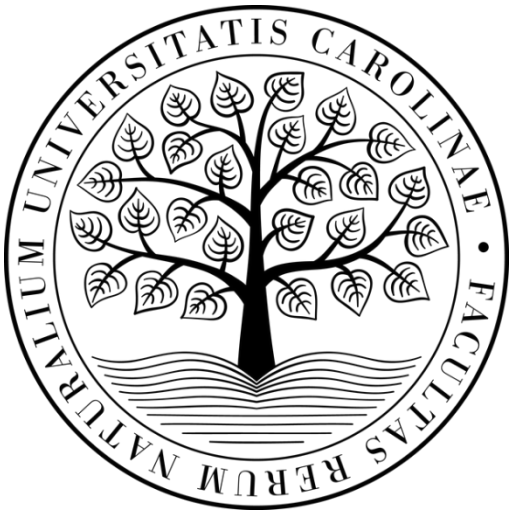
Univerzita Karlova

Přírodovědecká fakulta



Geoinformatika

Prostorová indexace

Matěj Hrabal, Kateřina Spudilová

1.N-GKDPZ

Praha 2025

ZADÁNÍ

Geoinformatika, prostorová indexace

Pro zadané bodové mračno tvořené body $p_i = [x_i, y_i, z_i]$ určete níže uvedené charakteristiky:

- prostorovou hustotu,
- křivost κ každém bodě.

Pro výpočet prostorové hustoty mračna využijte průměrnou vzdálenost d_{aver} k nejbližšímu bodu

$$\rho = \frac{1}{d_{aver}^3}.$$

Aproximovanou křivost určete metodou PCA v každém bodě mračna ze vztahu

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

počet *knn* volte 30. Prohledávání bodového mračna bude využívat následující metody:

1. naivní hledání,
2. akcelerované hledání s využitím voxelizace,
3. akcelerované hledání s využitím kd-tree.

Veškeré výše uvedené vyhledávací struktury implementujte samostatně (tj. bez použití knihoven). Určení hustoty bodového mračna realizujte metodami 1-3, výpočet křivosti metodami 2, 3. Hodnoty křivosti v každém z bodů mračna vizualizujte s využitím vhodné barevné škály. Vizualizujte výpočetní čas jako funkci velikosti vstupní množiny, u metody 2 také jako funkci velikosti voxelu.

Hodnocení:

Krok	**Hodnocení**
Prostorová indexace třemi metodami + výpočet charakteristik	20b
Akcelerované hledání s využitím Octree (vlastní implementace).	*+15b*
Akcelerované hledání s využitím Rtree.	*+5b*
Akcelerované hledání s využitím Rtree (vlastní implementace).	*+20b*
Max celkem:	**60b**

Zadané bodové mračno: tree_18.txt

1. Metoda naivního hledání

Algoritmus

Metoda naivního hledání (*The Brute Force Algorithm*) je nejzákladnějším přístupem k řešení problematiky hledání nejbližšího souseda (*Nearest Neighbor Search*) v bodovém mračnu. Nevyužívá žádné pomocné indexační struktury (např. grid, strom), vyhledávání tak probíhá přímo nad neseřazeným polem vstupních dat. Algoritmus tak postupně uvažuje každý jeden bod p_i a následně počítá euklidovskou vzdálenost mezi jím a všemi dalšími body p_j v bodovém mračnu, z těchto naměřených hodnot je následně vybrána ta nejnižší.

Hlavní nevýhodou metody naivního hledání je její vysoká výpočetní náročnost, jelikož samotný princip algoritmu je značně neefektivní. To se projevuje zejména při práci s vysokým množstvím bodů. Při výpočtu pro *n* bodů je nutné provést *n-1* dotazů, celkový počet se tak blíží n^2 – tato metoda tedy má kvadratickou časovou složitost ($O(n^2)$).

Následný výpočet prostorové hustoty ρ celého mračna pak vychází ze zadaného vzorce $\rho = \frac{1}{d_{aver}^3}$

Program

Program je psán v jazyce Python bez použití externích knihoven pro prostorovou indexaci, k vizualizaci výsledků slouží knihovna *matplotlib*. Analýza probíhá nad bodovým mračnem uloženým v souboru tree_18.txt, z něhož jsou nejprve pomocí funkce *loadPoints* načteny souřadnice jednotlivých bodů, jež jsou uloženy do polí X, Y, Z.

Následně je vytvořen prázdný seznam *nearest_list*, do nějž budou ukládány vzdálenosti k nejbližšímu sousedovi pro každý bod. Zároveň je spuštěno měření času pomocí *time.time()*. K procházení všech bodů bodového mračna slouží dva cykly *for*, přičemž vnější cyklus (*for i*) iteruje přes všechny referenční body a cyklus vnitřní (*for j*) přes všechny kandidátní body. Vnitřní cyklus pak obsahuje podmínku *if i == j: continue*, což zajišťuje, že nedochází k porovnání bodu se sebou samým.

Výpočet samotné vzdálenosti ve vnitřním cyklu je optimalizován tak, aby nedocházelo k opakovanému odmocňování výsledné hodnoty, což by významně prodlužovalo výpočetní čas. Vypočtena je tedy vždy druhá mocnina vzdálenosti *dist_sq* a k odmocnění dochází až po nalezení hledané nejnižší hodnoty (*min_dist_sq*, následně *nearest_dist*), která je posléze přidána do seznamu *nearest_list*.

```
for i in range(points_sum):
    min_dist_sq = float("inf")

    px = X[i]
    py = Y[i]
    pz = Z[i]

    for j in range(points_sum):
        if i == j:
            continue        # skip if i is the same as j

        dist_sq = (px - X[j])**2 + (py - Y[j])**2 + (pz - Z[j])**2      # compute distance squared
        if dist_sq < min_dist_sq:
            min_dist_sq = dist_sq       # new min distance squared

    nearest_dist = math.sqrt(min_dist_sq)

    nearest_list.append(nearest_dist)        # append nearest distance to the list
```

Za účelem výpočtu celkové hustoty mračna je pak proveden aritmetický průměr ze seznamu *nearest_list*, jehož hodnota je hodnotou d_{aver}. Výsledný čas a hustota jsou následně vypsány do konzole. Vizualizace výpočetního času jako funkce velikosti vstupní množiny byla provedena nad sadou testovacích velikostí, za účelem zkrácení výpočetního času. Grafem je parabola, která potvrzuje kvadratickou časovou složitost výpočtu.

Výsledky

Čas: 585.58 s
Hustota: 6060.374340762827

2. Akcelerované hledání s využitím voxelizace

Algoritmus

Metoda voxelizace (*grid-based indexing*), je výrazně efektivnějším způsobem prostorové indexace, který transformuje výpočetně náročný problém globálního vyhledávání sousedů na lokální úroveň. Princip spočívá v rozdělení prostoru mračna bodů do pravidelné trojrozměrné mřížky elementárních buněk – voxelů. Je tak nutné nejprve stanovit rozsah mračna a určit optimální velikost voxelu. Počet buněk na hranu mřížky n_r je odvozen z celkového počtu bodů N tak, aby byla zajištěna efektivní průměrná zaplněnost buněk. Každý bod p_i vstupního mračna je následně na základě svých normalizovaných souřadnic přiřazen do konkrétního voxelu, který je identifikován unikátním 1D indexem.

Samotné vyhledávání nejbližších sousedů se díky této struktuře značně zjednodušuje. Algoritmus neprochází celé mračno, ale omezuje se pouze na body, které se nacházejí ve stejném voxelu jako hledaný bod a v jeho okolí. Tím se radikálně snižuje počet nutných výpočtů vzdáleností, protože jsou ignorovány body ve vzdálených částech prostoru. Zatímco naivní metoda vykazuje kvadratickou složitost, časová složitost konstrukce mřížky i samotného vyhledávání se u voxelizace v ideálním případě blíží lineární závislosti.

Program

Programová implementace v jazyce Python využívá pro správu mřížky hashovací tabulku, realizovanou pomocí datové struktury *defaultdict* z modulu *collections*, která efektivně ukládá pouze neprázdné voxely. Program nejprve volá funkci *get_grid_params*, která vypočítá hranice mračna (*x_min, x_max*, ...) a na základě počtu bodů určí optimální počet buněk na hranu (*nr*). Následně v cyklu projde všechny body a pomocí funkcí *get_voxel_indices* a *get_hash* určí jejich příslušnost ke konkrétním voxelům. Funkce *get_voxel_indices* provádí normalizaci souřadnic bodu a vrací trojici celočíselných indexů j_x, j_y, j_z. Tyto 3D indexy jsou poté funkcí *get_hash* převedeny na unikátní jednorozměrný klíč (*h_idx*), pod kterým jsou indexy bodů uloženy do hashovací tabulky.

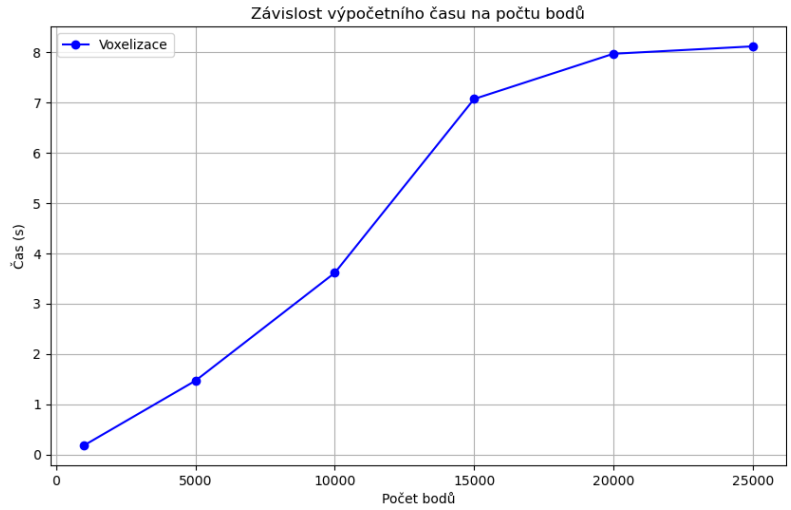
Při vyhledávání sousedů program v hlavním cyklu pro každý bod generuje klíče pro aktuální voxel a jeho 26 sousedních voxelů ve všech směrech. Pokud pro vygenerovaný klíč *neighbor_hash* existuje záznam v tabulce *H*, jsou body z tohoto voxelu přidány do seznamu *candidates* a je změřena jejich druhá mocnina vzdálenosti (*d_sq*). Tento seznam kandidátů je následně seřazen. Z něj je vybrán první nejbližší soused, jehož odmocněná vzdálenost je uložena do seznamu *nearest_list* pro finální výpočet prostorové hustoty. Prvních 30 nejbližších sousedů je uloženo do proměnné *k_subset* a využito pro výpočet křivosti.

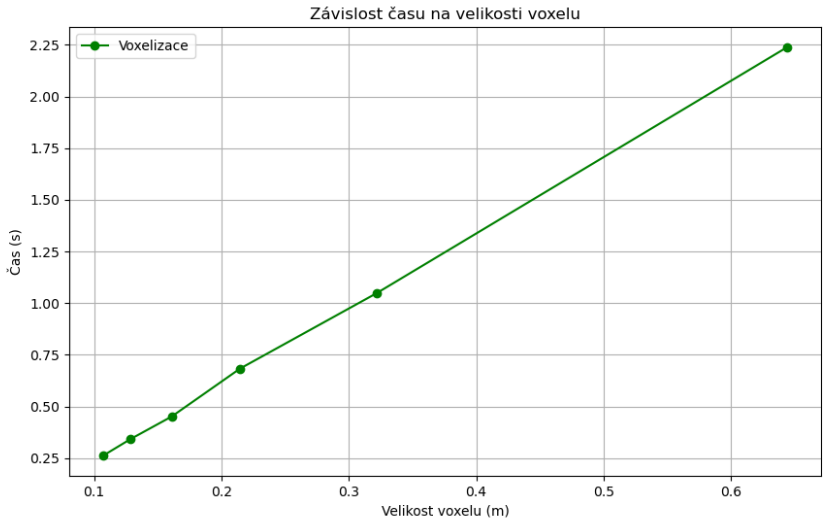
```
H = defaultdict(list)
for i in range(points_sum):
    jx, jy, jz = get_voxel_indices(X[i], Y[i], Z[i], x_min, x_max, y_min, y_max, z_min, z_max, nr)
    h_idx = get_hash(jx, jy, jz, nr)
    H[h_idx].append(i)

for i in range(points_sum):
    jx, jy, jz = get_voxel_indices(X[i], Y[i], Z[i], x_min, x_max, y_min, y_max, z_min, z_max, nr)

    candidates = []

    for kx in [-1, 0, 1]:
        for ky in [-1, 0, 1]:
            for kz in [-1, 0, 1]:
                nh = get_hash(jx+kx, jy+ky, jz+kz, nr)
                if nh in H:
                    for idx in H[nh]:
                        if i == idx: continue
                        d_sq = (X[i]-X[idx])**2 + (Y[i]-Y[idx])**2 + (Z[i]-Z[idx])**2
                        candidates.append((d_sq, idx))

    candidates.sort(key=lambda x: x[0])
```

Výpočet křivosti zajišťuje funkce *compute_curvature_pca*, která využívá knihovnu *numpy* pro operace lineární algebry. Z souřadnic 30 nejbližších sousedů je sestavena kovarianční matice, pro kterou funkce *np.linalg.eig* vypočítá vlastní čísla. Křivost je pak vypočtena jako podíl nejmenšího vlastního čísla ku součtu všech tří vlastních čísel. Součástí skriptu je vizualizace výpočetní náročnosti, která generuje dva grafy. První graf, vytvořený iterací přes seznam *test_nrs*, zobrazuje závislost času na velikosti voxelu pro nalezení optima. Druhý graf, iterující přes seznam *test_Ns*, zobrazuje závislost času na celkovém počtu bodů, přičemž výsledná křivka potvrzuje lineární průběh výpočetní náročnosti.

Výsledky

Čas: 13.8513 s
Hustota: 5751.849110935803
Křivost: 0.0822





3. Akcelerované hledání s využitím kd-tree

Algoritmus

Metoda kd-tree (*k-dimensional tree*) představuje přístup k prostorové indexaci, který pro organizaci bodů v prostoru využívá hierarchickou strukturu binárního vyhledávacího stromu. Na rozdíl od voxelizace, která dělí prostor na pravidelnou mřížku nezávisle na datech, se KD-strom dynamicky přizpůsobuje rozložení bodů v mračnu. Princip konstrukce stromu spočívá v rekurzivním dělení množiny bodů na dvě poloviny pomocí nadrovin kolmých k osám souřadnic. V každém kroku algoritmu se cyklicky vybírá jedna osa (X, Y, Z) a body jsou podle ní seřazeny. Prostřední prvek (medián) se stává uzlem stromu a zbývající body jsou

rozděleny do levého a pravého podstromu. Tento proces se opakuje, dokud nejsou všechny body zařazeny do struktury stromu.

Hlavní výhodou kd-tree je jeho vysoká efektivita při vyhledávání nejbližších sousedů. Algoritmus prochází strom od kořene k listům a postupně zužuje prohledávaný prostor. Klíčovým mechanismem pro zrychlení je tzv. *pruning*. Pokud algoritmus zjistí, že dělící rovina aktuálního uzlu je od hledaného bodu vzdálenější než dosud nalezený *k*-tý nejbližší soused, může s matematickou jistotou ignorovat celou druhou větev stromu, protože se v ní bližší bod nacházet nemůže. Díky tomu algoritmus nepočítá vzdálenosti ke všem bodům jako u naivní metody, ale pouze k malé podmnožině.

Program

Základem programu je třída *KDNode*, která reprezentuje jeden uzel stromu. Každý uzel uchovává souřadnice bodu, osu dělení a odkazy na levého a pravého potomka. Samotná stavba stromu je zajištěna rekurzivní funkcí *build_kdtree*, která v každém kroku seřadí předaný seznam bodů podle aktuální osy a rozdělí je mediánem.

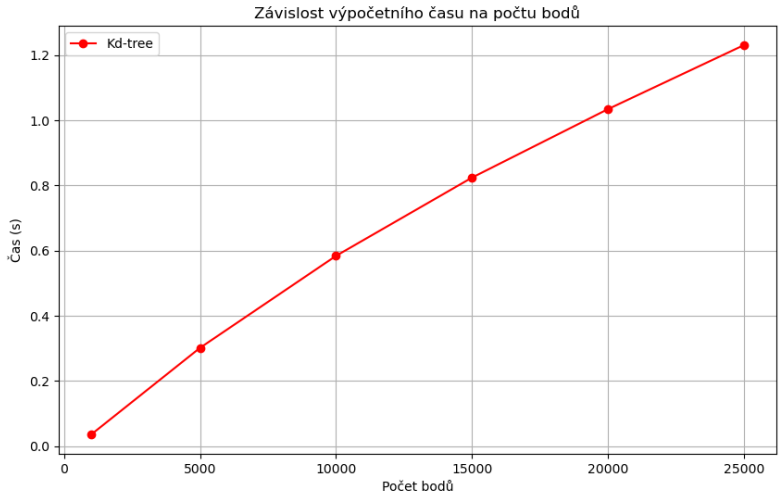
Pro vyhledávání *k* nejbližších sousedů slouží funkce *search_knn*. Ta prochází strom a udržuje si seznam *k* aktuálně nejlepších kandidátů. Funkce v každém kroku vyhodnocuje vzdálenost k dělící rovině (*diff*) a na základě porovnání s nejhorším kandidátem v seznamu rozhoduje, zda je nutné prohledávat i vzdálenější větev stromu.

```python
def search_knn(node, target, target_idx, k, heap):
    if node is None:
        return

    dist_sq = get_dist_sq(node.point, target)

    if node.index != target_idx:
        heap.append((dist_sq, node.point))
        heap.sort(key = lambda x: x[0])
        if len(heap) > k:
            heap.pop()

    axis = node.axis
    diff = target[axis] - node.point[axis]

    if diff < 0:
        near, far = node.left, node.right
    else:
        near, far = node.right, node.left

    search_knn(near, target, target_idx, k, heap)

    if len(heap) < k or (diff**2 < heap[-1][0]):
        search_knn(far, target, target_idx, k, heap)
```

Výsledkem vyhledávání je pro každý bod seznam sousedů, ze kterého je vybrán první nejbližší soused pro výpočet prostorové hustoty a skupina 30 nejbližších sousedů pro výpočet křivosti pomocí funkce *compute_curvature_pca*. Výsledky hustoty jsou shodné s výsledky naivní metody. Součástí skriptu je vizualizace výpočetního času v závislosti na počtu bodů.

Výsledky

Čas: 19.6479 s
Hustota: 6060.374340762827
Křivost: 0.0877



ZDROJE

https://dev.to/danudenny/optimizing-geometric-overlap-detection-a-deep-dive-into-spatial-indexing-with-python-2ndc

https://r-lidar.github.io/lidRbook/spatialindex.html

https://www.geeksforgeeks.org/dsa/understanding-efficient-spatial-indexing/