Masarykova univerzita
Fakulta informatiky

Vyhľadávanie najbližších vektorov s využitím knižnice FLANN

Bakalárska práca

Tomáš Durčák

Brno, Jar 2015

Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Tomáš Durčák

Vedúci práce: RNDr. David Novák, Ph.D.

Poďakovanie

I would like to thank my supervisor . . .

Zhrnutie

The aim of the bachelor work is to provide . . .

Kľúčové slová

keyword1, keyword2 . . .

Obsah

1. Algoritmus pre Hierarchické zhlukovacie stromy

1 Úvod

2 Základné pojmy

2.1 Vektorový priestor

Vektorový priestor $\Omega = D_1 D_2 ... D_n$ má dimenziu n. Objekt (resp. vektor alebo bod) $\mathcal{O} = [a_1, a_2, ..., a_n]$ patriaci do vektorového priestoru je jednoznačne určeny svojimi súradnicami $a_i \in D_i, 1 \leq i \leq n$, ktorých je práve n. Každá jednotlivá dimenzia má svoju doménu D_i - množinu hodnôt (resp. vlastností), ktoré môže prislušná vektorová súradnica nadobúdať. [1]

Vektorový priestor Ω, definujeme nad určitým poľom $\mathbf{P}$, s význačným prvkom $\mathbf{0}$ a dvomi binárnymi operáciami, sčítaním $+ : \Omega \times \Omega \to \Omega$ a násobením $. : \mathbf{P} \times \Omega \to \Omega$, takými že platí:

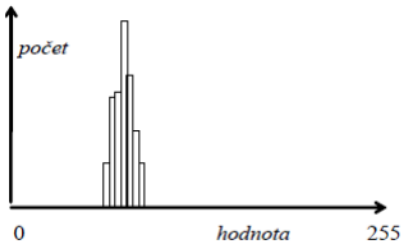
$$\forall u, v, w \in \Omega : (u + v) + w = u + (v + w)$$
$$\exists \mathbf{0} \in \Omega : u + \mathbf{0} = \mathbf{0} + u = u$$
$$\forall u \in \Omega \; \exists \; u \in \Omega : u + (-u) = \mathbf{0}$$
$$\forall a, b \in \mathbf{P} \; \exists \; u \in \Omega : (a.b).u = a.(b.u)$$
$$\forall u \in \Omega : 1.u = u \text{ kde } 1 \in \mathbf{P} \text{ je jednotkový prvok z } \mathbf{P}$$
$$\forall a, b \in \mathbf{P} \; \exists \; u \in \Omega : (a + b).u = a.u + b.u$$
$$\forall a \in \mathbf{P} \; \exists \; u, v \in \Omega : a.(u + v) = a.u + a.v$$

Poznámka: Vektorový podpriestor je tiež vektorový priestor.

Príklad: Obrázok (resp. jeho histogram farieb) može byť vektorom vo vektorovom priestore a mať súradnice podľa počtu pixelov z každého farebného odtieňu. Potom vektor z histogramu farieb je:

$$\{(\alpha_1, \beta_1), ...(\alpha_M, \beta_M)\} \tag{2.1}$$

kde M je počet farebných odtieňov.
Iným príkladom môže byť dokument reprezentovaný ako vektor v m-rozmernom priestore príznakov, ktoré zodpovedajú jednotlivým slovám – tzv.termom. Množina n dokumentov je reprezentovaná ako matica $n \times m$. Neplnovýznamové slová (pomocne slovesá, spojky...) sa zvyčajne odstránia. [2]



Obr. 2.1: Histogram farieb so siedmimi farebnými odtieňmi.

2.2 Metrický priestor

Metrický priestor je množina, na ktorej je definovaná vzdialenosť pre všetky prvky z množiny. Táto vzdialenosť sa nazýva metrika. Metriku môžme definovať ako funkciu, ktorá určuje vzdialenosť medzi dvomi objektami.

Definícia: Nech $X \neq 0$ je množina. Definujme zobrazenie $d : X \times X \to R$, ktoré spĺňa nasledujúce vlastnosti:

$$\begin{aligned} \text{Pre všetky } x, y, z \in X \qquad d(x,x) &= 0 \\ d(x,y) &> 0 \qquad x \neq y \\ d(x,y) &= d(y,x) \\ d(x,z) + d(z,y) &\geq d(x,y) \end{aligned}$$

Množinu X nazývame **základnou množinou**, zobrazenie d **metrikou** a usporiadanu dvojicu (X, d) **metrickým priestorom**. V tej istej množine možme definovať rôzne metriky. Metrika d musí byť vždy nezáporná. [3]

Príklad: Vzdialenosť dvoch bodov na priamke v $\mathbf{R}$ $d = |x - y|$ alebo v rovine v $\mathbf{R^2}$ $d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$.

2.3 Vzdialenostné funkcie

Vzdialenostné funcie predstavuju hlavný spôsob určenia blízkosti alebo podobnosti dvoch objektov v určitej doméne. Táto vzdialenosť može byť definovaná rôzne. Najčastejšie v zavislosti od použitého datového typu alebo účelu aplikácie. Vzdialenostné funkcie rozdeľujeme na dva zakladne typy podľa charakteru ich návratovej hodnoty:

- **diskrétne –** vracajú len vopred určenú množinu hodnôt malého rozsahu, napr. len hodnotu 1 alebo -1
- **spojité –** rozsah vrátenej množiny hodnôt je veľmi veľký alebo aj nekonečno, napr. hodnoty z intervalu $[0, 1]$

Ďalej si popíšeme najčastejšie používané vzdialenostné funkcie. Jednu z nich budeme používať aj v testovaní a porovnávaní testovaných knižníc. [4, 5]

2.3.1 Minkowského vzdialenosť

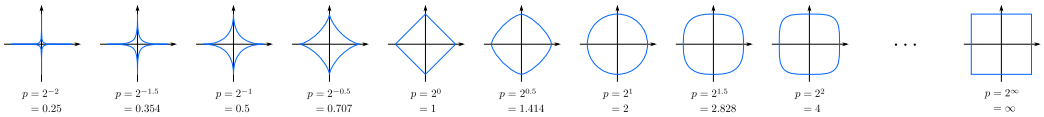
Minkowského vzdialenosť je generalizovaná metrika, ktorá zahŕňa tzv. L_p metriky v zovšebecnenej forme. Minkowskí ju definoval takto:

$$L_p((x_1, ..., x_n), (y_1, ..., y_n)) = \sqrt[p]{\left(\sum_{i=1}^{n} \mid x_i - y_i \mid^p\right)}$$

Rovnica udáva vzdialenosť medzi dvomi objektami v n-dimenzionalnom priestore pre $p \geq 1$ (ak by bolo $p < 1$ nešlo by o metriku). [4]

Medzi najznámejšie a napoužívanejšie metriky patria:

- L_1 – Manhattanská metrika vyjadruje najkratšiu vzdialenosť, ktoru musíme prejsť aby sme sa v meste dostali z jedneho bodu do druhého. Je pomenovaná podľa pravouhélo systému ulíc v meste New York.



Obr. 2.2: Znázornenie L_p- metrik v R^2 podľa parametra p

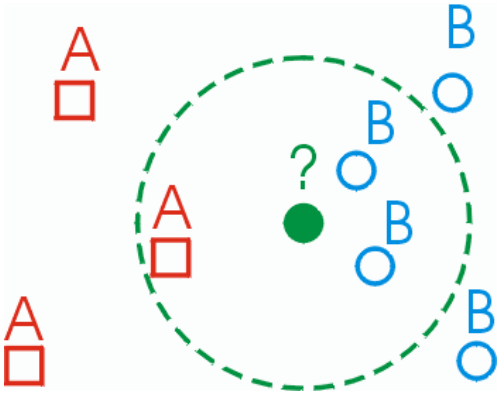
- L_2 – Eklidovská metrika vyjadruje najkratšiu vzdušnú vzdialenosť medzi dvomi objektmi. L_2 je použitá aj pri testoch.

- L_∞ – Čebyševova metrika, nazýva sa tiež maximálna alebo šachovnicová vzdialenosť, pretože na šachovnici, je to minimalny počet ťahov potrebných na prejdenie kráľom z jedného štvorca na iný.

2.4 Vyhľadávanie v metrických a vektorových priestoroch

Základnou charakteristikou vyhľadávania vo vektorových a metrických priestoroch je neurčitosť. Týka sa to hlavne problému **Najdenia najbližšieho suseda** (ang. Nearest neighbor search) [6]. Tento problém obecne definujeme takto:
Je daná množina bodov $P = \{p_1, ..., p_n\}$ v priestore X a dotazované body $q \in M$, nájdite najbližšie body z P ku q.
Pod pojmom **najbližši sused** rozumieme objekt alebo bod podobný dotazovanému bodu. Nepožadujeme aby boli vysledky vyhľadávania vždy na 100% presné, ale aby výpočet skončil v čo najkratšom čase. Pred vyhľadávaním sa musíme rozhodnúť, ako chceme určovať blízkosť objektov, akú metriku použijeme. V našich testoch bola použitá L_2 metrika, čiže klasická Euklidovská metrika, ktorá je vhodná pre husto vzorkované dáta. Ďalej sa musíme rozhodnúť, koľko najbližších susedov budeme hľadať. Existujú 3 základné druhy dotazov:

- **Dotaz na najbližšieho suseda**- hľadáme najližší najpodobnejší objekt k dotazu, výsledkom je 1 objekt ku každému dotazu.



Obr. 2.3: Znázornenie kNN a rNN v L_2-metrike

Definícia:

$$NN(q) = \{x \in X, \forall y \in X | d(q,x) \leq d(q,y)\}$$

Výsledkom je bod x, ktorého vzdialenosť od dotazu q je najmenšia spomedzi všetkých bodov z množiny X.

- **Dotaz na k-najbližších susedov**- niekedy nám nemusí stačiť len 1 najbližší objekt, preto hľadáme k susedov, ktoré sú mu podobné.

  Definícia:

  $$kNN(q) = \{A | A \subseteq X, |A| = k, \forall x \in A, \forall y \in X - A, d(q,x) \leq d(q,y)\}$$

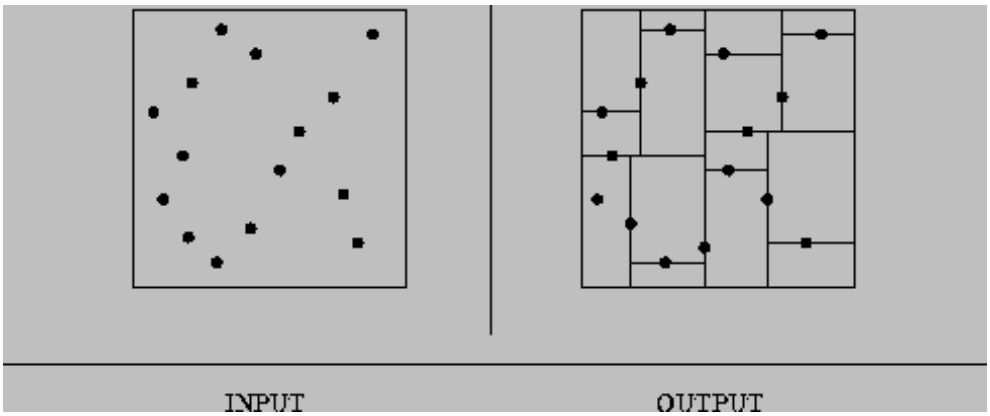
- **Rozsahový dotaz**- použijeme, keď chceme nájsť všetkych najbližších susedov, ktorý su od dotazovaného bodu q vzdialený maximálne zvolenú hodnotu r (polomer vyhľadávania).

  Definícia:

  $$rNN(q, X, r) = \{x \in X | d(q,x) \leq r\}$$

2.5 Indexové štruktúry

Pri vyhľadávaní v metrických alebo vektorových prietoroch je vhodné dáta rozdeliť do menších podmnožin a niektoré data pri prezeraní



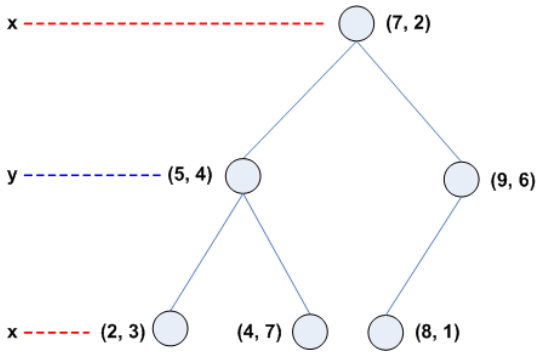
Obr. 2.4: Znázornenie kNN a rNN v L_2-metrike

vynechať, čím sa môže odpoveď na dotaz značne urýchliť. Na roztriedenie dát používame špeciálne indexové štruktúry. Dve z nich si v podkapitolách popíšeme.

2.5.1 KD-stromy

KD = k-dimenzionálny strom [7] je binárny strom slúžiaci na reprezentáciu priestorových dát vyšších dimenzií vo vektorovom priestore. Dáta sú v strome rozdeľované podľa viacdimenzionálnych kľúčov, ktorými su zvyčajne súradnice bodov alebo vektorov.

Vytvorenie KD-stromu
Vstup: Množina objektov v k-dimenzionálnom priestore.
Problem: Vytvoriť strom, ktorý rozdeľuje priestor polrovinami, tzn. každý objekt je vo svojom vlastnom kvádri. Viď Obr. 2.4.
Samotná konštrukcia stromu:
– v každej úrovni stromu vyberáme postupne jeden z koordinátov $\{x_1, x_2, ..., x_k\}$ ako základ rozdeľovania ostaných objektov
– napríklad v koreni stromu sa rozhodujeme podľa x_1 a ako v binárnom vyhľadávacom strome, všetky objekty s hodnotou koordinátu x_1 menšou ako hodnota koreňového uzlu budú v ľavom podstrome a s väčšou alebo rovnou hodnotou budú v pravom podstrom
– v ďalšej podúrovniach stromu triedime objekty postupne podľa



Obr. 2.5: Vkladanie objektu do KD-stromu

ďalších koordinátov, keď už bude mať strom hĺbku k, začíname porovnavať opäť od začiatku podľa $x_1, ...$
– ak by sme vkladali objekty do stromu v nahodnom poradí, strom by bol nevyváženy, preto sa vkladané objekty vyberajú podľa priemernej hodnoty všetkých hodnôt na danom koordináte alebo podľa mediánu
– po log(n) krokoch sa rozklad zastaví, každý objekt má svoje miesto

Vkladanie objektu do stromu:
Príklad: Do stromu na Obr. 2.5 vložíme objekt s kľúčom (11,8)
– v koreni sa podľa koordinátu x porovná s uzlom (7,2), kedže 11>7 ide doprava
– v ďalšej urovní sa porovná podľa y s uzlom (9,6), kedže 8>6 ide doprava, tento uzol je prázdny, preto sa na toto miesto uloži objekt (11,8)

Vyhľadávanie v KD-strome
KD-strom umožňuje vyhľadávať najbližšie body. Na Obr. 2.4 je znázornené ako strom rozdelí vektorový priestor na menšie obdĺžniky – polroviny. Nájdenie najbližšieho objektu teda znamená najdenie oblasti, v ktorej je dotaz q a prehľadanie okolitých oblastí, kde by sa ešte mohol nachádzať ešte bližší bod.
Algoritmus:
–nájde sa bunka c obsahujúca objekt q
–q je ohraničená nejakým objektom p (to ale nemusí byť najližší ob-

jekt)
–nájdeme všetky najbližšie bunky c' vo vzdialenosti $d(p, q)$
–otestujeme či c' neobsahuje bližší objekt

Použitie:
KD-stromy sa používajú hlavne na indexovanie objektov s nižšou dimenziou. Pri vyšších dimenziách sa začne prejavovať tzv. *prekliatie dimenzionality*, ktoré spôsobuje neefektívnosť pri vyhľadávaní.

2.5.2 M-stromy

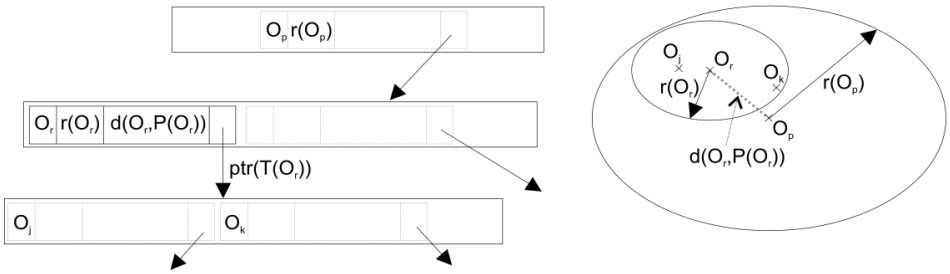
Pre indexovanie objektov v metrických priestoroch sa často využíva datová štruktúra M-strom [1]. Štruktúru stromu tvorí opäť n-arny strom ako u KD-stromu. Rozdiel spočíva v obsahu uzlov a listov stromu. Samotné objekty sú uložené len v listových uzloch stromu. Nelistové ulzy obsahujú tzv. **smerovacie objekty**.

Každý smerovací objekt v uzle obsahuje:

- Samotný objekt O_r, ktorý može byť *virtuálny* – vypočítaný pre účely M-stromu alebo *skutočný* – uloženy v niektorom liste $T(O_r)$
- Odkaz $ptr(T(O_r))$ n svoj podstrom $T(O_r)$
- Polomer $r(O_r)$
- Vzdialenosť $d(O_r, P(O_r))$ od svojho rodičovského smerovacieho objektu $P(O_r)$

Listove uzly vypadajú podobne, ale miesto odkazu na svoj podstrom a polomeru obsahujú $oid(O_j)$ – identifikátor celého objektu.

Princíp hierarchie M-stromu spočíva rovnako ako u KD-stromu v rozdelení priestoru na menšie regióny, ktoré ale nemusia byť nutne disjunktné ako je to u KD-stromu. K rozdeleniu slúžia smerovacie objekty, pričom všetky objekty v listových uzloch, ktoré obsahuje



Obr. 2.6: Uzly M-stromu obsahujúce objekty (vľavo) a regióny smerovacích objektov znazornené v metrickom priestore (vpravo)
Zdroj: [1]

podstrom $T(O_r)$ smerovacieho objektu O_r, sú vo vzdialenosti maximálne $r(O_r)$ od O_r. Teda:

$$\forall O_i \in T(O_r) \qquad d(O_r, O_i) \leq r(O_r)$$

Na obrázku 2.6 je znázornená štruktúra stromu a vzťahy medzi smerovacími objektami.

Vytvorenie M-stromu:
Rovnaká množina objektov môže byť v M-strome indexovaná mnohými rôznymi spôsobmi, pričom neporušíme ani jeden z axiomov metriky alebo vlastnosti M-stromov. Avšak každý z tychto zpôsobov memusí byť práve ideálny z hľadiska efektivity operácií vykonávaných nad M-stromom. Obecne hľadáme také spôsoby konštrukcie stromov, aby zložitost dotazovania bola minimálna. Efektívnosť vyhľadávania v značnej miere ovplyvňuje výber smerovacích objektov tzv. *pivotov*, ktoré potrebujeme na rozdelenie metrického priestoru pri vytváraní stromu. Dôležitý je vyber pivotov, ktorí sú vo vhodnej vzdialenosti od osnatných objektov. Dobrí pivoti by mali byť navzájom ďaleko od seba a taktiež by sa mali nachádzať ďaleko od ostatných objektov v metrickom priestore.

Vyhľadávanie v M-strome:
V M-stromoch môžme vyhľadávať *k-najbližších susedov* ako aj pomocou *rozsahového dotazu*.

Pri vyhľadávaní k-najbližších susedov je dôležitá prioritná rada, z ktorej postupne vyberáme objekty na vyhľadávanie v strome $T(O_r)$. Každý objekt v rade má tvar:
$(prt(T(O_r)), d_{min}(T(O_r)))$. Prvý člen je už známy ukazateľ na podstrom, druhý člen je spodná hranica medzi dotazovaným objektom Q a každým objektom O_j z $T(O_r)$.

Použitie:
M-stromy požívame, keď dáta nie je možné indexovať vo vektorovom priestore, alebo keď sú vektory dát príliš dlhé (dimenzia >20). Metrický priestor umožňuje definovať špeciálne metriky a tým rôzne interpretovať vzdialenosť (resp. podobnosť) objektov na základe ich vlastností.

3 Knižnica FLANN

FLANN[1] je knižnica poskytujúca rýchly programátorsky nástroj, ktorý slúži na *vyhľadávanie najbližších susedov*. Pre tento účel obsahuje kolekciu nástrojov (algoritmov) a tiež systém automatického výberu najlepšieho algoritmu a optimálnych parametrov použitých pri vyľadávani najbližších susedov. Aby bolo vyhľadávanie čo najrýchlejšie, je knižnica FLANN napísaná v programovacom jazyku C++ a je možné ju použiť aj pri programovaní v jazykoch C, Matlab, Python a Ruby. Knižnica je voľne šíriteľná pod licenciou BSD[2].[8]

V ďalších podkapitolách si popíšeme algoritmy *priority search k-means tree*[3] a *multiple randomized k-d trees*[4], ktoré sú obsiahnuté v knižnci FLANN a javia sa najrýchlejšie z pomedzi existujúcich algoritmov na vyhľadávanie najbližších susedov.

3.1 Algoritmus náhodných KD-stromov

Klasické algoritmy využívajúce KD-stromy sú vhodné na vyhľadávanie najbližších susedov v nízko-dimenzionálnych dátach, avšak pre vyššie dimenzie vykazujú značný pokles výkonu. Tento problém viedol k vývoju nového a vylepšeného algoritmu využívajúceho KD-stromy. [8]

Algoritmus je založený na stavaní mnohopočetných nahodných KD-stromov, ktoré sú prehľadávané paralelne. Originálne KD-stromy pri výtvaraní delia dáta na polovicu v každej úrovni stromu vždy postupne podľa dimenzií zaradom. Na porovnanie nahodne KD-stromy vyberajú deliacu dimenziu náhodne z prvých D dimenzií, kde majú dáta najväčší rozptyl. Knižnica FLANN používa v implementácií algoritmu hodnotu $D = 5$, ktorá sa pri testoch ukázala ako optimálna. Pri vyhľadávani sa používa prioritná rada skrz všetky

1. Z anglického: Fast Library for Approximate Nearest Neighbors
2. BSD licencia sa používa pre voľne šíriteľný softvér, je pomenovaná podľa Berkeley Software Distribution, UNIXový operačný systém
3. Algoritmus pre prioritné K-means stromy
4. Algoritmus náhodných KD-stromov

randomizované KD-stromy. Objekty v tejto rade sú zoradené podľa zvyšujúcej sa vzdialenosti od deliacej hranice v každej úrovni, čo zvyšuje rychlosť vyhľadávania. Každý objekt, ktorý už bol porovnaný s dotazom v niektorom strome, je označený a v ďalších stromoch sa už neporovnáva. Presnosť aproximácie vyhľadávania sa reguluje nastavením maximálneho počtu listových uzlov (skrz všetky stromy), ktoré sa pri vyhľadávaní skontrolujú. [8]

3.2 Algoritmus pre prioritné K-means stromy

Pri niektorých typoch dát, može byť efektívnejší algortmus K-means stromov, vzlášt ak je prioritou vysoká presnosť výsledkov. Algoritmus prioritných K-means stromov sa snaží lepšie využívať prirodzenú štruktúru vstupných dát. Na rozdiel od randomizovaných KD-stromov, delí a zoskupuje data do skupín za základe vzdialeností skrz všetky dimenzie, pričom KD-stromy využívajú na delenie v každej úrovni len jeden rozmer. [8]

3.2.1 Opis algoritmu

Pri tvorení K-means stromu sa dáta delia v každej úrovni do K rôznych regiónov pomocou *K-means zhlukovacieho algoritmu*. Rovnaká metóda sa rekurzívne aplikuje na každu novú skupinu dát. Delenie končí, keď počet objektov v každom regióne je menší ako hodnota K. Presnejší popis a štruktúra algoritmu [8].

3.2.2 Výhľadávanie v strome

K-means strom sa prehľadáva postupne od koreňa k najbližšiemu listu. Vetvy, ktoré boli pri prechádzaní stromu preskočené sa ukladajú do prioritnej rady, zoradené podľa vzdialenosťi k dotazovanému objektu. Algoritmus potom ešte skontroluje podstromy uložené v tejto rade. Presnejší náhľad algoritmu je opäť v [8].

Počet regiónov K, na ktoré sa vstupné dáta delia pri vytváraní stromu určuje parameter algoritmu nazvaný *počet vetiev stromu*. Vý-

ber optimálneho K ovplyvňuje rýchlosť a správnosť vyhľadávania. Ďalším parametrom je I_{max} a určuje maximálny počet iterácií, ktoré sa vykonajú pri delení stromu na regióny. Menší počet iterácié urýchli stavbu stromu, ale zníži presnosť vyhľadávania. Posledným parametrom je výber algoritmu, ktorý určí ako sa bude počítať rozhodovacia hranica pri delení objektov do regiónov. K dispozícií mame: *náhodný výber*, *Gonzalesov algoritmus* (výber objektov, ktoré su ďaleko od seba) alebo *KMeans++ algoritmus* [9]. [8]

3.3 Algoritmus The Hierarchical Clustering Tree[5]

Prechádzajúce dva algoritmy sú určené predovšetkým na vyhľadávanie najbližších susedov vo vektorových dátach. Niesu však vhodné na vyhľadávanie v binárnych dátach. Pre tento účel bol vyvinutý nový algoritmus, ktorý využíva novú datovú štruktúru: *hierarchické zhukovacie stromy*. [10]

Vytváranie hierarchického zhlukovacieho stromu je podobné K-means stromu. Zo vstupných dát sa nahodne vyberie K *zhlukovacích centier* a objekty sa rozdelia do K zhlukov podľa toho, ku ktorému zhlukovaciemu centru sú najbližšie. Hodnota K je vstupný parameter algoritmu. Delenie potom prebieha rekurzívne na všetkych nových zhlukoch, až kým nie je počet objektov v zhluku menši ako K. Vtedy sa vytvoria listové uzly stromu. [10]

Na rozdiel od algoritmu pre prioritné K-means stromy, nevytvárame zo vstupných dát len jeden strom, ale celý les stromov, v ktorom sa vyhľadáva paralelne. Aby bolo vytváranie stromov rýchlejšie, je výhodné vyberať zhukovacie centrá náhodne, pričom nedochádza k výraznému poklesu presnosti. Vyhľadávanie je podobné algoritmu pre K-means stromy. [10]

Pred vytváraním stromu možme nastaviť *rozvetvenie stromu (K)*, *počet stromov*, *maximálny počet objektov v listových uzloch* a *algoritmus pre výber prvých K zhukovacích centier*. [10]

5. Algoritmus pre Hierarchické zhlukovacie stromy

3.4 Automatický výber algoritmu

Výber správneho algoritmu a optimálnych parametrov pri vyhľadávanie najbližších susedov nie je triviálny problém. Správný voľba závisí od niekoľkých faktorov ako napríklad: štruktúra vstupných dát a zvolená presnosť vyhľadávania. Každý z možných algoritmov má množinu nastaviteľných parametrov, ktoré v značnej miere ovplyvňujú výsledky vyhľadávania. Je to napríklad počet náhodných stromov pri použití KD-stromu alebo počet vetiev každého uzla pri hierarchickom K-means strome. [11]

Výber algoritmu je optimalizačný problém, ktorým sa snažíme nájsť najlepšie riešenie. Cenú výpočtu počítame ako kombináciu času potrebného na vyhľadávanie dotazov, času za ktorý sa vytvorí vyhľadávací index (strom) a pamäťovej záťaže. Každy z týchto faktorov má inú váhu v závislosti na aplikácií a použití. Ak vytvárame vyhľadávací index len raz, ale vyhľadávanie prebieha mnohokrát, je dôležitejší čas vyhľadávania ako čas potrebný na vytvorenie indexu. Niekedy vyhľadávame v indexe len raz, napríklad pri on-line aplikáciach, vtedy potrebujeme aby sa index vytvoril čo najrýchlejšie. Môžu nastať aj situácie, kedy potrebuje čo najmenšie pamäťová zaťaženie.Váha pamäťového zaťaženie je w_m a váha času vytvorenie indexu je w_b. Celkovú cenu počitame pomocou:

$$\text{cost} = \frac{s + w_b b}{(s + w_b b)_{opt}} + w_m m \tag{3.1}$$

kde s je čas vyhľadávania a b je čas potrebný na vytvorenie vyhľadávacieho indexu. Parameter $m = m_t / m_d$ reprezentuje pomer pamäte použitej na uloženie indexu (m_t) a uloženia dát (m_d). Parameter w_b reguluje čas potrebný na vytvorenie indexu vzhľadom ku času vyhľadávania. Ak nastavíme $w_b = 0$, znamená to, že chceme čo najrýchlejšie vyhľadávať a nezaujíma nás rýchlosť vytvorenia indexu. Nastavením $w_b = 1$, priradíme času vyhľadávania a vytvorenia indexu rovnaku doležitosť. Podobne w_m nastavuje prioritu medzi časom (čas vyhľadávania vytvorenia indexu) a pamäťou, ktorú využiva vyhľadávací index. Hodnota $w_m < 1$ kladie väčšiu váhu na čas nehľadiac na pamäť a hodnota $w_m > 1$ kladie prioritu na množstvo

využitej pamäte. Nastavením $w_m = 1$ priradíme rovnakú prioritu času aj využitiu pamäte. [11]

Výber najlepšieho algoritmu a optimálnych parametrov sa vykonáva v dvoch ktorokoch: V prvom kroku vyberáme algoritmus s najlepšimi výsledkami podľa rovnice 3.1. Pri testovaní sa využívaju hodnoty z množiny {1,4,8,16,32} ako počet náhodných KD-stromov, {1,5,10,15} ako počet iterácií a {16,32,64,128,256} ako počet vetiev v K-means strome. V druhom kroku sa využíva Nelder-Mead downhill simplex metóda [12], pomocou ktorej sa určia optimálne parametre pre algoritmus vybraný v prvom kroku. Optimalizácia može byť počítaná na celých vstupných dátach, alebo len na časti. Typicky stačí $5 - 10\,\%$ a výsedky sa stále veľmi blížia optimálnym hodnotám (ak sú dáta podobnej štruktúry). [11]

3.5 Použitie knižnice FLANN v C++

Jadro knižnice FLANN je naprogramované v programovacom jazyku C++, preto je na jej samotnú kompiláciu potrebný C++ prekladač. Knižnicu je možne použiť Linuxovými distribúciami ale aj Windowsom a OS X. Pre viac-jadrovú podporu je nutné mať nainštalovanú knižnicu OpenMP. Predkompilovanú knižnicu FLANN je možné použiť aj pomocou knižnice PCL (Point Cloud Library)[6], do ktorej je implementovaná jedna zo starších verzií. V d'alších častiach si popíšeme niektoré základné triedy a funkcie knižnice. [13]

3.5.1 Trieda flann::Index

Táto trieda tvorí základ knižnice. Je abstraktnou triedou pre všetky typy indexov a je šablonovaná na rôzne druhy vzdialenostných funkcií. [13]

Konštruktor:

```
Index(const IndexParams& params, Distance distance = Distance() );

Index(const Matrix<ElementType>& points, const IndexParams& params,
      Distance distance = Distance() );
```

6. http://www.pointclouds.org

Trieda **Matrix<ElementType>** obsahuje uložené objekty, z ktorých sa má vytvoriť vyhľadávací index. Objekty sú uložené v poradí za sebou. [13]

IndexParams je štruktúra, ktorá obsahuje parametre indexu. Typ indexu, ktorý sa vytvorí zavysí na tomto parametri. IndexParams môže mať rôzne typy:

- **LinearIndexParams** Index s týmto typom vyhľadáva najbližšie objekty pomocou lineárneho vyhľadávania. [13]
- **KDTreeIndexParams** Index s týmto typom vytvorí randomizované KD-stromy, ktoré sa prehľadávajú pararelne. V štruktúte je možné nastaviť počet stromov (`trees`), ktoré sa majú vytvoriť. [13]

```
struct KDTreeIndexParams : public IndexParams
{
        KDTreeIndexParams( int trees = 4 );
};
```

- **KMeansIndexParams** Index s týmto typom vytvorí hierarchický k-means strom.

```
struct KMeansIndexParams : public IndexParams
{
        KMeansIndexParams( int branching = 32,
                int iterations = 11,
                flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,
                float cb_index = 0.2 );
};
```

  V štruktúre môžme nastaviť rozvetvenie stromu (`branching`), počet iterácií (`iterations`) alebo algoritmus na výber deliacej hranice (`centers_init`). Hodnota `cb_index` určuje, ktore objekty sa pri prechádzaní stromu porovnávajú ako prvé. Ak je `cb_index` = 0, uprednostňujú sa objekty bližšie k deliacej hranici. [13]

- **CompositeIndexParams** Index s týmto typom je kombináciou randomizovaných KD-stromov a hierarchického K-means stromu. [13]

```
struct CompositeIndexParams : public IndexParams {
        CompositeIndexParams( int trees = 4,
                int branching = 32,
                int iterations = 11,
```

```
                flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,
                float cb_index = 0.2 );
};
```

- **KDTreeSingleIndexParams** Index s týmto typom vytvorí jeden KD-strom, ktorý sa hodí a je optimalizovaný na vyhľadávanie v nízkodimenzionálnych dátach (napríklad 2D a 3D priestor).

```
struct KDTreeSingleIndexParams : public IndexParams
{
        KDTreeSingleIndexParams( int leaf_max_size = 10 );
};
```

  `leaf_max_size` nastavuje maximálny počet objektov v listových uzloch. [13]

- **KDTreeCuda3dIndexParams** Index s týmto typom vytvorí jeden KD-strom. Vytvorenie a vyhľadávanie v strome prebieha na CUDA kompatibilných grafických kartách. Tento typ indexu sa hodí na vyľadávanie veľkého počtu dotazov. [13]

- **HierarchicalClusteringIndexParams** Index s týmto typom použíje na vyhľadávanie hierarchické zhlukovacie stromy. [13]

```
struct HierarchicalClusteringIndexParams : public IndexParams
{
        HierarchicalClusteringIndexParams(int branching = 32,
                flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,
                int trees = 4,
                int leaf_max_size = 100)
};
```

- **AutotunedIndexParams** Tento typ indexu je určený pre automatický výber najlepšej vyhľadávacej štruktúry a jej optimálnych parametrov. [13]

```
struct AutotunedIndexParams : public IndexParams
{
        AutotunedIndexParams( float target_precision = 0.9,
                float build_weight = 0.01,
                float memory_weight = 0,
                float sample_fraction = 0.1 );
};
```

  Parameter `target_precision` určuje požadovanú presnosť vyhľadávania, parametre `build_weight` a `memory_weight` su podrobnejšie vysvetlené v podkapitole 3.4, kde predstavujú parametre w_b a w_m. Parameter `sample_fraction` može byť hodnota

medzi 0 a 1, a určuje koľko percent objektov, v ktorých sa má vyhľadávať, sa využije na výpočet optimálneho algoritmu a parametrov. [13]

- **SavedIndexParams** Umožňuje načítať predtým vytvorený a uložený index. [13]

```
struct SavedIndexParams : public IndexParams
{
        std::string filename;
};
```

3.5.2 Metodý triedy flann::Index

- **flann::Index::buildIndex** Metóda vytvorí vyhľadávací index. K dispozícii sú dve preťažené varianty, jedna vytvorí index z dát poskytnutých v konštruktore Indexu, druhá varianta použije dáta z argumetu metódy. [13]

```
void buildIndex();
void buildIndex(const Matrix<ElementType>& points);
```

- **flann::Index::addPoints** Metóda umožňuje pridať objekty do indexu, potom ako už bol vytvorený. Hodota `rebuild_threshold` umožňuje nastaviť, kedy sa má index prestavať, špeciálne pri veľkom počte pridávaných objektov. Prednastavená hodnota 2 znamená prestavanie indexu pri zdvojnásobení počtu objektov. [13]

```
void addPoints(const Matrix<ElementType>& points,
        float rebuild_threshold = 2);
```

- **flann::Index::removePoint** Metóda umožňuje odstrániť z indexu objekt s jedinečným identifikačným číslom `point_id`. [13]

```
void removePoint(size_t point_id);
```

- **flann::Index::getPoint** Metóda vráti ukazateľ na objekt s identifikačným číslom `point_id`. Index je neskôr možne opätovne použiť. [13]

```
ElementType* getPoint(size_t point_id);
```

- **flann::Index::save** Metóda uloží vytvorený index do súboru `filename`. [13]

```
void Index::save(std::string filename);
```

- **flann::Index::knnSearch** Metóda na vyhľadávanie K-najbližších susedov pre množinu dotazovaných objektov, ktoré su uložene v objekte `queries` triedy `Matrix`. Argumenty `indices` a `dists` slúžia a uloženie výsledkov vyhľadávania. Argument `knn` určuje počet najbližších susedov, ktorý sa maju vyhľadať.

```
int Index::knnSearch(const Matrix<ElementType>& queries,
        Matrix<int>& indices,
        Matrix<DistanceType>& dists,
        size_t knn,
        const SearchParams& params);
```

  Argument `params` je štruktúra, ktorá nastavuje špecialne parametre použité počas vyhľadávania.

```
struct SearchParams
{
        SearchParams(int checks = 32,
                float eps = 0,
                bool sorted = true);
        int checks;
        float eps;
        bool sorted;
        int max_neighbors;
        tri_type use_heap;
        int cores;
        bool matrices_in_gpu_ram;
};
```

  Hodnota `checks` určuje počet listových uzlov, ktoré sa maju skontrolovať a može byť nastavená na špecialne hotnoty: `FLANN_CHECKS_UNLIMITED` (skontroluje všetky listové uzly), `FLANN_CHECKS_AUTOTUNED` (použije vypočítanú hodnotu, ak bol index vytvorený automaticky). Môžme tiež nastaviť, či majú byť výsledky zoradené podľa vzdialeností (`sorted = true`) alebo počet jadier procesora, použitých pri vyhľadávaní (`cores`). [13]

- **flann::Index::radiusSearch** Metóda na rozsahové vyhľadávanie najbližších objektov do vzdialenosti polomeru `radius`. [13]

```
int Index::radiusSearch(const Matrix<ElementType>& queries,
        Matrix<int>& indices,
        Matrix<DistanceType>& dists,
        float radius,
        const SearchParams& params);
```

- **flann::hierarchicalClustering**

- **flann::KdTreeCuda3dIndex**

4 Výsledky testov

5 Záver

Literatúra

[1] M. Krátký, T. Skopal, and V. Snáše. Porovnání některých metod pro vyhledávání a indexování multimediálních dat. In *Zbornik konference Kybernetika - história, perspektívy, teória a prax*, pages 118 – 134. Žilinská univerzita v Žiline, 2003.

[2] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[3] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[4] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

[5] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, September 2001.

[6] Alexandr Andoni. *Nearest Neighbor Search: the Old, the New, and the Impossible*. PhD thesis, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2009.

[7] Ashraf M. Kibriya and Eibe Frank. An empirical comparison of exact nearest neighbour algorithms. In *Proc 11th European Conference on Principles and Practice of Knowledge Discovery in Databases*, Warsaw, Poland, pages 140–151. Springer, 2007.

[8] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[9] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[10] Marius Muja and David G. Lowe. Fast matching of binary features. In *Computer and Robot Vision (CRV)*, pages 404–410, 2012.

[11] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[12] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[13] Marius Muja and David G. Lowe. *FLANN - Fast Library for Approximate Nearest Neighbors User Manual*. UBC.