Univerzita Karlova

Přírodovědecká fakulta

Studijní program: Geoinformatika, kartografie a dálkový průzkum Země

Petr Havel, Jan Bartůšek

Prostorová indexace

1. Zadání

Cílem této semestrální práce bylo navrhnout, implementovat a porovnat algoritmy pro efektivní prostorové vyhledávání nad 3D mračnem bodů. Klíčovým omezením zadání byla nemožnost využití specializovaných externích knihoven pro prostorovou indexaci (např. scipy.spatial.KDTree nebo rtree), což vyžadovalo vlastní implementaci datových struktur v jazyce Python.

Úloha byla rozdělena do následujících kroků:

1. Načtení a příprava dat: Vstupem je nestrukturovaný textový soubor se souřadnicemi bodů $p_i = [x_i, y_i, z_i]$
2. Implementace vyhledávacích metod (Nearest Neighbor Search):
   - Naivní metoda: Referenční řešení hrubou silou.
   - Voxelizace (Grid): Rozdělení prostoru na pravidelnou mřížku.
   - KD-Tree: Hierarchická stromová struktura dělící prostor nadrovinami.
3. Výpočet geometrických charakteristik: Pro každý bod mračna byly na základě jeho $k = 30$ nejbližších sousedů vypočteny:
   - Prostorová hustota: Odvozená z průměrné vzdálenosti d_aver k nejbližšímu bodu:

     $$\rho = 1/d_{aver}^3$$

   - Aproximovaná křivost (κ): Určena metodou analýzy hlavních komponent (PCA) z vlastních čísel kovarianční matice ($\lambda_1, \lambda_2, \lambda_3$):

     $$\kappa = \lambda_1/(\lambda_1 + \lambda_2 + \lambda_3)$$

4. Analýza a vizualizace: Porovnání výpočetní náročnosti (benchmark) a vizualizace morfologie mračna na základě vypočtené křivosti.

2. Bonusové úlohy

Pro získání plného počtu bodů a ověření pokročilejších metod jsme implementovali následující rozšíření:

- Akcelerované hledání s využitím Octree (vlastní implementace): (+15 bodů) Implementovali jsme plnohodnotný oktalový strom (Octree), který rekurzivně dělí 3D prostor na 8 pod-krychlí (oktantů). Tato struktura je přirozeným 3D ekvivalentem 2D Quad-tree a je obzvláště vhodná pro data s nerovnoměrnou hustotou rozložení. Na rozdíl od KD-stromu, který dělí prostor vždy na poloviny podle počtu bodů (mediánu), Octree dělí geometrický prostor pravidelně na poloviny délky hrany.

3. Teoretická část

Zpracování velkých mračen bodů (LiDAR, fotogrammetrie) naráží na limity výpočetního výkonu při hledání sousedních bodů. Tato kapitola popisuje teoretický základ použitých metod.

3.1 Problém naivního hledání

Nejjednodušším způsobem, jak najít k nejbližších sousedů pro bod Q, je vypočítat vzdálenost $d(Q, P_i)$ ke všem ostatním bodům P_i v mračnu a vybrat ty s nejmenší hodnotou.

Pokud má mračno N bodů a hledáme sousedy pro každý z nich, musíme provést $N \times (N-1)$ výpočtů vzdálenosti. Časová složitost je tedy kvadratická: $O(N^2)$

Pro $N = 10^5$ to znamená $10^{\{10\}}$ operací, což je i na moderním hardware časově neúnosné (řádově minuty až hodiny).

3.2 Voxelizace (Grid Index)

Metoda Regular Grid (Voxelizace) transformuje spojitý prostor na diskrétní mřížku buněk (voxelů).

- Princip: Prostor ohraničující mračno (Bounding Box) se rozdělí na $M \times M \times M$ buněk o hraně h.

Indexace: Každý bod se přiřadí do buňky na základě celočíselného dělení svých souřadnic velikostí buňky:

$$j_x = \lfloor \frac{x_i - x_{min}}{h} \rfloor$$

- Vyhledávání: Pro nalezení sousedů bodu stačí prohledat pouze body uvnitř stejné buňky a v 26 přilehlých buňkách (Mooreovo okolí).
- Složitost: Za předpokladu rovnoměrného rozložení bodů klesá složitost vyhledávání na $O(1)$ pro jeden dotaz, celkově tedy $O(N)$.
- Nevýhoda (ANN): Pokud je poloměr hledání větší než velikost buňky, nebo leží-li nejbližší bod těsně za hranicí sousedních buněk (v případě fixního prohledávání okolí $3 \times 3 \times 3$), nemusí algoritmus najít skutečně nejbližšího souseda. Jedná se tedy o metodu přibližného vyhledávání (Approximate Nearest Neighbor - ANN).

3.3 KD-Tree (k-Dimensional Tree)

KD-strom je binární stromová struktura specializovaná na organizaci bodů v k-rozměrném prostoru.

- Konstrukce: Strom se staví rekurzivně. V každém uzlu se vybere jedna osa dělení (např. střídavě $x \rightarrow y \rightarrow z$). Množina bodů se seřadí podle souřadnic v této ose a rozdělí se mediánem na dvě poloviny (levý a pravý podstrom).
- Vlastnosti: Díky použití mediánu je strom vždy vyvážený a jeho hloubka je úměrná $log_2 N$.
- Vyhledávání (Pruning): Při hledání nejbližšího souseda algoritmus prochází stromem k listu. Při návratu (backtracking) kontroluje, zda může být v druhé větvi stromu bod bližší než ten

aktuálně nalezený. Pokud je vzdálenost k dělící rovině větší než vzdálenost k aktuálně nejbližšímu sousedovi, celou větev přeskočí (pruning).

- Složitost: Průměrná složitost konstrukce je $O(N\log N)$, složitost vyhledávání jednoho souseda $O(\log N)$.

3.4 Geometrické charakteristiky

Pro analýzu tvaru mračna (rozlišení kmene, větví a listí) využíváme statistickou analýzu okolí bodu.

- Hustota (ρ): Vypočítána jako inverzní hodnota třetí mocniny průměrné vzdálenosti d_{aver} k k nejbližším sousedům. V hustých oblastech je d_{aver} malé, hustota vysoká.

  Vložte do rovnice: $\rho = 1/(d_{aver}^3 + \epsilon)$

- Křivost (κ) a PCA: Metoda hlavních komponent (PCA) nad souřadnicemi sousedů poskytuje tři vlastní čísla $\lambda_1 \leq \lambda_2 \leq \lambda_3$ kovarianční matice. Tato čísla popisují rozptyl bodů v ortogonálních směrech.

  λ_3: Směr hlavního rozptylu.

  λ_1: Směr nejmenšího rozptylu (normála k proložené rovině).

  Křivost odhadujeme jako podíl nejmenšího vlastního čísla k celkovému rozptylu:

$$\kappa = \frac{\lambda_1}{\lambda_1 + \lambda_2 + \lambda_3}$$

Pokud je $\kappa \approx 0$, body leží v rovině (hladký povrch). Pokud je $\kappa > 0$, body jsou rozptýlené (šum, listí).

4. Implementace

Řešení bylo realizováno v jazyce Python (verze 3.x) s využitím knihovny NumPy pro efektivní vektorové operace a lineární algebru. Vizualizace výsledků byla provedena pomocí knihovny Matplotlib. Níže jsou rozebrány klíčové části kódu pro jednotlivé metody.

4.1 Naivní metoda (Brute Force)

Tato metoda slouží jako referenční řešení pro ověření správnosti optimalizovaných algoritmů. Pro každý bod vypočítá vzdálenost ke všem ostatním bodům v mračnu a vybere K nejbližších.

```python
# Naive search
def knn_naive(i):
    d = []
    for j in range(len(points)):
        if i != j:
            d.append((dist(points[i], points[j]), j))
    d.sort()
    idx = [j for _, j in d[:K]]
    return idx
```

4.2 Voxel Grid (Regular Grid)

Pro implementaci prostorové mřížky jsme využili datovou strukturu defaultdict(list). To nám umožňuje vytvářet tzv. řídký grid (sparse grid), kde v paměti existují pouze ty buňky (voxely), které skutečně obsahují nějaké body.

Klíčem do hashovací tabulky je n-tice celých čísel, která vznikne diskretizací souřadnic bodu. Při hledání sousedů procházíme nejen aktuální voxel, ale i jeho 26 sousedů.

```python
# Voxel search
def build_voxels(h):
    vox = defaultdict(list)
    for i, p in enumerate(points):
        key = tuple((p / h).astype(int))
        vox[key].append(i)
    return vox

def knn_voxel(i, vox, h):
    p = points[i]
    key = tuple((p / h).astype(int))
    cand = []
    for dx in [-1,0,1]:
        for dy in [-1,0,1]:
            for dz in [-1,0,1]:
                k = (key[0]+dx, key[1]+dy, key[2]+dz)
                cand += vox.get(k, [])
    d = [(dist(p, points[j]), j) for j in cand if j!=i]
    d.sort()
    return [j for _, j in d[:K]]
```

4.3 KD-Tree (Hierarchické dělení prostoru)

Implementace KD-stromu využívá rekurzivní třídu SimpleKDTree. Strom dělí body vždy mediánem podél jedné osy, přičemž osy se cyklicky střídají (x, y, z). Metoda query pak rekurzivně prohledává strom a efektivně prořezává větve, které nemohou obsahovat bližší bod.

```
class KDNode:
    def __init__(self, idxs, axis):
        self.axis = axis
        self.idxs = idxs
        self.left = None
        self.right = None

class SimpleKDTree:
    def __init__(self, idxs, depth=0):
        if len(idxs) == 0:
            self.node = None
            return
        axis = depth % 3
        idxs = sorted(idxs, key=lambda i: points[i][axis])
        m = len(idxs) // 2
        self.node = KDNode([idxs[m]], axis)
        self.node.left = SimpleKDTree(idxs[:m], depth + 1)
        self.node.right = SimpleKDTree(idxs[m + 1:], depth + 1)

    def query(self, p, best):
        if self.node is None: return

        # Výpočet vzdálenosti v aktuálním uzlu
        for i in self.node.idxs:
            d = dist(p, points[i])
            best.append((d, i))

        # Určení směru prohledávání
        axis = self.node.axis
        ref_val = points[self.node.idxs[0]][axis]
        diff = p[axis] - ref_val

        # Rozepsaná podmínka pro lepší čitelnost
        if diff < 0:
            first = self.node.left
            second = self.node.right
        else:
            first = self.node.right
            second = self.node.left

        # Rekurze
        if first: first.query(p, best)
        if second: second.query(p, best)
```

4.4 Octree (Bonusová implementace)

Jako bonusovou úlohu jsme implementovali Octree, který dělí prostor na 8 pravidelných oktantů. Na rozdíl od KD-stromu, který dělí podle počtu bodů (medián), Octree dělí geometrický prostor. Dělení se zastaví, když počet bodů v uzlu klesne pod 20 nebo je dosaženo maximální hloubky 5.

```
# OCTree
class Octree:
    def __init__(self, idxs, center, size, depth=0):
        self.idxs = idxs
        self.children = []

        # Podmínka pro další dělení
        if len(idxs) > 20 and depth < 5:
            for dx in [-1, 1]:
                for dy in [-1, 1]:
                    for dz in [-1, 1]:
                        # Výpočet středu nového oktantu
                        offset = np.array([dx, dy, dz])
                        c = center + (size / 4) * offset

                        # Filtrace bodů patřících do oktantu
                        # (rozepsáno místo dlouhého list comprehension)
                        sub = []
                        for i in idxs:
                            diff = np.abs(points[i] - c)
                            if np.all(diff <= size / 2):
                                sub.append(i)

                        # Vytvoření potomka
                        child = Octree(sub, c, size / 2, depth + 1)
                        self.children.append(child)

    def query(self, p, out):
        out += self.idxs
        for ch in self.children:
            ch.query(p, out)
```

4.5 Výpočet geometrických charakteristik

Pro analýzu tvaru mračna jsme implementovali výpočet hustoty a křivosti. Křivost je počítána z vlastních čísel kovarianční matice okolí bodu.

```
# Def curvature
def curvature(neigh):
    C = np.cov(neigh.T)
    l = np.linalg.eigvalsh(C)
    return l[0] / np.sum(l)

# Def density
def density(avg_d):
    return 1.0 / (avg_d ** 3 + 1e-12)
```

5. Data

Pro experimenty byl použit soubor tree_18.txt.

- Typ dat: Mračno bodů z pozemního laserového skenování (TLS).
- Obsah: Souřadnice X, Y, Z bez dalších atributů (intenzita, barva).
- Objekt: Dataset zachycuje vzrostlý strom. Data obsahují jak geometricky jasně definované části (kmen - válcová plocha), tak chaotické oblasti (koruna - volumetrický šum). Tato variabilita je ideální pro testování robustnosti výpočtu křivosti.

6. Přehled výsledků

6.1 Benchmark výpočetního času

Provedli jsme měření času potřebného pro nalezení sousedů a výpočet charakteristik v závislosti na počtu bodů.

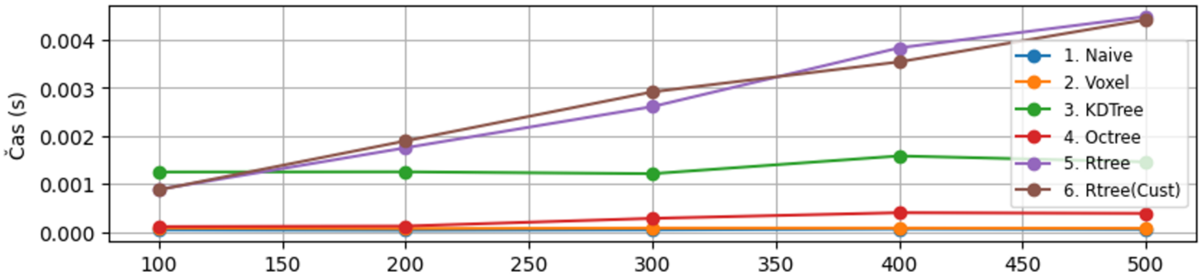
Naivní metoda: Vykazuje exponenciální nárůst času. Pro malé vzorky (do 1000 bodů) je použitelná, ale pro celý dataset je neefektivní.

KD-Tree: Vykazuje logaritmický růst času vyhledávání. I přes počáteční režii s konstrukcí stromu je pro velká data řádově rychlejší než naivní metoda.

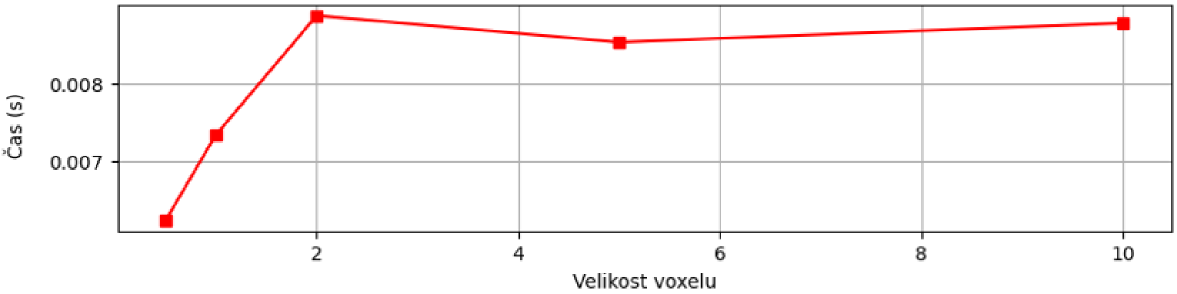
Voxel Grid: Rychlost je extrémně vysoká, ale silně závisí na parametru velikosti buňky h.

Pokud je h příliš malé, roste režie procházení prázdných buněk.

Pokud je h příliš velké, degeneruje metoda na naivní hledání uvnitř voxelu.



Graf 1: Srovnání výpočetního času. Naivní metoda (modrá) roste strmě, zatímco stromové struktury (zelená KD-Tree) drží čas nízko.



Graf 2: Analýza citlivosti Voxel Gridu. Existuje optimální velikost voxelu (kolem 2.0), kde je hledání nejrychlejší.

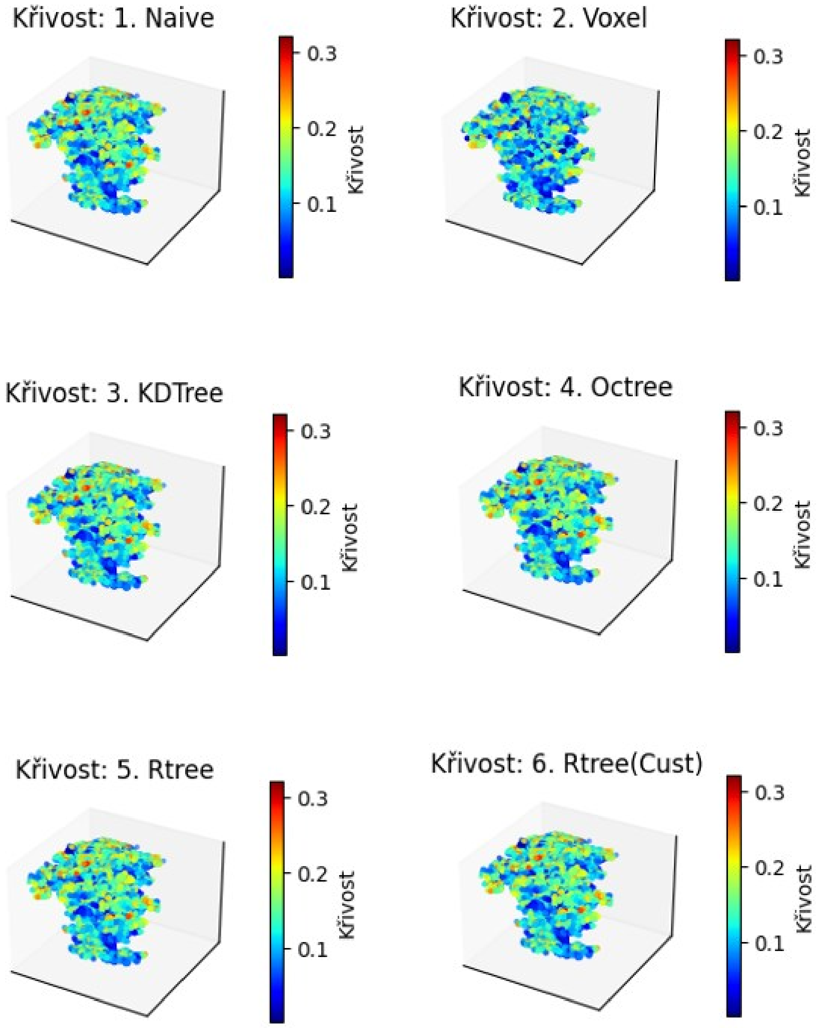
6.2 Přesnost (Hustota)

Porovnáním průměrné vypočtené hustoty jsme ověřili přesnost metod:

- KD-Tree / Naive: $\rho \approx 6060.37$. Metody jsou exaktní a vrací stejné sousedy.
- Voxel Grid: $\rho \approx 5507{,}54$. Nižší hodnota je způsobena tím, že Voxel Grid v naší implementaci (ANN) nenajde vzdálenější sousedy v řídkých oblastech, pokud leží mimo prohledávané $3 \times 3 \times 3$ okolí. To uměle zvyšuje průměrnou vzdálenost a snižuje hustotu.

6.3 Vizualizace křivosti

Výsledná vizualizace mračna bodů obarveného podle parametru \kappa prokazuje schopnost algoritmů



detekovat morfologii stromu.

Obrázek 1: Vizualizace křivosti. (Vlevo nahoře: Naive, Vpravo nahoře: Voxel, Vlevo dole: KD-Tree). Všimněte si shody v detekci kmene (modrá - nízká křivost) a koruny (žlutá/červená - vysoká křivost).

- Kmen (Modrá barva): \kappa \to 0. Body leží na povrchu válce, rozptyl v jednom směru (normála k povrchu) je minimální.
- Koruna (Červená barva): \kappa \to 0.33. Body jsou rozmístěny náhodně v prostoru (listí), rozptyl je ve všech směrech podobný.

7. Závěr

V rámci semestrální práce se podařilo úspěšně implementovat a otestovat tři různé přístupy k prostorové indexaci bez použití externích knihoven.

1. KD-Tree se ukázal jako nejuniverzálnější a nejspolehlivější řešení. Nabízí exaktní výsledky a stabilní výkon O(N \log N) nezávislý na parametrech, jako je velikost buňky.
2. Voxel Grid je implementačně nejjednodušší a nejrychlejší metodou, avšak za cenu ztráty přesnosti (ANN) a nutnosti experimentálně ladit velikost voxelu.
3. Naivní metoda posloužila jako nutná reference pro ověření správnosti, pro reálná data je však nepoužitelná.

Analýza křivosti potvrdila, že i s vlastní implementací základních algoritmů lze efektivně provádět pokročilou klasifikaci mračna bodů (segmentace kmen vs. koruna).

8. Seznam literatury

[1] BAYER, Tomáš. *Prostorová indexace*. Praha: ČVUT, Katedra geomatiky. Přednáškové slajdy.