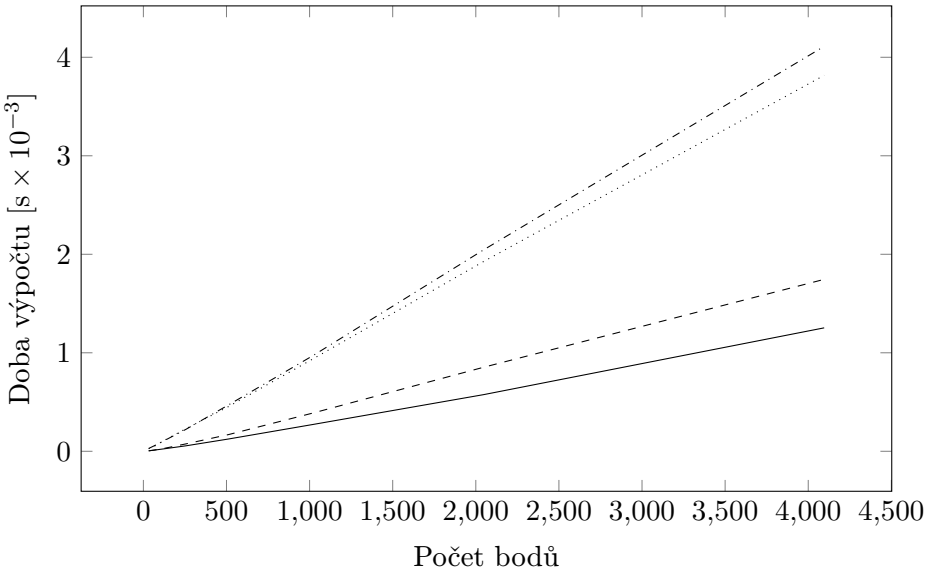
Poznámka V této práci bohužel nebylo provedeno objektivní srovnání s jinými metodami, protože autorovi při jeho nejlepším vědomí není znám způsob, kterým by bylo možné v omezeném čase dosažené výsledky objektivně porovnat s podobnými již existujícími metodami.

5.1 Ověření průměrné výpočetní složitosti

Hlavním záměrem tohoto testu bylo empiricky ověřit průměrnou výpočetní složitost jednotlivých kroků navržené metody. Testovací data byla tvořena hranicemi světadílů, které byly popsány pomocí 15 146 bodů (viz obr. 5.1). Data byla získána z obrázku, který je uložen na serveru `http://commons.wikimedia.org`[3]. Metodika testování byla následující: Z testovacích dat bylo vytvořeno 9 testovacích množin $T_1, T_2, \ldots, T_8$. V každé testovací množině bylo 100 různých lomených čar, přičemž počet bodů všech lomených čar v testovací množině T_i byl 2^{i+4} (např. v množině T_3 bylo 100 lomených čar a každou z nich tvořilo 128 bodů). Lomené čáry dané testovací množiny byly získány z hranic světadílů jako segmenty příslušné délky.

Obrázek 5.1: Grafické znázornění testovacích dat.

[3] `http://commons.wikimedia.org/wiki/File:World_map_blank_without_borders.svg`



Obrázek 5.2: Srovnání doby výpočtu vytvoření BLG stromu (——), kontroly vlastní topologie (- - -), kódování (-·-·-) a dekódování (·····).

Pro každou lomenou čáru z testovací množiny byla měřena doba (1) vytvoření BLG stromu, (2) kontroly vlastní topologie (test „lomená čára“), (3) zakódování lomené čáry do výstupního proudu, (4) dekódování lomené čáry z výstupního proudu. Z naměřených hodnot v rámci testovací množiny byl vypočítán aritmetický průměr. Výsledky testování jsou graficky znázorněny na obrázku 5.2.

Graf na obrázku 5.2 poskytuje nejen srovnání doby výpočtu jednotlivých kroků navržené metody, ale také empirické ověření výpočetní složitosti. V sekci 3.3 bylo uvedeno, že průměrná výpočetní složitost všech kroků algoritmu je $\mathcal{O}(n \log n)$, což potvrzuje i tento test. Z grafu je dále patrné, že operace kódování a dekódování jsou časově nejnáročnější. Tento stav je způsoben relativně velkou časovou náročností zápisu, resp. čtení dat z bitového proudu.

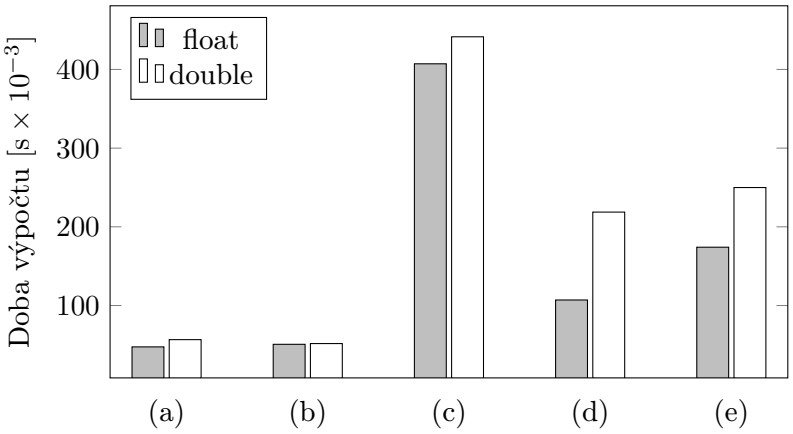
5.2 Test časové náročnosti

Cílem tohoto testu bylo ověřit časovou náročnost jednotlivých kroků popsané metody. Testovací množinu tvořilo 3726 lomených čar, které reprezentovaly hlavní vodní toky České republiky. Geografická data byla získána z digitální vektorové geografické databáze České republiky ArcČR® 500. Označme n_i počet bodů i-té lomené čáry z testovací množiny. Základní statistické vlastnosti testovací množiny udává tabulka 5.1.

$\sum n_i$	$\bar{n}$	$\min(n_i)$	$\max(n_i)$	σ
223 599	60	2	497	54

Tabulka 5.1: Základní statistické vlastnosti testovací množiny.

V průběhu testování byly sledovány tyto veličiny: (1) doba vytvoření BLG stromů všech lomených čar, (2) doba kontroly vlastní topologie všech lomených čar (test „lomená čára“), (3) doba kontroly topologie vůči ostatním lomených čarám (test „lomená čára - lomená čára“), (4) doba kódování kompletní geometrie všech lomených čar do výstupního proudu a (5) doba dekódování kompletní geometrie všech lomených čar z výstupního proudu. Je třeba dodat, že v rámci doby vytvoření BLG stromů není započítána doba čtení testovacích dat ze souboru. Výsledky testu jsou znázorněny v grafu na obrázku 5.3.



Obrázek 5.3: Porovnání absolutní doby (a) vytvoření BLG stromů, (b) kontroly vlastní topologie, (c) kontroly topologie s ostatními lomenými čarami, (d) kódování a (e) dekódování pro všechny lomené čáry z testovací množiny (3726 lomených čar, 223 599 bodů).

Graf zobrazený na obrázku 5.3 znázorňuje dobu výpočtu jednotlivých kroků navržené metody. V tomto případě bylo hlavním cílem udělat si názornou představu o absolutní době výpočtu jednotlivých kroků. Z grafu je patrné, že doba vytvoření BLG stromů a doba kontroly vlastní topologie všech lomených čar byla přibližně stejná. S přihlédnutím k množství zpracovaných dat lze tuto dobu považovat za velmi uspokojivou.

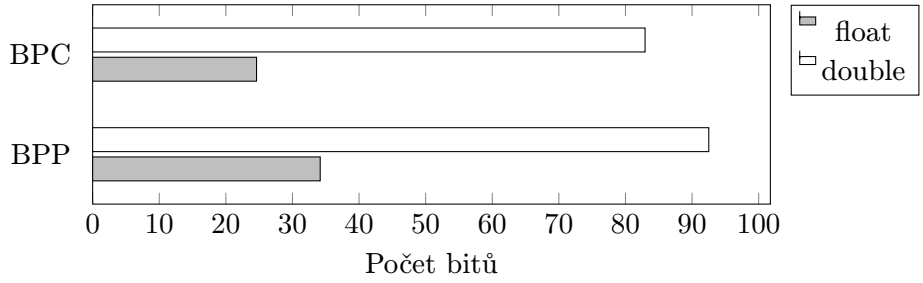
Na první pohled je největší „brzdou“ algoritmu kontrola topologie vůči ostatním objektům. Je však třeba říci, že tuto problematiku navržená metoda neřeší. Navržená metoda totiž řeší pouze algoritmus kontroly topologie mezi dvěma objekty, nikoliv však způsob, jakým jsou vybírány konkrétní dva objekty k tomuto testu. V průběhu testu byla tato problematika řešena „hrubou silou“ stylem „každý s každým“. Výpočetní geometrie však poskytuje efektivnější způsoby řešení, např. vhodné dělení prostoru v závislosti na poloze objektů by mohlo tento krok velmi urychlit, proto by bylo vhodné tuto problematiku v budoucnu prozkoumat.

Tento test také potvrdil, že čas kódování, resp. dekódování patří k pomalejším krokům navržené metody. Je však patrný rozdíl v použití datového typu souřadnic. Vysvětlení je však jednoduché, při použití datového typu float je do bitového proudu zapisováno o mnoho méně dat než v případě použití typu double.

Poznámka V článku [4] zabývajícím se topologicky konzistentní simplifikací uvádějí autoři výsledky jejich experimentu, který zde bude pro velmi hrubé srovnání zmíněn. Experiment byl uskutečněn na testovací množině s vektorovými daty, které tvořilo 166 157 bodů. Autoři uvádějí, že čas výpočtu topologicky konzistentní simplifikace této množiny byl 5.65 s. Čas výpočtu topologicky konzistentní simplifikace pomocí navržené metody byl pro vektorová data tvořená 223 599 body přibližně 0.5 s (součet doby vytvoření BLG stromu, kontroly vlastní topologie a kontroly topologie vůči ostatním objektům). Rychlost navržené metody je při hrubém srovnání několikanásobně vyšší. Hlavní příčinou je pravděpodobně použití obdélníkových obálek, které sice vede k velkému urychlení, ovšem za cenu toho, že mohou být některé body nesprávně označeny za potřebné (viz obr. 3.1). Příkladem této situace může být srovnání topologicky konzistentní simplifikace a simplifikace bez kontroly topologie znázorněné na obrázku A.6, resp. A.7. Z obrázků je patrné, že při použití topologicky konzistentní simplifikace bylo relativně mnoho bodů chybně označeno za potřebné. Bylo by proto vhodné se na problematiku nesprávného označování bodů za potřebné zaměřit v budoucnu.

5.3 Test komprese

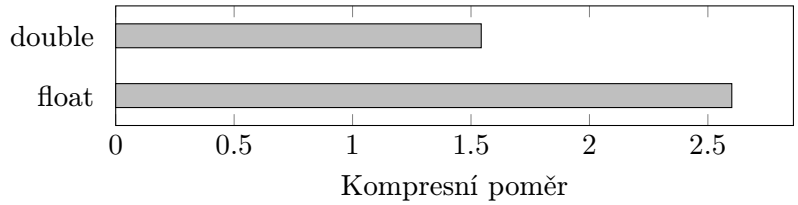
Cílem tohoto testu bylo ověřit kvalitu kompresního algoritmu souřadnic, který používá popsaná metoda. Testovací množina byla stejná jako v předchozím testu, tedy vodní toky ČR (viz tabulka 5.1). V průběhu testování byly sledovány tyto veličiny: (1) počet bitů na bod, (2) počet bitů na souřadnice bodu, (3) kompresní poměr. Výsledky testu jsou znázorněny v grafech na obrázcích 5.4 a 5.5.



Obrázek 5.4: Porovnání hodnot BPP a BPC dosažených při zakódování kompletní geometrie všech lomených čar z testovací množiny pro datový typ souřadnic float a double.

Graf zobrazený na obrázku 5.4 znázorňuje porovnání naměřených hodnot BPP a BPC. Z grafu je možné vyčíst, jaký podíl zakódovaných dat tvoří geometrie (pouze souřadnice bodů) a jaký dodatečné informace (hlavičky, indexy bodů, atd.).

Graf zobrazený na obrázku 5.5 znázorňuje kompresní poměry (CR) při použití datového typu float a double. Vzhledem k tomu, že je komprese bezztrátová a že se kódují desetinná čísla, lze dosažené kompresní poměry považovat



Obrázek 5.5: Porovnání kompresního poměru (CR) dosaženého při zakódování kompletní geometrie všech lomených čar z testovací množiny pro datový typ souřadnic float a double.

za uspokojivé. Navíc je kompresní algoritmus jednoduchý a rychlý. Rozdíl mezi kompresním poměrem dosaženým při použití datového typu float a double je značný. Tuto situaci lze vysvětlit tak, že proces převodu souřadnic na typ double nebo float lze považovat za kvantizaci, přičemž počet kvantizačních úrovní je při převodu na typ float několikanásobně menší, a proto dochází k většímu kompresnímu poměru.

Závěr

V rámci diplomové práce byl podrobně popsán, implementován a na reálných datech otestován algoritmus progresivního přenosu liniových prvků. Popsaný algoritmus v průběhu progresivního přenosu zachovává topologii mezi již přenesenými liniovými prvky, což je zejména v oblasti GIS velmi důležitá vlastnost. Součástí popsaného algoritmu je také bezztrátový kompresní algoritmus, který slouží k redukci objemu přenášených dat. Jádrem popsaného algoritmu je jednoduchá datová struktura BLG strom, která je efektivně využita jak v průběhu kontroly topologie, tak pro realizaci progresivního přenosu. Díky BLG stromu je očekávaná výpočetní složitost jednotlivých kroků algoritmu progresivního přenosu (vytvoření BLG stromu, kontrola topologie, progresivní kódování, dekódování) $\mathcal{O}(n \log n)$, kde n je počet bodů liniového prvku.

Popsaný algoritmus byl implementován v jazyce C++ ve formě „header-only“ generické knihovny. Knihovnu lze využít nejen k progresivnímu kódování liniových prvků, ale také k jejich topologicky konzistentní simplifikaci a kompresi. Funkce, které knihovna nabízí, tedy mohou být užitečné i v jiných oblastech než GIS. V rámci testování na reálných datech, které tvořily vodní toky v České republice ($\sim$225 000 bodů), bylo zjištěno, že se doba trvání všech kroků algoritmu pohybovala řádově v desetinách sekundy. Vzhledem k relativně velkému objemu dat, na kterých byla doba výpočtu testována, výsledky naznačují, že by algoritmus progresivního přenosu liniových prvků mohl být vhodný pro aplikace pracující v reálném čase.

Budoucí práce Ačkoliv byl algoritmus progresivního přenosu liniových prvků testován na reálných datech, nebyl testován v reálné praxi, tedy v situacích, kdy jsou data přenášena přes internet a zobrazována pomocí speciálního programu nebo webového prohlížeče. Testování v tomto prostředí by jistě odhalilo některé nedostatky ve vytvořené knihovně a poskytlo by věrohodnější odpověď na otázku, zda navržená metoda opravdu splňuje požadavky aplikací pracujících v reálném čase.

Literatura

[1] Agarwal, P. K., and Varadarajan, K. R. Efficient Algorithms for Approximating Polygonal Chains. *Discrete & Computational Geometry 23*, 2 (2000), 273–291.

[2] Bertolotto, M. Progressive Techniques for Efficient Vector Map Data Transmission: An Overview. In *Spatial Data on the Web*, A. Belussi, B. Catania, E. Clementini, and E. Ferrari, Eds. Springer Berlin Heidelberg, 2007, pp. 65–84.

[3] Corcoran, P., and Mooney, P. Topologically Consistent Selective Progressive Transmission. In *Advancing Geoinformation Science for a Changing World*, S. Geertman, W. Reinhardt, and F. Toppen, Eds., Lecture Notes in Geoinformation and Cartography. Springer Berlin Heidelberg, 2011, pp. 519–538.

[4] Da Silva, A. C., and Wu, S.-T. A Robust Strategy for Handling Linear Features in Topologically Consistent Polyline Simplification. In *GeoInfo* (2006), pp. 19–34.

[5] de Berg, M., van Kreveld, M., and Schirra, S. Topologically Correct Subdivision Simplification Using the Bandwidth Criterion. *Cartography and Geographic Information Systems 25*, 4 (1998), 243–257.

[6] Douglas, D. H., and Peucker, T. K. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization 10*, 2 (1973), 112–122.

[7] Hershberger, J., and Snoeyink, J. Speeding Up the Douglas-Peucker Line-Simplification Algorithm. Tech. rep., Vancouver, BC, Canada, Canada, 1992.

[8] Li, L., Wang, Q., Zhang, X., and Wang, H. An Algorithm for Fast Topological Consistent Simplification of Face Features. *Journal of Computational Information Systems 9*, 2 (2013), 791–803.

[9] Li, Z. *Algorithmic Foundation of Multi-Scale Spatial Representation*. CRC Press, 2006.

[10] O'Rourke, J. *Computational Geometry in C*, 2nd ed. Cambridge University Press, New York, NY, USA, 1998.

[11] Saalfeld, A. Topologically Consistent Line Simplification with the Douglas-Peucker Algorithm. *Cartography and Geographic Information Science 26*, 1 (1999), 7–18.

[12] Van Oosterom, P. The Reactive-tree: A Storage Structure for a Seamless, Scaleless Geographic Database. *Auto-Carto 10: Technical Papers of the 1991 ACSM-ASPRS Annual Convention*, 6 (1991), 393–407.

[13] Visvalingam, M., and Whyatt, J. D. Line Generalisation by Repeated Elimination of Points. *The Cartographic Journal 30*, 1 (1993), 46–51.

[14] Yang, B., Purves, R., and Weibel, R. Efficient Transmission of Vector Data over the Internet. *Int. J. Geogr. Inf. Sci. 21*, 2 (Jan. 2007), 215–237.

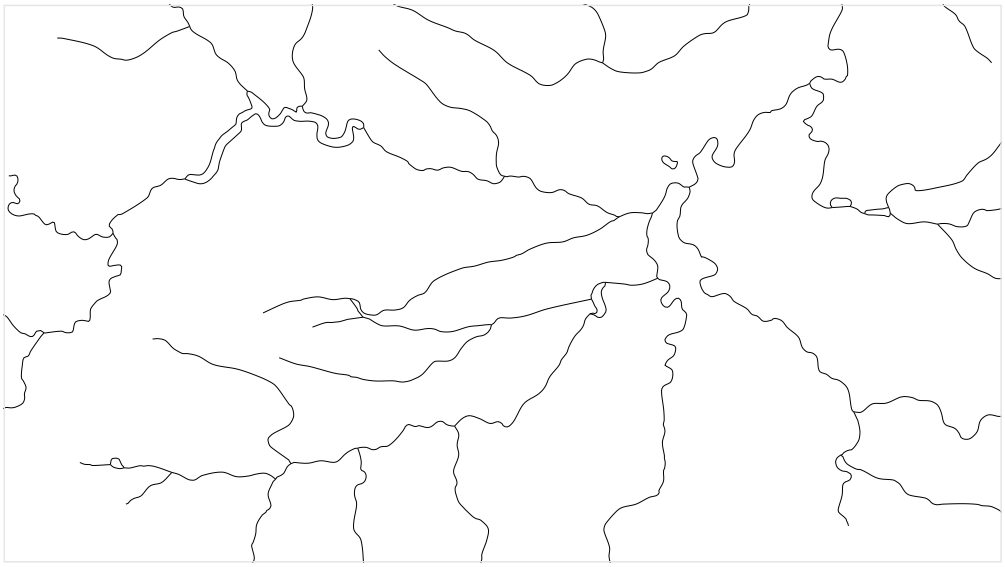
[15] Yang, B., and Weibel, R. Editorial: Some Thoughts on Progressive Transmission of Spatial Datasets in the Web Environment. *Comput. Geosci. 35*, 11 (Nov. 2009), 2175–2176.

Příloha A

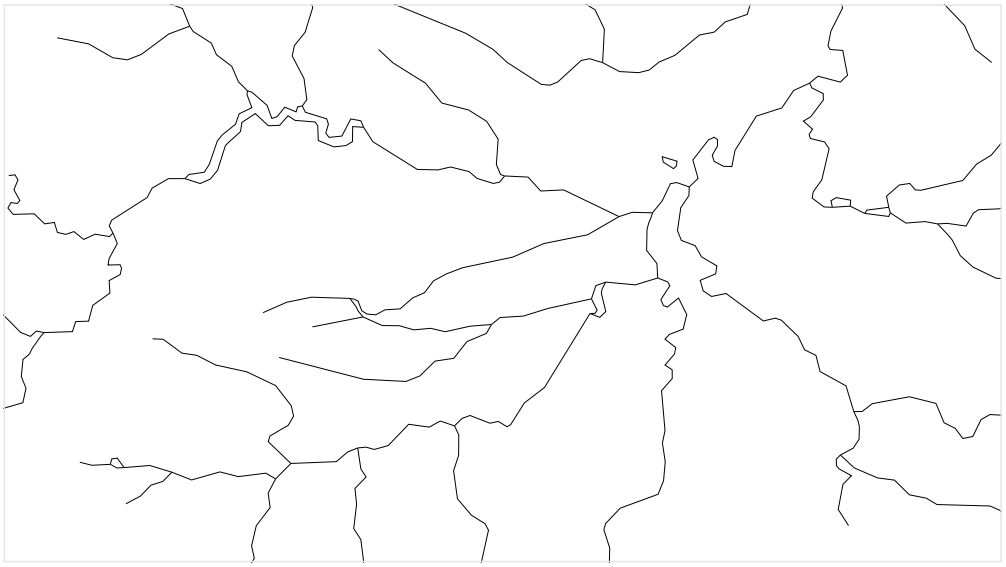
Obrazová příloha

A.1 Ukázka simplifikace

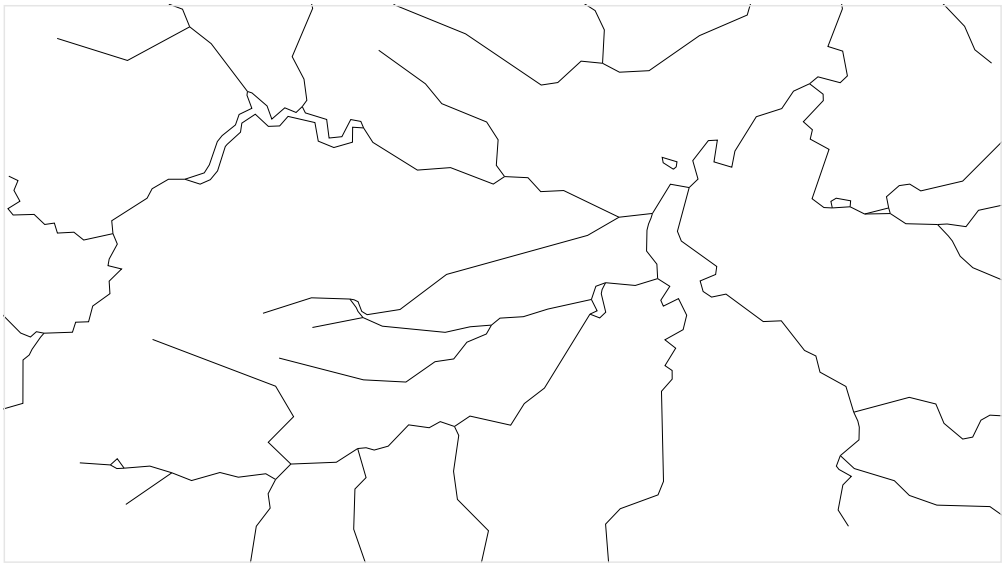
V této části bude názorně ukázána topologicky konzistentní simplifikace vodních toků v části Plzeňského kraje. Součástí každého obrázku je popisek se zvolenou tolerancí zjednodušení a s počtem bodů, které tvořily zjednodušenou říční síť. Obrázek A.7 ilustruje situaci, kdy nebyla v průběhu simplifikace kontrolována topologie.



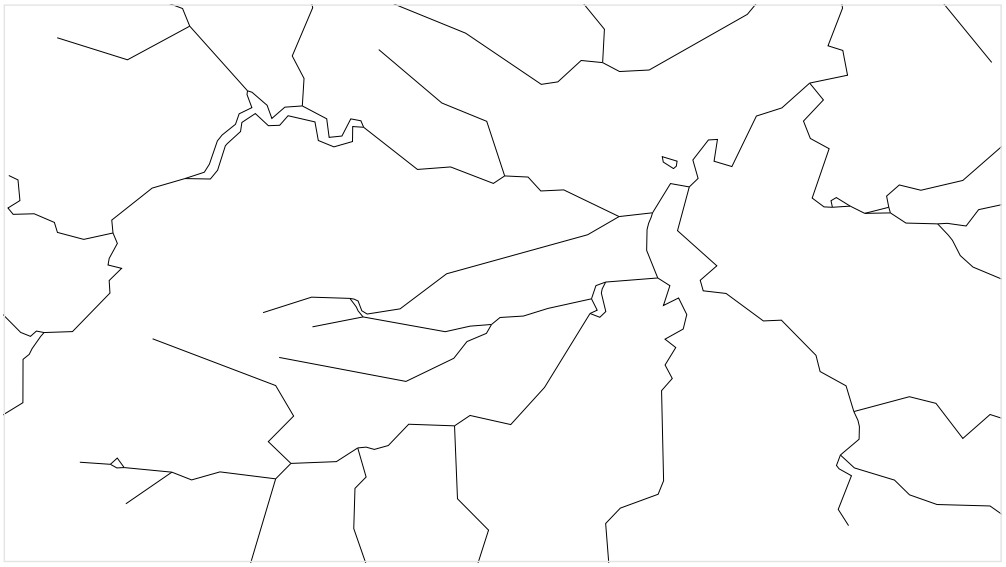
Obrázek A.1: Tolerance: 0 m, počet bodů: 4738.



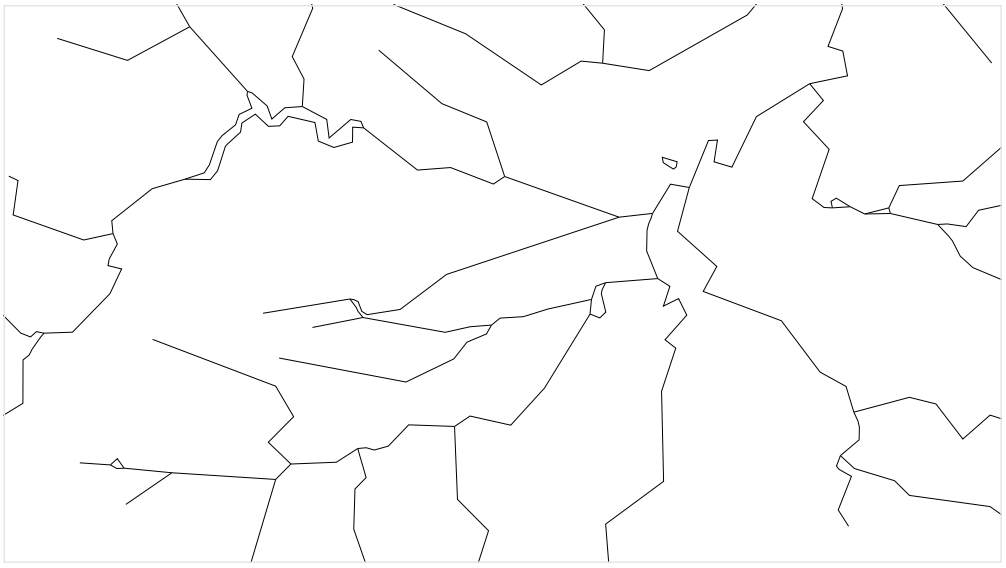
Obrázek A.2: Tolerance: 25 m, počet bodů: 590.



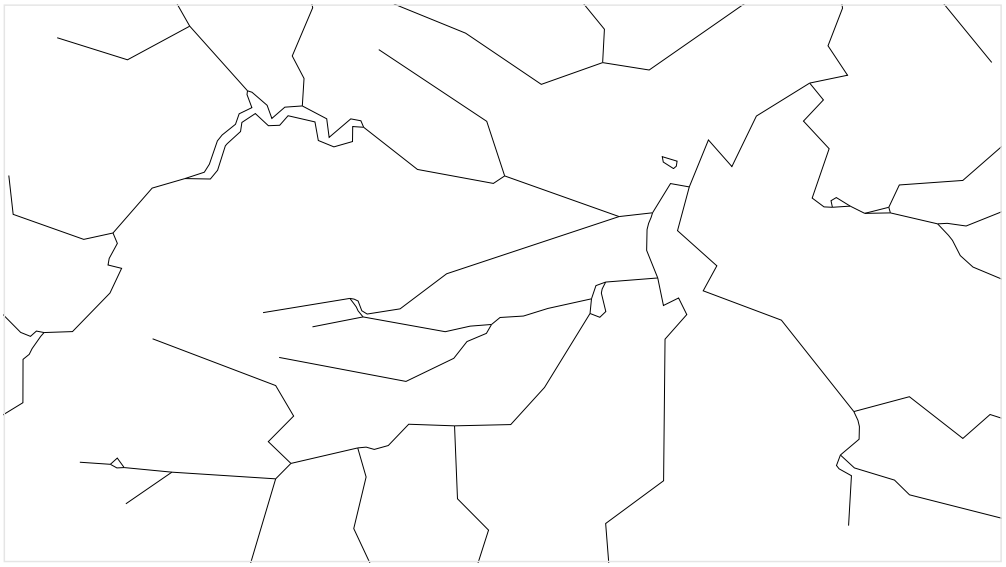
Obrázek A.3: Tolerance: 50 m, počet bodů: 437.



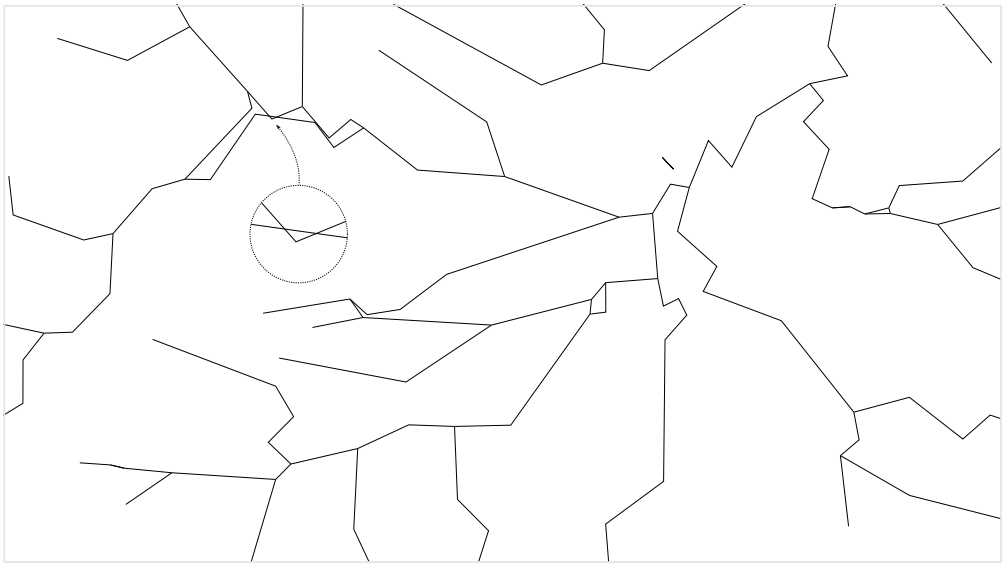
Obrázek A.4: Tolerance: 75 m, počet bodů: 363.



Obrázek A.5: Tolerance: 100 m, počet bodů: 326.



Obrázek A.6: Tolerance: 150 m, počet bodů: 303.



Obrázek A.7: Tol.: 150 m, počet bodů: 209. Simplifikace bez kontroly topologie.

Příloha B

Ukázka zdrojového kódu

V této části bude uveden praktický příklad použití knihovny jak na straně serveru (sekce B.1), tak na straně klienta (sekce B.2).

B.1 Server

Příklad použití knihovny na straně serveru popisuje kód B.1, který zachycuje situaci, kdy je do výstupního proudu kódován jeden bod, jedna lomená čára a jeden polygon. Parametrem funkce `Encode` je výstupní proud, do kterého budou zakódovány objekty `point`, `polyline` a `polygon` tak, aby byla splněna požadovaná tolerance zjednodušení (`tolerance`). Nejprve se z liniových prvků vytvoří BLG stromy, které jsou základem všech následujících operací. Poté se pro oba liniové prvky provede kontrola vlastní topologie (zabránění samoprůnikům) a kontrola topologie vůči ostatním kódovaným objektům. Ještě před tím, než mohou být jednotlivé objekty zakódovány, je nutné z výstupního proudu vytvořit výstupní bitový proud `obs`[1] a nastavit atributy kódovaných objektů (způsob kódování je popsán v sekci 3.2.2). Souřadnice bodu jsou kódovány pomocí funkce `codec::write_point`, zatímco liniové prvky využívají k jejich kódování speciálních funkcí, které poskytuje třída `BLGTreeServer`. Na závěr kódování je nutné ukončit výstupní bitový proud a po jeho ukončení nesmí být do výstupního proudu kódována žádná data.

```
#include "BLGTree.h"

using namespace blgtree; // root namespace of BLGTree.h library

double tolerance; // tolerance of simplification
std::vector<double> point; // coordinates of point
std::vector<double> polyline; // coordinates of polyline
std::vector<double> polygon; // coordinates of polygon
unsigned id0, type0; // id and type of point
unsigned id1, type1; // id and type of polyline
unsigned id2, type2; // id and type of polygon
BLGTreeServer<2, double> t1, t2;
```

[1] Třída `util::obitstream`, resp. `util::ibitstream` je definována v souboru `util.h`.

```
void Encode(std::ostream &os) {
  // create BLG trees
  t1.Create(polyline.begin(), polyline.end());
  t2.Create(polygon.begin(), polygon.end());
  // test topology
  t1.TestTopology(); // to prevent self-intersections
  t2.TestTopology();
  t1.TestTopologyVsPoint(point.begin());
  t1.TestTopologyVsPoly(t2);
  t2.TestTopologyVsPoint(point.begin());
  t2.TestTopologyVsPoly(t1);
  // write objects to output stream
  util::obitstream obs(os); // create output bit stream
  id0 = 0; type0 = 0; // set attributes of point
  id1 = 1; type1 = 1; // set attributes of polyline
  id2 = 2; type2 = 2; // set attributes of polygon
  // write point
  codec::write_header(id0, type0, 0, obs);
  codec::write_point<2>(point.begin(), obs);
  // write polyline
  codec::write_header(id1, type1, 0, obs);
  t1.BeginEncoding(obs);
  codec::write_header(id1, type1, 1, obs);
  t1.WritePointsTol(tolerance, obs);
  // write polygon
  codec::write_header(id2, type2, 0, obs);
  t2.BeginEncoding(obs);
  codec::write_header(id2, type2, 1, obs);
  t2.WritePointsTol(tolerance, obs);
  obs.end(); // important (!) - end of writing bits
}
```

Kód B.1: Příklad použití knihovny na straně serveru.

V případě, že klient zašle serveru žádost o zpřesnění geometrie liniových prvků, není třeba opětovně vytvářet BLG stromy a kontrolovat topologii. Stačí pouze do nově vytvořeného výstupního proudu zakódovat body, které zpřesní geometrii liniových prvků na požadovanou toleranci (viz kód B.2).

```
void Refine(std::ostream &os) {
  util::obitstream obs(os); // create output bit stream
  // refine polyline
  codec::write_header(id1, type1, 1, obs);
  t1.WritePointsTol(tolerance, obs);
  // refine polygon
  codec::write_header(id2, type2, 1, obs);
  t2.WritePointsTol(tolerance, obs);
  obs.end(); // important (!) - end of writing bits
}
```

Kód B.2: Příklad použití knihovny na straně serveru - zpřesnění geometrie.

Z uvedených příkladů vyplývá, že každý bod BLG stromu, který je zapsán do výstupního proudu, je z BLG stromu odstraněn.

B.2 Klient

Příklad použití knihovny na straně klienta popisuje kód B.3. Obecně má klient k dispozici pouze vstupní proud dat, který obsahuje zakódované objekty, nemá tedy k dispozici žádnou informaci o jejich počtu, typu ani velikosti. Dekódované objekty jsou v závislosti na jejich typu (bod, lomená čára, polygon) ukládány do příslušného asociativního pole (mapy), jehož klíčem je identifikátor (id) a hodnotou geometrie, resp. BLG strom objektu. Proces dekódování je popsán funkcí `Decode`, jejíž parametrem je vstupní proud se zakódovanými objekty. Vstupní proud je třeba nejprve změnit na vstupní bitový proud `ibs`. Samotný proces dekódování probíhá ve smyčce, dokud jsou ve vstupním proudu data k dekódování. Nejprve se dekódují atributy objektu, které jsou uloženy v hlavičce. V závislosti na typu dekódovaného objektu se vybere příslušná mapa, ve které je vyhledán záznam dle id dekódovaného objektu. V případě, že je dekódovaný objekt bod a jeho id nebylo v mapě nalezeno, je do mapy tento objekt uložen (běžný případ). Pokud id dekódovaného bodu mapa obsahuje, jsou souřadnice původního bodu přepsány souřadnicemi dekódovaného bodu. V případě, že je dekódovaným objektem liniový prvek, vybere se příslušná mapa a zkontroluje se hodnota atributu `flag`. Pokud `flag` = 1, předpokládá se, že daný objekt je v mapě uložen a dojde ke zpřesnění jeho geometrie (funkce `ReadPoints`). Pokud `flag` = 0, zjistí se, zda mapa obsahuje id dekódovaného objektu. Pokud ne, je do mapy tento objekt uložen a zahájí se jeho dekódování (funkce `BeginDecoding`). V opačném případě se zahájí dekódování již existujícího objektu, čímž dojde k přepsání původní hodnoty.

```
#include "BLGTree.h"

using namespace blgtree;

typedef std::map<unsigned, std::vector<double> > PointsMap;
typedef std::map<unsigned, BLGTreeClient<2, double> > PolysMap;
PointsMap pointsMap; // container to decoded points
PolysMap polylinesMap; // container to decoded polylines
PolysMap polygonsMap; // container to decoded polygons
PointsMap::iterator itPointsMap;
PolysMap::iterator itPolysMap;

void Decode(std::istream &is) {
  util::ibitstream ibs(is); // create input bit stream
  unsigned id, type, flag;
  while(is.peek() != EOF) { // decoding loop
    codec::read_header(id, type, flag, ibs);
    if(type == 0) { // object is point
      // find point with this id
      itPointsMap = pointsMap.find(id);
      if(itPointsMap == pointsMap.end()) { // not found
        std::pair<unsigned, std::vector<double> > pair;
        pair.first = id;
        pair.second.resize(2);
        codec::read_point<2>(pair.second.begin(), ibs);
        pointsMap.insert(pair);
      } else { // found
        codec::read_point<2>(itPointsMap->second.begin(), ibs);
```

```
      }
    } else { // object is polyline or polygon
      PolysMap &map = (type == 1) ? polylinesMap : polygonsMap;
      itPolysMap = map.find(id); // find poly with this id
      if(flag) { // refine existing geometry
        itPolysMap->second.ReadPoints(ibs);
      } else { // decode geometry from scratch
        if(itPolysMap == map.end()) { // not found
          std::pair<unsigned, BLGTreeClient<2, double> > pair;
          pair.first = id;
          pair.second.BeginDecoding(ibs);
          map.insert(pair);
        } else { // found
          itPolysMap->second.BeginDecoding(ibs);
        }
      }
    }
  }
}
```

Kód B.3: Příklad použití knihovny na straně klienta.

V průběhu dekódování má klient uloženy již dekódované objekty v příslušných asociativních polích. Liniové objekty jsou navíc v asociativních polích uloženy ve formě BLG stromu. V případě, že klient požaduje uložit souřadnice již dekódovaných objektů do lineární datové struktury, může použít způsob uvedený v kódu B.4.

```
std::vector<std::vector<double> > points;
std::vector<std::vector<double> > polylines;
std::vector<std::vector<double> > polygons;

// save points from map to vector
itPointsMap = pointsMap.begin();
for(; itPointsMap != pointsMap.end(); itPointsMap++) {
  points.push_back(itPointsMap->second);
}
// save polylines from map to vector
itPolysMap = polylinesMap.begin();
for(; itPolysMap != polylinesMap.end(); itPolysMap++) {
  std::vector<double> polyline;
  itPolysMap->second.ToPoly(std::back_inserter(polyline));
  polylines.push_back(polyline);
}
// save polygons from map to vector
itPolysMap = polygonsMap.begin();
for(; itPolysMap != polygonsMap.end(); itPolysMap++) {
  std::vector<double> polygon;
  itPolysMap->second.ToPoly(std::back_inserter(polygon));
  polygons.push_back(polygon);
}
```

Kód B.4: Příklad uložení dekódovaných objektů do lineární datové struktury.