Semestrálna práca číslo 15
k-nearest neighbor search and range search with kd-trees

Dominika Kubániová[1]
Katedra počítačové grafiky a interakce,
Fakulta elektrotechnická, ČVUT Praha

Abstract

Implement k-nearest neighbour search based on kd-trees with sliding midpoint approach. Also implement the range search for rectangular and spherical queries. Test it on various distributions (uniform, Gaussian, scattered Gaussian, exponential etc.) of points in 2D, 3D, and 4D and compare it with the naive implementation. Use the data sets of various size among 10^3 to 10^8. Test the performance for 2 or 3 different implementation of priority queue.

Keywords: kd-tree, search, k-nearest, neighbour, sliding, midpoint, range, naive, algorithm

1. Úvod

Cieľom mojej semestrálnej práce bolo implementovať stavbu kd-stromu, vyhľadávanie pomocou kd-stromov a porovnať implementáciu na dátach s rôznymi distribúciami, dimenzionalitou, na meniacich sa parametroch stavby a vyhľadávania (k, polomer vyhľadávacej sféry, veľkosť vyhľadávajúceho kvádru, maximálny počet bodov uložených v jednom liste) a rôznych implementáciach prioritnej fronty. Súčasťou implementácie bolo generovanie dát s rôznymi distribúciami a dimenzionalitou, vizualizovanie priebehu vyhľadávania a grafové vizualizovanie komplexity algoritmu v závislosti na meniacich sa parametroch.

Ďalším cieľom bolo zistiť za akých okolností je vyhľadávanie pomocou kd-stromov rýchlejšie než naivné pre veľký počet dát s maximálnou dimenzionalitou rovnou 4.

2. Popis algoritmu

Semestrálnu prácu som implementovala v jazyku *C++*. Ako rôzne typy distribúcií som si zvolila uniformnú, normálovú, a exponenciálnu, ktorých implementáciu som použila zo štandardnej knihovny *C++*. Ďalej som algoritmus testovala na krivej normálovej distribúcii (skewed-normal), ktorej implementáciu používam z knihovny *Boost* a na koniec som implementovala generovanie dát distribuovaných do kružnice, resp. sféry. Priebeh vyhľadávania, vygenerované dáta a samotný kd-strom vizualizujem pomocou knihovny *VTK*. Z nameraných dát generujem štatistiky a grafy pomocou knihovny *Matplotlib* v jazyku *Python*.

Samotná stavba stromu a vyhľadávanie sú implementované v súbore kd_tree.cpp. Podotýkam že v tomto súbore sa všetky vyhľadávacie metódy vyskytujú dvakrát a to z toho dôvodu,

[1] A4M39DPG – Dominika Kubániová, zimný semester 2019/2020

že vždy jednu metódu som volala na meranie časovej zložitosti a druhú na vizualizovanie priebehu vyhľadávania. V týchto metódach sa robí to samé akurát si v nich ukladám a predávam viac dát než v nevizualizovacích.

Stavbu stromu som implementovala rekurzívne metódou *sliding-midpoint*, ktorá ako deliacu dimenziu v každom kroku *split_dim* vyberá tú dimenziu, v ktorej je obalovací kváder dát najširší. Prvotne ako deliacu hodnotu *split_value* v tejto dimenzii zvolí strednú hodnotu. Parametre *split_dim* a *split_value* spolu tvoria deliacu rovinu. Ak sa body v deliacej dimenzii nachádzajú na oboch stranách tejto deliacej roviny, algoritmus podobne ako v Quicksorte prechádza poľom bodov z oboch strán a prehadzuje body podľa pivotu (deliacej roviny), čím dostaneme dve množiny dát tých, ktoré sú menšie ako pivot a tých, ktoré sú väčšie ako pivot. V opačnom prípade sa deliaca rovina posunie v smere v ktorom sa body nachádzajú do bodu s najbližšou súradnicou. Z tohoto bodu vytvorí list a na druhej množine bodov d'alej aplikuje rekurzívny algoritmus. Pseudokódy stavania stromu a vyhľadávania som sa rozhodla neuvádzat' v reporte. Miesto toho uvediem pseudokód metódy starajúcej sa o posúvanie deliacej roviny.

```
1  // assume there will be happening sliding midpoint. If points on both sides of
       split_plane -> set this to false
2  sliding_midpoint = true
3  sliding_plane_from_right.value = MAX_FLOAT
4  sliding_plane_from_left.value  = MIN_FLOAT
5
6  point_b = points[first]
7  point_e = points[last]
8
9  // partition data
10 partition = true
11 while (partition){
12   while (point_b[split_dim] <= split_value){
13         if (sliding_midpoint) {
14           // iteratively find candidate for splitting plane from left direction
15             if (point_b[split_dim] > sliding_plane_from_left.value){
16               // set so far candidate with highest coords in split_dim
17                 sliding_plane_from_left.value = point_b[split_dim]
18             }
19         }
20         point_b = point_b.next
21     }
22
23     while (point_e[split_dim] > split_value){
24         if (sliding_midpoint) {
25           // iteratively find candidate for splitting plane from right
       direction
26             if (point_e[split_dim] < sliding_plane_from_right.value){
27               // set so far candidate with lowest coords in split_dim
28                 sliding_plane_from_right.value = point_e[split_dim]
29             }
30         }
31         point_e = point_e.prev
32     }
33     if (point_b >= point_e){  // index of point_b is bigger or equal than index
       of point_e in array of points
34         partition = false
35         splitting_plane = point_b
36
```

```
37          if (point_b == points[first]){
38            // all points are on right side of mid-plane -> slide to right
39            ...
40              splitting_plane = sliding_plane_from_right
41
42          }
43          if (point_e == points[last]){
44            // all points are on left side of mid-plane -> slide to left
45            ...
46              splitting_plane = sliding_plane_from_left
47          }
48          return splitting_plane
49      }
50      else {
51        // we found one point which is bigger than split_value and one which is
      less
52        // no sliding midpoint needed anymore. Just continue partitioning data
53          sliding_midpoint = false;
54          swap(point_b, point_e)
55      }
56  }
57
```

Následne sa upravia hraničné hodnoty obalovacieho boxu pre obe množiny a na obe množiny sa rekurzívne spustí rovnaký algoritmus. Rekurzia končí v momente keď máme rozdelit' menší počet bodov ako je stromom dovolené (definované v premennej max_points_in_leaf).

Stavbu stromu som implementovala aj nerekurzívne, avšak táto implementácia sa ukázala ako časovo náročnejšia. Dôvodom bolo väčšie množstvo kopírovania a vkladania dát na zásobník, čo som nedokázala obmedzit'. Nerekurzívnu implementáciu som v kóde neponechala.

Ďalej som implementovala kNN vyhľadávanie pomocou prioritnej fronty zo štandardnej *C++* knižnice a binomiálnej haldy. Klasickú binárnu haldu som nezvolila z dôvodu, že prioritná fronta z std knižnice je touto haldou implementovaná. Použila som implementáciu binomiálnej haldy, ktorú som našla na internete (Zdroj: Binomial heap implementation).

S nerekúrzívnou implementáciou kNN vyhľadávacieho algoritmu (a ostatných vyhľadávacích algoritmov) som sa inšpirovala pseudokódami z prednášky. Pri prehľadávaní v danom uzli sa pozriem na ktorej strane deliacej roviny sa query bod nachádza, napríklad vľavo, a podľa toho traverzujem ľavý podstrom. Zároveň ale ešte pred začatím traverzácie spočítam vzdialenost' do vzdialenejšieho uzlu a pridám uzol do prioritnej fronty. Na základe spočítanej vzdialensoti sa uzol vo fronte uloží na správne miesto pre neskoršie spracovanie. Do fronty vkladám uzly pomocou štruktúry kNN_search_str definovanej v súbore search_structures.cpp v ktorej mám uložený ukazateľ na daný uzol, vzdialenost' do tohoto uzlu a hraničné hodnoty obalovacieho boxu uzlu. Ďalej má táto štruktúra definované operátory nerovností pre zaradenie sa správne do fronty. Pridala som ešte jednu podmienku, ktorá zabezpečí skončenie vyhľadávania keď prioritná fronta je už naplnená k prvkami a zároveň najbližší uzol uložený vo vyhľadávajcej prioritnej fronte je vzdialenejší než aktuálny k-tý najbližší súsed query bodu. Algoritmus teda môže skončit' ešte pred tým než sa prioritná vyhľadávacia fronta vyprázdni.

Kvádrový vyhľadávací algoritmus sa mi ako jediný podaril naimplementovat' nerekurzívne bez kopírovania väčšieho množstva dát na zásobník. Algoritmus spočíva s testovaním v každom vnútornom uzle či sa vyhľadávací kváder celý nachádza v ľavom podstrome, alebo celý v pravom podstrome alebo v obidvoch. Na základe tejto informácie sa traverzujú d'alej oba podstromy alebo len jeden. V tomto vyhľadávaní som nepotrebovala d'alšiu pomocnú štruktúru

na zásobníku, stačilo naň vkladať iba ukazateľ na uzol. Pri spracovávaní listu testujem či sa súradnice query bodu nachádzajú v intervale daným rozsahom kvádru v konkrétnej dimenzii. Ak je táto podmienka pre všetky dimenzie splnená, bod sa pridá do výstupného poľa.

Pri sférickom vyhľadávaní som sa opäť inšpirovala implementáciou z prednášky, avšak nerekurzívne. Pri hľadaní sa vždy v uzli algoritmus pozrie na ktorej strane sa nachádza query bod a do tohoto smeru sa vydá, ale zároveň otestuje či je vhodné prehľadať aj druhý smer (ak vzdialenosť do tohoto smeru je menšia ako zadaný polomer). V kladnom prípade prehľadá aj druhý smer. Všimnime si, že na zásobník je potrebné ukladať uzly v opačnom poradí než by sa volala štandardne rekurzia. Algoritmus je podobný kNN vyhľadávaniu, avšak rozdiel je v tom, že v tomto prípade uzly ktoré sa od query bodu nachádzajú v zadanom polomere nikdy nemôžeme odrezať ako to bolo pri kNN vyhľadávaní kde sa nám polomer postupne znižoval.

Vo všetkých troch vyhľadávaniach ako výstupné pole používam pole pointrov na body, čim predchádzam duplikáciam dát v pamäti. Toto pole je definované v prípade sférického a kvádrového vyhľadávania ako std::deque pre ktoré je vkladanie prvkov časovo lacnejšie než vkladanie do std::vector, keďže neprebieha kopírovanie ak vektor je naplnený. Pri takomto vyhľadávaní nikdy presne nevieme koľko presne bodov na výstupe bude, preto pre mňa táto voľba bola časovo lepšia. Samozrejme dalo by sa odhadovať veľkosti výstupov z predošlých dotazov.

Štruktúra kd_node v sebe uchováva informáciu či sa jedná o list alebo vnútorný uzor (obe informácie v jednom integeri), v prípade vnútorného uzlu o akú deliacu dimenziu sa jedná a ktorou hodnotou má deliaca rovina prechádzať. Ďalej uchováva index do poľa uzlov predstavujúci pozíciu ľavého potomka v poli, pričom strom staviam tak aby pravý potomok sa nachádzal v poli na pozícií index + 1. Ďalej mám vytvorené pole štruktúr Leaf_points. Táto štruktúra obsahuje vektorový iterátor prvého bodu a iterátor posledného bodu. Ak daný uzol je list, potom index predstavuje pozíciu v poli štruktúr leaf_points. Všetky body medzi dvomi iterátormi uloženými v štruktúre patria k danému listu. Týmto spôsobom som sa dostala na veľkosť uzlu 12 bytov.

```
1 struct kd_node {
2     int index;  // index to left child in array of nodes (internal node)
3                 // right child at index
4                 // or index to array of data info (leaf node) storing iterator
    to first and last point of this leaf
5     int is_leaf_split_dim;
6     float split_value;
7 }
```

Pole uzlov a pole štruktúr leaf_points sa alokuje ešte pred stavbou stromu s maximálnymi veľkosťami aké môžu nabúdať. Vďaka tomu nie je potrebné v každom tvorení uzlu v rekurzívnej stavbe alokovať po jednom tieto štruktúry čo sa tiež ukázalo na časovej zložitosti.

V súbore search_comparator.cpp mám implementované metódy ktoré spúšťajú na povel jednotlivé vyhľadávacie kd a naivné algoritmy v závislosti na zadaných parametroch (využitie binomiálnej haldy miesto prioritnej fronty, mód vizualizovania atď.) a vypisujú na štandardný výstup informácie o časovej náročnosti a veľkosti, resp. vzdialenosti výstupov. Ďalej tento súbor obsahuje veľké množstvo experimentov, ktoré som nad mojou implementáciou testovala. Výstupy týchto experimentov meraných na počítačoch v učebni E:327 som ďalej spracovala pomocou python skriptov v priečinku python_vis_scripts. Výsledky týchto meraní budem prezentovať v sekcií meraní. Tieto meriace a spracovávajúce kódy nebudem viac vysvetlovať. Bola to skôr manuálna práca napísať kód ktorý moju implementáciu otestuje, než že by sa v

tom skrýval nejaký zaujímavý algoritmus.

Súbor visualizer.cpp obsahuje metódy ktoré sa starajú o vykreslenie dát pozbieraných pri stavbe a vyhľadávaní v strome. Vychádzala som z príkladov uvedených na stránke VTK documentation and examples. Nakoniec súbor data_generator.cpp obsahuje metódy pre vygenerovanie dát v rôznych distribúciách. Na všetky distribúcie okrem skew normálovej mi stačila štandardná knihovna. Skew normálové dáta generujem pomocou knihovny Boost.

3. Ťažkosti pri implementácií

Pri implementovaní som mala najväčí problém s napísaním algoritmov vyhľadávania nerekurzívne pomocou zásobníku tak, aby boli časovo a pamäťovo optimalizované. Pri kNN vyhľadávaní a sférickom vyhľadávaní som dlho času strávila implementovaním algoritmu tak aby nebolo potrebné na zásobník vlkadať celé pole s obalovacím boxom pri každom kroku priechodu uzlami v kNN vyhľadávaní a vkladania poľa offsetov do všetkých deliacích rovín pri sférickom vyhľadávaní. Nakoniec sa mi to nepodarilo. Snažila som sa aspoň obmedzit' počet alokovaní a dealokovaní premenných tým že som ich deklarovala vždy ešte pred prehľadávajúcou while-smyčkou. Pri stavaní stromu obalovací box predávam do funkcie referenciou a pred vyskočením z rekurzie ho vraciam do pôvodného stavu, čim sa kopírovanie nespôsobuje.

Ďalším nedostatkom mojej implementácie je pamäťová zložitosť, kvôli ktorej som nedokázala otestovať môj algoritmus na 10^8 dátach. Chyba asi bola v používaní dátovej štruktúry std::vector miesto napríklad alokovaného poľa na systémovej halde, a ďaľších pamäťovo náročnejších štruktúr.

Ďalej som strávila dlho času rozhodovaním sa na aké presne parametre sa pri meraní budem zameriavať. V mojej implementácií je veľké množstvo variabilných premenných pri meraní, ktoré všetky prispievajú ku zlepšeniu alebo zhoršeniu časovej náročnosti algoritmu, ako napríklad voľba rôzných distribúcií a rôzne parametre distribúcií (ako rozsah, stredná hodnota, smerodajná odchýľka ...), tri možné dimenzie, dve implemetácie prioritnej fronty, počet bodov, počet bodov uložených v liste, parametre vyhľadávania atď.

Ďalej bolo pre mňa ťažšie ujasnit' si ktoré variabilné premenné budem spolu porovnávať a na akej závislosti. Prvá vec čo napadla bolo merať čas stavby stromu a vyhľadávania v závislosti na zvyšujúcom sa počte dát. Nakoniec si ale myslím, že dominantnejším faktorom pri stavbe stromu je maximalný počet dovolených bodov v uzle (ďalej len MLP, maximum leaf points). Pred meraním som očakávala, že so zvyšujúcim sa MLP sa bude znižovať stavba stromu, ale zvyšovať časová náročnosť vyhľadávania, teda blížiť sa ku časovej náročnosti naivného algoritmu. Toto očakávanie sa na grafoch potvrdilo ako neskôr ukážem.

Problém so zdrojmi nebol žiaden. V podstate mi na naimplementovanie stačilo pochopiť algoritmus z článkov a prednášok. Časovo náročnejšie bolo pre mňa pripraviť štatistické dáta na vizualizáciu a samotné vizualizovanie, či už priebehu algoritmu alebo grafov, keďže som s tým nemala skúsenosti. Pri implementovaní vizualizácií som čerpala s príkladových kódov knihovny VTK, ktoré som upravila podľa svojích potrieb.

Celkovo odhadujem počet hodín strávených nad implementáciou, prezentáciami a reportom na približne 150 hodín.

4. Spustenie programu

Pre otestovanie implementácie som pripravila niekoľko vzorových vstupných súborov v priečinku inputs prezentujúce vyhľadávanie pre rôzne distribúcie a dimenzionality bodov.

Implementáciu po kompilácií spustením príkazu cmake CMakeLists.txt a príkazu make, by ste mali vediet' spustit' príkazom
./exe/project < inputs/<nazov suboru>, napríklad **./exe/project < inputs/input1_uniform.txt**

Vstupné súbory majú následujúci formát:
<pocet bodov> <dimenzionalita> <maximalny pocet bodov v liste> < 1 = s vizualizaciou, 0 = bez vizualizacie>
<nazov distribucie uniform, normal, skewnormal, exponential alebo circle> <parametre distrubucie>
<pocet M vyhladavacich dotazov>
// M riadkov obsahujucich:
<typ dotazu kNN, sph alebo rect> <parametre dotazu>

Detajlnejšie popísanie parametrov distribucie:
uniform <2 * pocet dimenzii parametrov predstavujucich rozsahy min a max v jednotlivych dimenziach>
normal <1 * pocet dimenzii parametrov predstavujucich strednu hodnotu a 1 * pocet dimenzii parametrov odchylky>
skewnormal <1 * poc_dim parametrov location, 1 * poc_dim parametrov scale a 1 * poc_dim parametrov alpha> //podl'a definície skew normal distribution na wikipedii
exponential <1 * poc_dim parametrov lambda>
circle <stred sféry = 1 * poc_dim parametrov a polomer>

Detajlnejšie popísanie parametrov dotazu:
kNN <parameter k, parameter pq ak vyhladavanie prioritnou frontou alebo bh ak haldou, nepovinny query bod teda 1 * poc_dim cisel>
sph <polomer, nepovinny query bod teda 1 * poc_dim cisel>
rect <1 * poc_dim parametrov predstavujucich rozsahy v jednotlivych dimenziach, nepovinny query bod teda 1 * poc_dim cisel>

Poznámka: Ak query bod nie je zadaný, program si vygeneruje náhodne ako query bod jeden z vygenerovaných bodov. Formátovanie je nutné dodržiavat' inak program nemusí fungovat'. Uznávam, že forma vstupov nie je optimálna, preto som ich pripravila čo najviac aby testovanie bolo jednoduchšie a maximálne obnášalo zmenu pár čisiel už v pripravených súboroch.

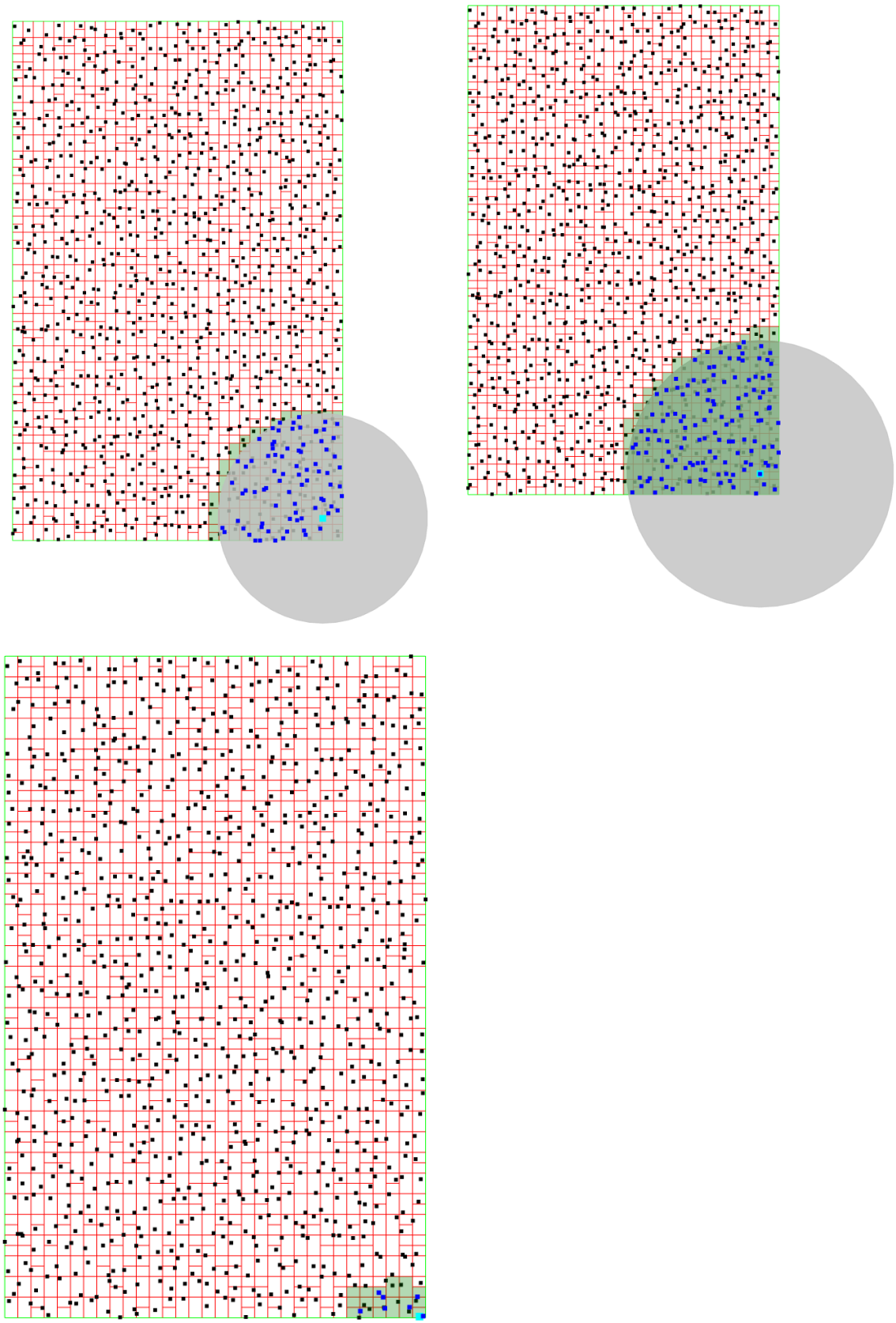
5. Vizualizácia vyhľadávania

V následujúcich vizualizáciách 1 až 7 bude zobrazený priebeh jednotlivých vyhľadávaní v rôznych distribúciach. Uvádzam vizualizácie len v prípade 2D s počtom bodov 10^5. Podotýkam, že vo vizualizáciách ale nezobrazujem všetky body ale iba podmnožinu z nich, kvôli ich veľkému počtu. Preto sa môže zdať že na niektorých stranách rozdelovacích rovín, resp. na čiarach predstavujúcich deliace roviny, sa žiadne body nenachádzajú. V skutočnosti to tak nie je.

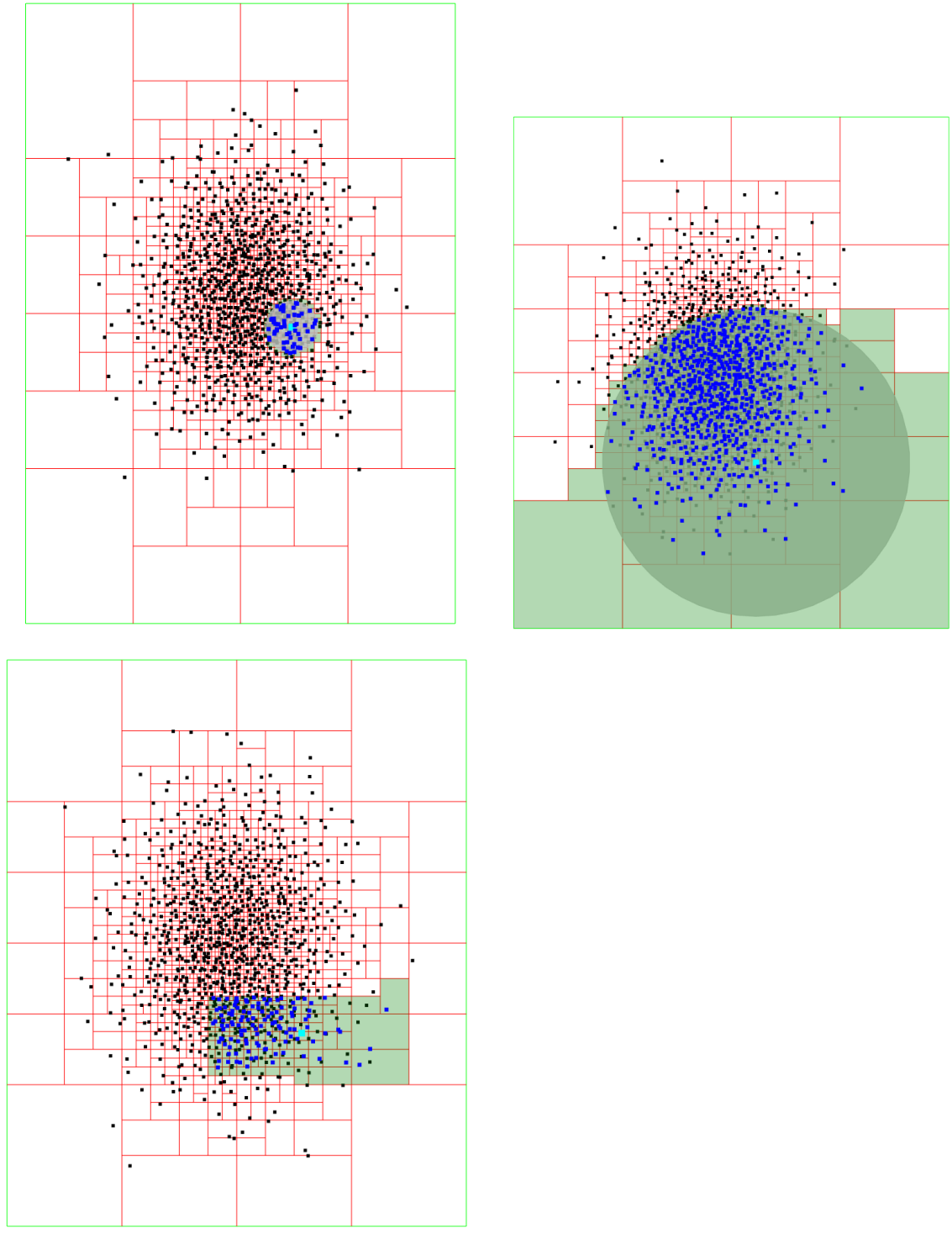
Parametre s ktorými sú vizualizácie vygenerované sa nachádzajú v rovnakom poradí ako idú obrázky za sebou vo vstupných súboroch:
input1_uniform.txt
input6_normal.txt
input14_skewnormal.txt
input9_exponential.txt
input12_circle.txt

Listy, ktoré sa pri vyhľadávaní navštívili a spracovali sú zobrazené zelenou farbou. V kNN prípade šedou farbou znázorňujem kruh o polomere vzdialenosti medzi query bodom a jeho k-tým najbližším súsedom. V tomto type vyhľadávania sa navštívené listy môžu nachádzať aj mimo tento polomer. Nie je to tak v sférickom vyhľadávaní kde navštívené listy by mali byť len tie, ktoré sa pretínajú s vyhľadávajúcou sférou/kruhom, zobrazeným šedou farbou. Môžeme na vizualizácií vidieť, že naozaj navštívené uzly sú len tie do ktorých šedý kruh zasahuje. To potvrdzuje správnosť mojej implementácie. Vo všetkých prípadoch body ktoré odpovedajú vstupnému dotazu sú vykreslené modrou. Kvôli veľkému počtu bodov sú ale aj niektoré body, ktoré by mali byť modrou farbou v dotazovom objekte vynechané a kvôli tomu miesto nich je vidno čierne body ktoré ležia presne pod nimi ale náhodou sa vykreslili. Nie je to chyba algoritmu, ale chyba mojej nesprávnej implementácie zobrazovania bodov.

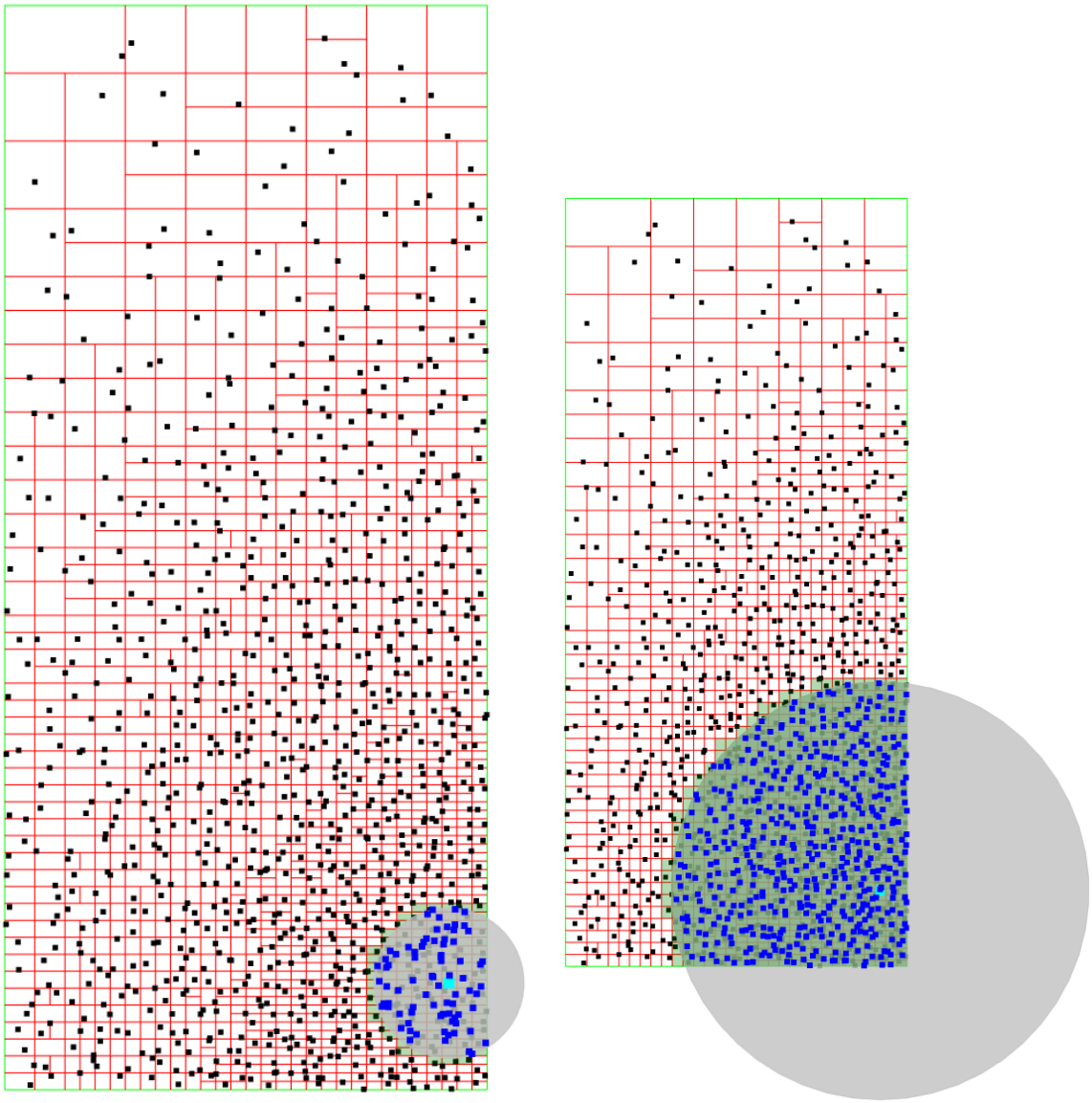
Čo sa týka kvádrového vyhľadávania, z neznámych dôvodov sa mi nedarilo zobraziť šedý dotazovací kváder ale len opäť zelenou navštívené listy pri vyhľadávaní. V 3D vizualizáciách (ktoré neuvádzam) som nezobrazovala navštívené listy z dôvodu že by bolo potrebné vykreslovať celé zelené kvádre, čo sa mi nepodarilo s mojími schopnosťami docieliť. Z rovnakého dôvodu v 3D kvádrovom vyhľadávaní nezobrazujem vyhľadávací kváder.



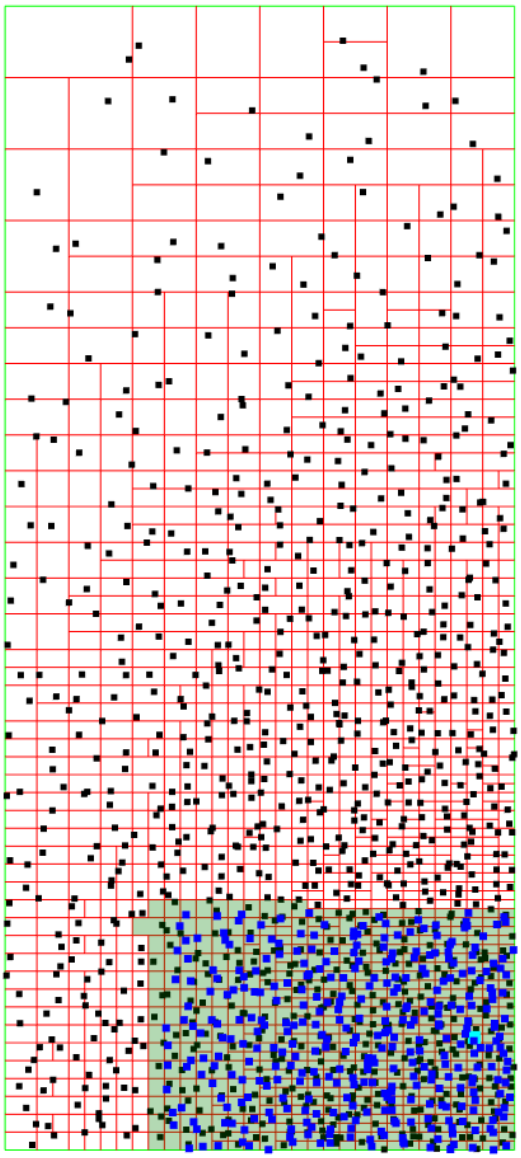
Obr. 1: Vizualizácia v poradí kNN, sférického a kvádrového vyhľadávania a navštívenosti listov pre uniformné rozdelenie v 2D prípade



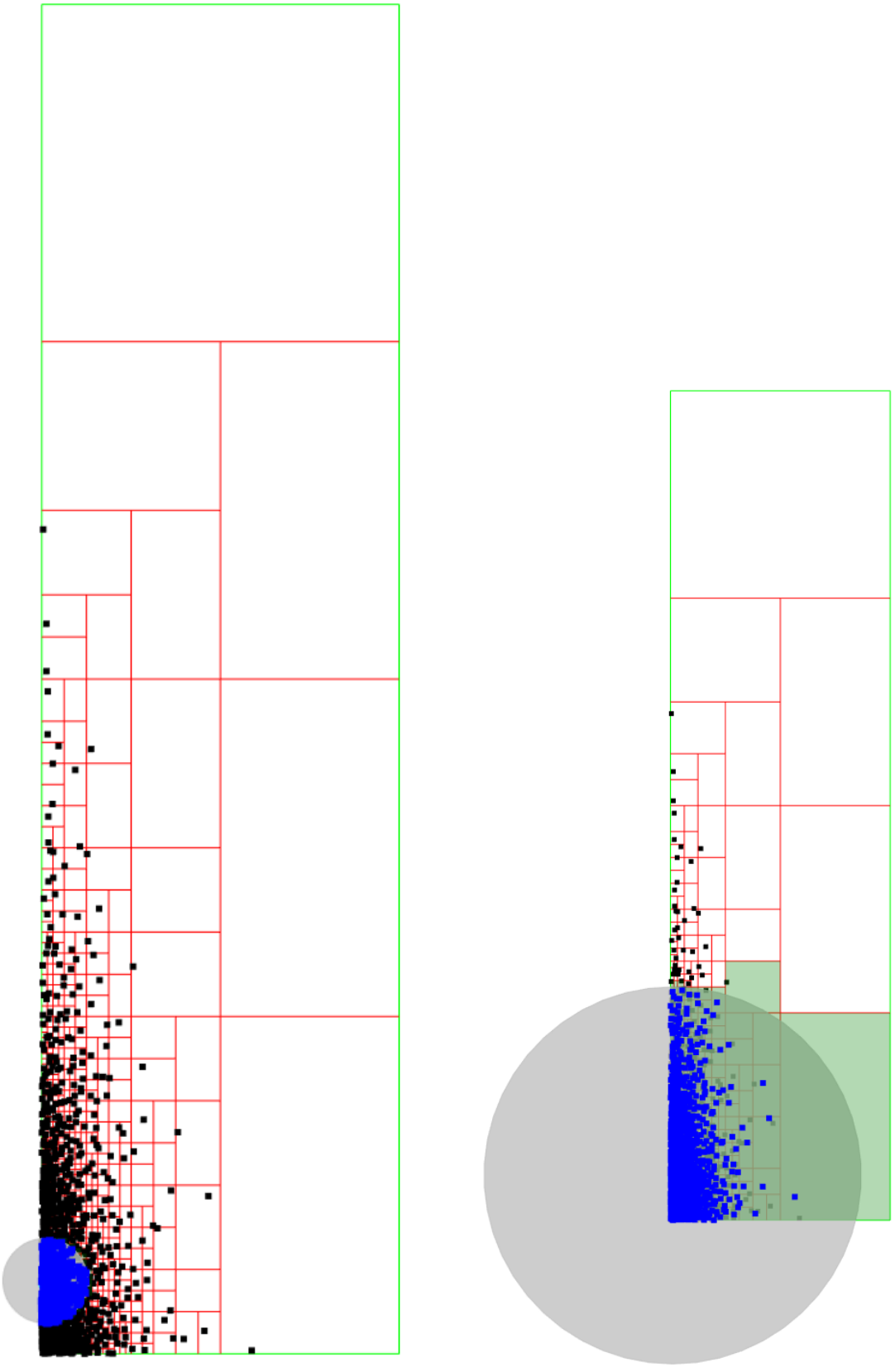
Obr. 2: Vizualizácia v poradí kNN, sférického a kvádrového vyhľadávania a navštívenosti listov pre normálové rozdelenie v 2D prípade



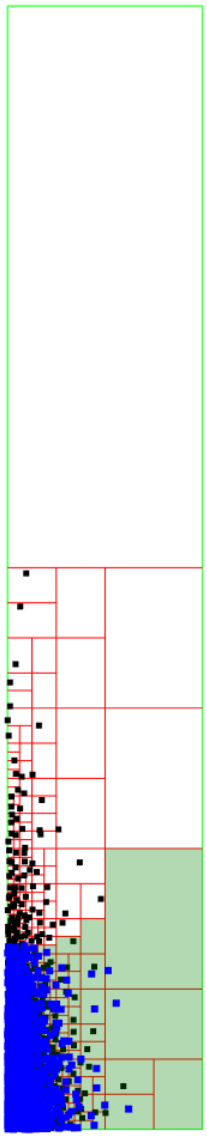
Obr. 3: Vizualizácia v poradí kNN, sférického vyhľadávania a navštívenosti listov pre skew-normálové rozdelenie v 2D prípade



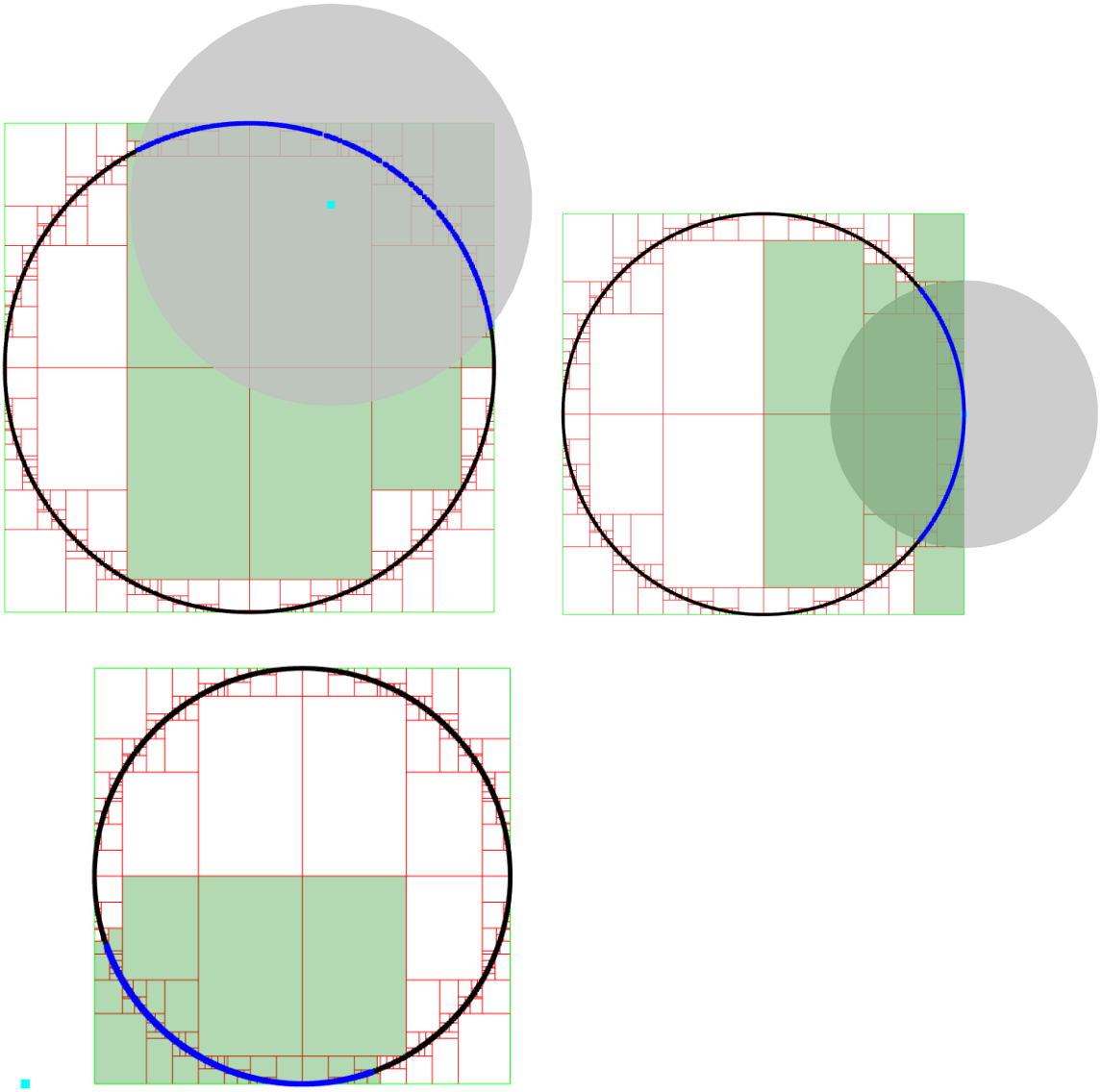
Obr. 4: Vizualizácia kvádrového vyhľadávania a navštívenosti listov pre skew-normálové rozdelenie v 2D prípade



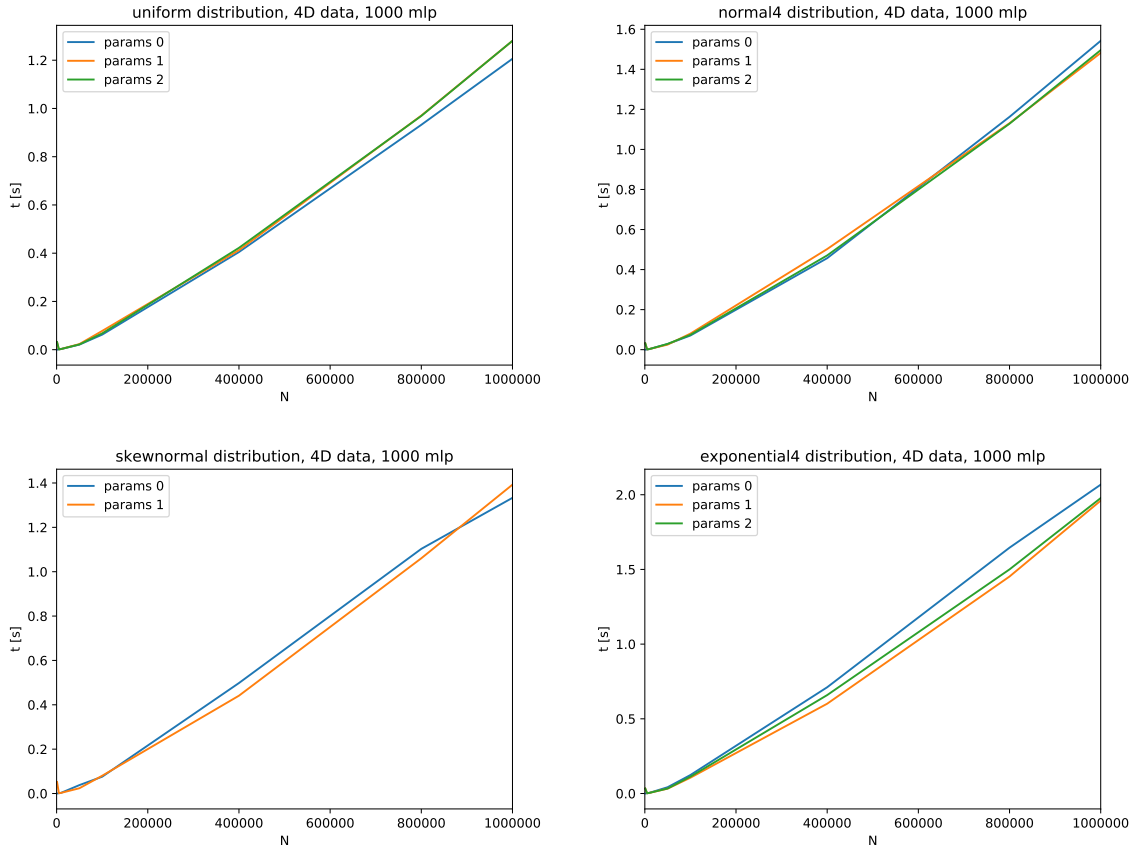
Obr. 5: Vizualizácia v poradí kNN, sférického vyhľadávania a navštívenosti listov pre exponenciálne rozdelenie v 2D prípade



Obr. 6: Vizualizácia kvádrového vyhľadávania a navštívenosti listov pre exponenciálne rozdelenie v 2D prípade



Obr. 7: Vizualizácia v poradí kNN, sférického a kvádrového vyhľadávania a navštívenosti listov pre sférické rozdelenie v 2D prípade



Obr. 8: Porovnanie času stavby stromu pre rôzne parametre distribúcií v závislosti so zvyšujúcim sa počtom 4-dimenzionálnych dát N

6. Namerané výsledky

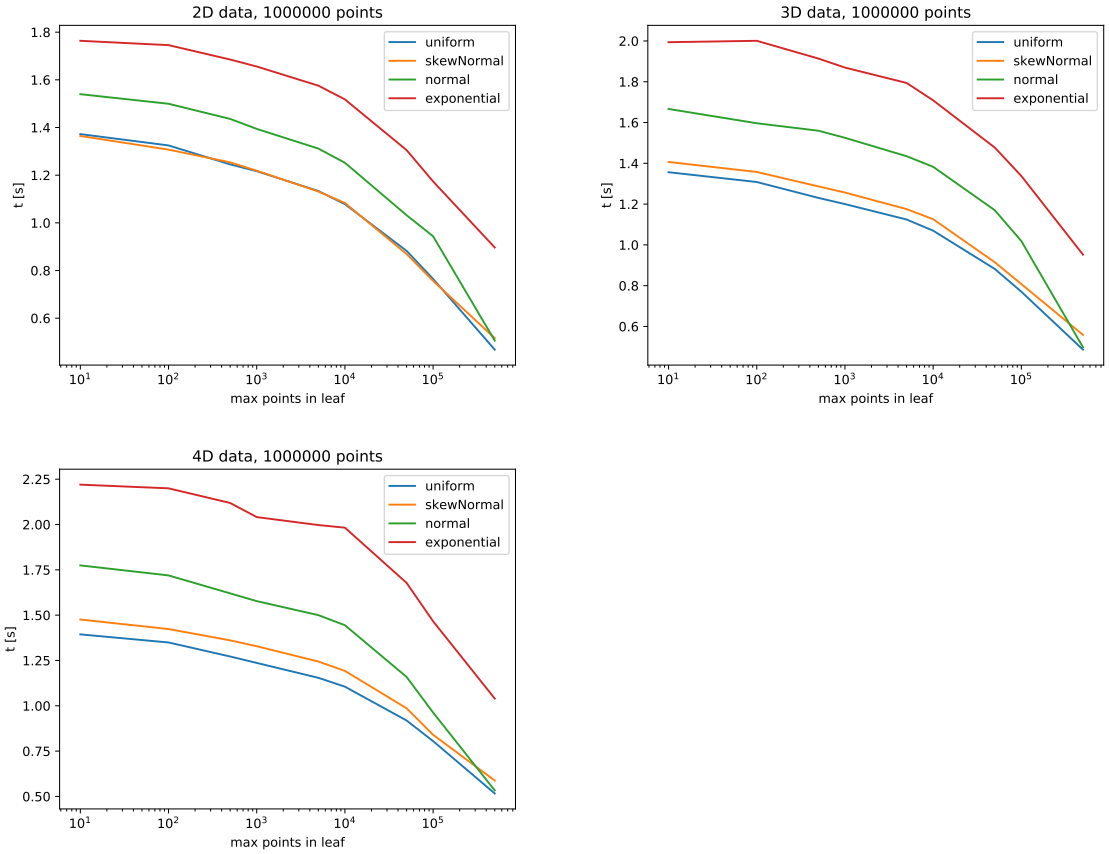
Poznámka: Pri meraní časovej náročnosti som vždy čas namerala 5x a zobrala strednú hodnotu.

Ako prvé ma zaujímalo či rôzne zvolené parametre distribúcií budú mať vplyv na časovú náročnosť stavby stromu. Po nameraní času v závislosti na zvyšujúcom sa N (počet bodov) a meniacich sa parametroch distribúcie som zistila, že moje parametre žiaden vplyv nemajú. Samozrejme by sa táto myšlienka dala testovať na väčšej množine parametrov s väčšou varianciou, ale tomuto testu som sa ďalej nevenovala. Na obrázku 8 môžeme vidieť, že pre jednotlivé distribúcie sa časova krivka pre rôzne parametre nemení. Môžeme si ale všimnúť že stavba stromu s exponenciálne distribuovanými dátami trvá dlhšie. Časovo najlepšie je na tom uniformné rozdelenie.

V následujúcich meraniach som pre každú distribúciu uvažovala vždy už len jeden typ parametrov. Konkrétne:

- Uniformná distribúcia (rozsah v jednotlivých dimenziách)
  $x : [15.0, 50.0]$, $y : [35.0, 90.0]$, $z : [-60.0, 40.0]$, $w : [-50.0, 36.0]$

- Normálová distribúcia $\mu = [0.0, 0.0, 0.0, 0.0]$, $\sigma^2 = [100.0, 30.0, 80.0, 50.0]$



Obr. 9: Porovnanie času stavby stromu pre rôzne distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste pre tri dimenzie

- Skew normálová distribúcia
  $\xi = [100.0, 250.0, 150.0, 300.0]$, $\omega = [0.5, 0.75, 0.25, 0.5]$, $\alpha = [20.0, -40.0, -10.0, 10.0]$
- Exponenciálna distribúcia $\lambda = [50.0, 10.5, 30.5, 20.0]$

Parametre skew normálovej a exponenciálnej distribúcie som naschál svolila tak aby body boli huste rozmiestnené v okolí ich počiatku. V meraniach sa pre takto zvolené distribúcie ukáže, že časová zložitosť vyhľadávania v kd-stromu sa bude podobať časovej zložitosti naivného algoritmu.

Merala som znižujúcu sa časovú náročnosť stavby stromu so zvyšujúcim sa MLP pre všetky tri dimenzie, a porovnávala na všetkých distribúciach. Počet bodov som fixne nastavila na 10^6. So zvyšujúcou sa dimenziou môžeme pozorovať na obrázkoch 9 nárast v čase, ale tvar časových kriviek sa výrazne nemení. Zároveň opäť môžeme vidieť, že dáta s exponenciálnou distribúciou najviac zvyšujú náročnosť stavby.

Následujúce meranie testuje časové náročnosti pre všetky tri typy vyhľadávania. Prvé meranie porovnáva kNN vyhľadávanie pre rôzne distribúcie s konkrétnymi hodnotami k, 10^6 bodami a konkrétnou dimenzionalitou, v závislosti na zvyšujúcom sa MLP. Meraný algoritmus používa ako prioritnú frontu zo štandardnej *C++* knižnice. Meranie uvádzam pre 2D a 4D dáta. V prvom stĺpci na grafoch 10-11 môžeme vidieť časovú zložitosť a v druhom stĺpči

znižujúci sa počet navštívených listov pri vyhľadávaní odpovedajúce ku danému meraniu v prvom stĺpci. Rovnaké meranie som uskotočnila s použitím binomiálnej haldy, viď 12-13.

Ďalej uvádzam meranie kNN vyhľadávania pre 10^7 4D dát pre uniformné a normálové rozdelenie s použitím prioritnej fronty, viď 14. Pre exponenciálne a skew normálové rozdelenie sa mi nepodarilo získať časy. Dôvodom môže byť to, že na tieto rozdelenia kd stromy nie sú úplne optimálne. Poznámka: Grafy reprezentujúce meranie na 10^7 dátach sú chybne označené - uvádzajú chybne 10^6 bodov.

Rovnaké merania uvádzam pre sférické vyhľadávanie a kvádrové vyhľadávanie, viď 15-17.

Z grafov 15-17 vidíme, že vyhľadávanie sa nechová optimálne pre exponenciálnu a skew normálovú distribúciu pre oba polomeri a obe dimenzionality. Dôvodom môže byť to, že v okolí query bodu s vyhľadávacím polomerom sa nachádza väčšina vygenerovaných bodov (čo pre exponenciálne a skew normálové podľa mnou zvolených parametrov distribúcií by mala byť pravda). Môžeme to pozorovať aj z počtu navštívených listov pri vyhľadávaní. Vidíme, že pre tieto dve distribúcie je počet navštívených listov výrazne väčší než pre uniformnú a normálovú. To by malo byť spôsobené práve tým, že v okolí query bodu je distribúcia bodov veľmi hustá a teda aj rozdelenie bodov rovinami do listov je veľmi husté.

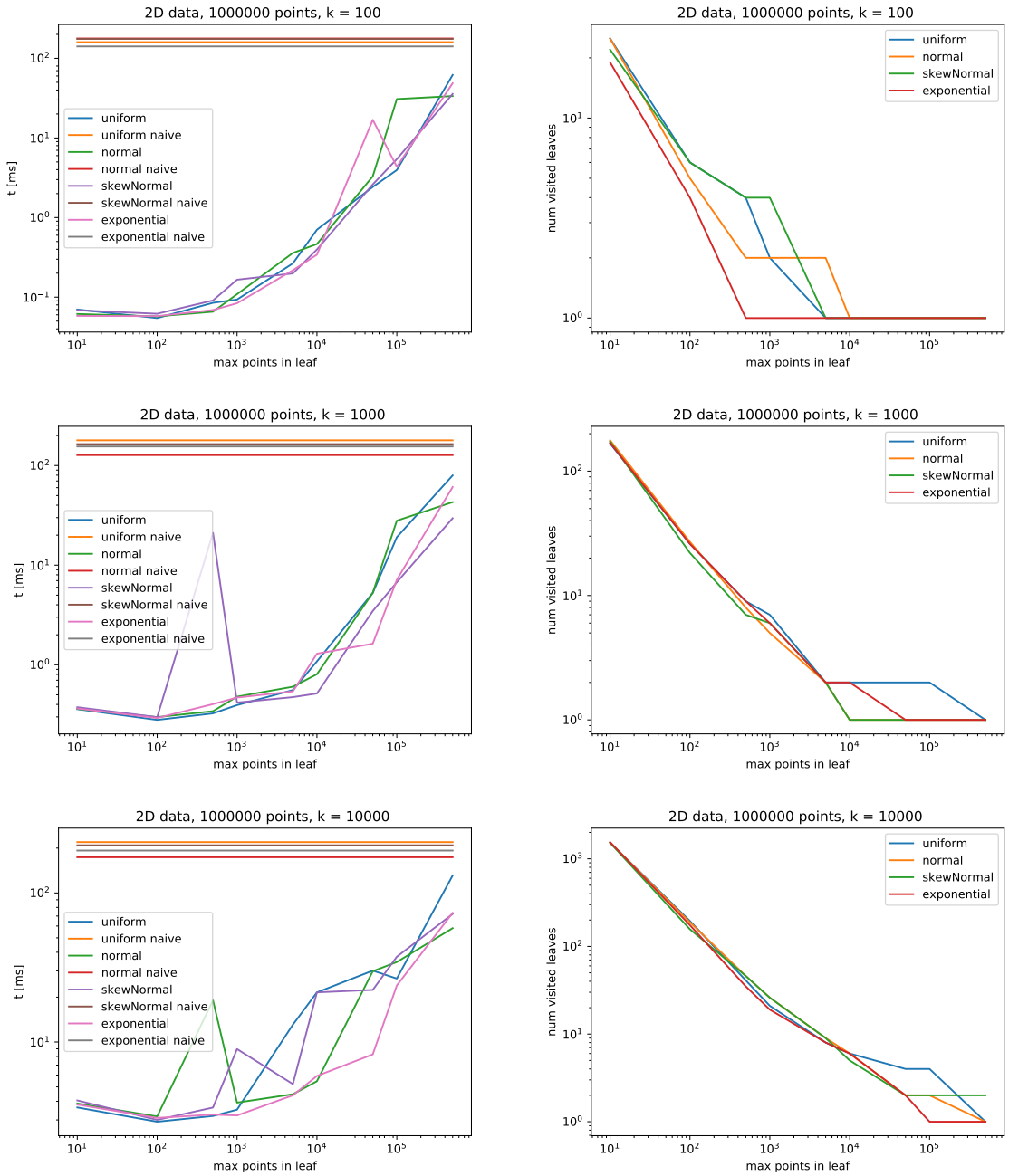
Dôvod prečo kd sférické vyhľadávanie pri uniformnej distribúcii pri polomere rovnom 50.0 sa časovo chová ako naivné vyhľadávanie môže byť spôsobené tým, že v niektorých dimenziách je veľká väčšina bodov priestoru obsažených v sfére o polomere 50.0 s centrom v query bode. Keď sa pozrieme na parametre uniformnej distribúcie a uvážime len prvé dve dimenzie, vidíme že rozdiel medzi maximálnymi a minimálnymi hodnotami je približne 50. To práve potvrdzuje graf s 2D bodami. kd vyhľadávanie pre uniformné rozdelenie je pre malý počet MLP dokonca horšie než naivné vyhľadávanie.

V prípade merania kvádrového vyhľadávania som zvolila dva typy kvádru. Jeden vyhľadávací kváder je viacrozmerná kocka o strane 50 (v grafe big) a druhý je viacrozmerný kváder ktorý má striedavo v dimenziách úzke a široké rozsahy (v grafe small). Konkrétne:

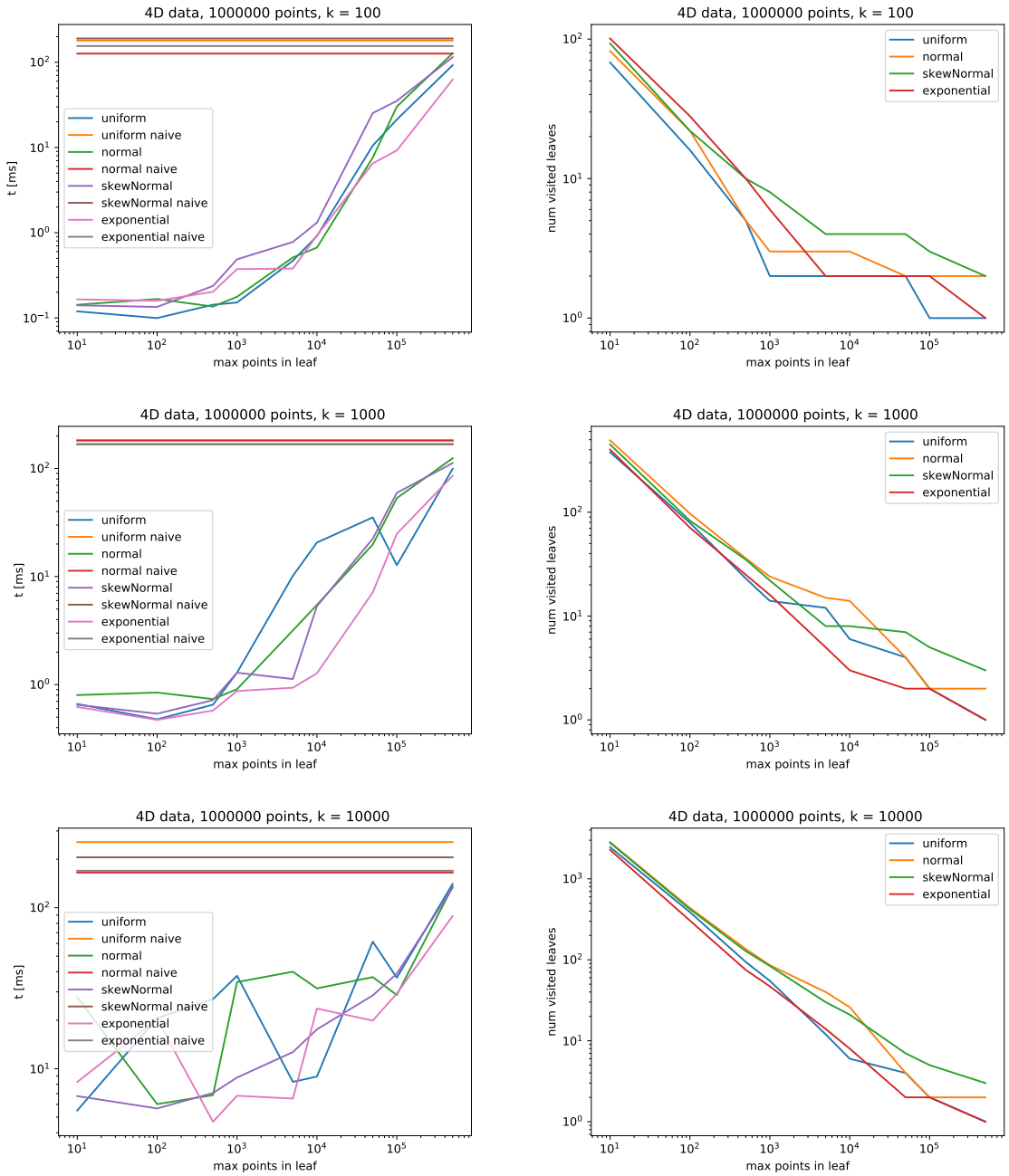
- big $x : [0.0, 50.0]$, $y : [0.0, 50.0]$, $z : [0.0, 50.0]$, $w : [0.0, 50.0]$
- small $x : [60.0, 70.0]$, $y : [25.0, 40.0]$, $z : [0.0, 100.0]$, $w : [-5.0, 30.0]$

Opäť na grafoch 18-20 môžeme vidieť, že sa kd vyhľadávanie v uniformnej distribúcii časovo blíži ku naivnému v prípade kocky z rovnakého dôvodu ako v predošlom odstavci so sférickým vyhľadávaním. Čo sa týka druhého vyhľadávacieho kvádru, kd vyhľadávanie pre uniformné aj normálové distribúcie sa chová lepšie než naivné keďže kváder nepokrýva väčšinu priestoru bodov.

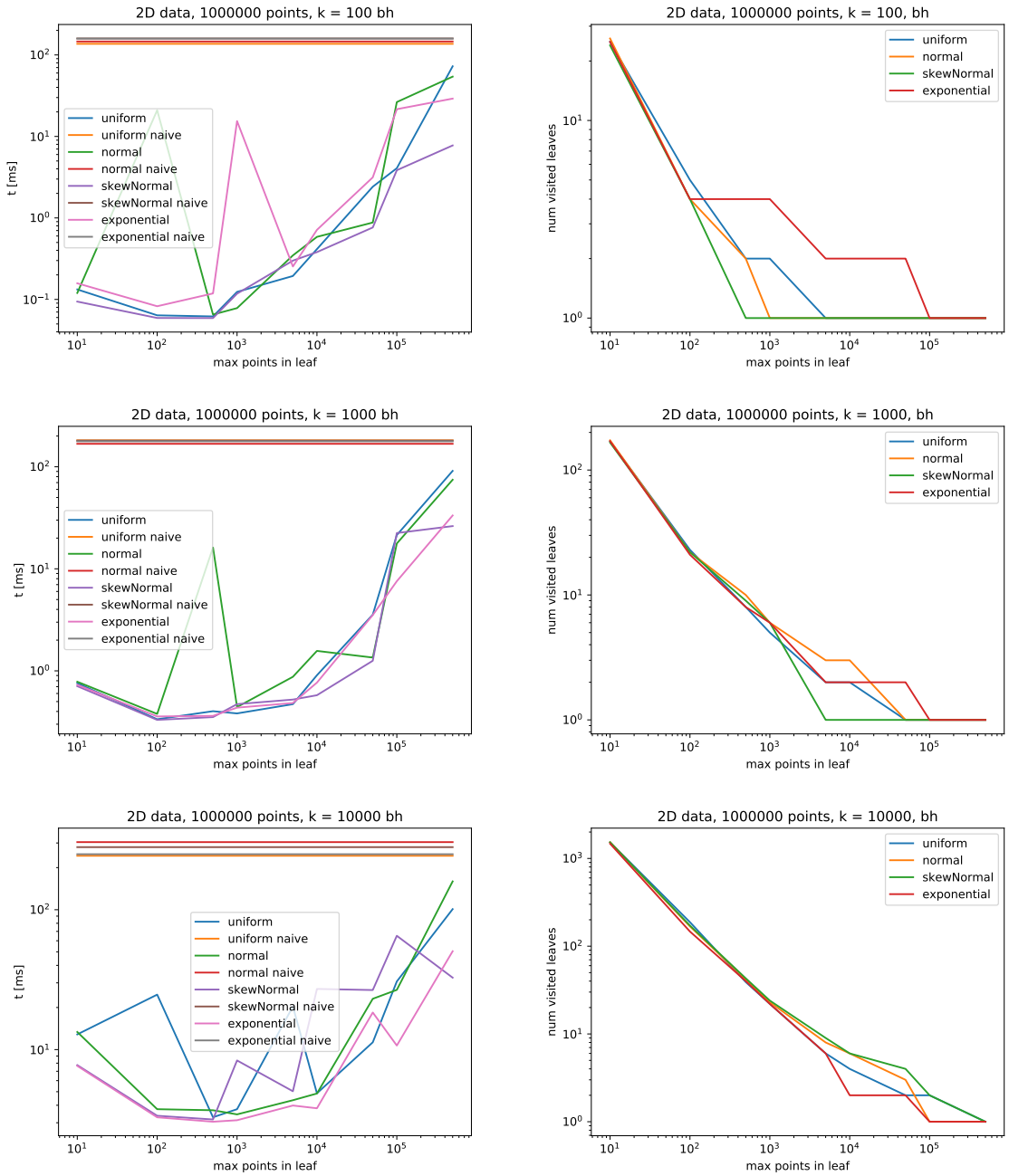
Merania som uskutočnila na počítači v učebni E:327. Konfigurácia počítača: Procesor: Intel Xeon E3-1231, 4 jadrá, 3.4 GHz, L1 cache: 256 kB L2 cache: 1 MB L3 cache: 8 MB Operačná pamäť: 16 GB DDR3 s taktom 1600 MHz Operačný systém: Windows 10 Verzia kompilátru: MSVC15



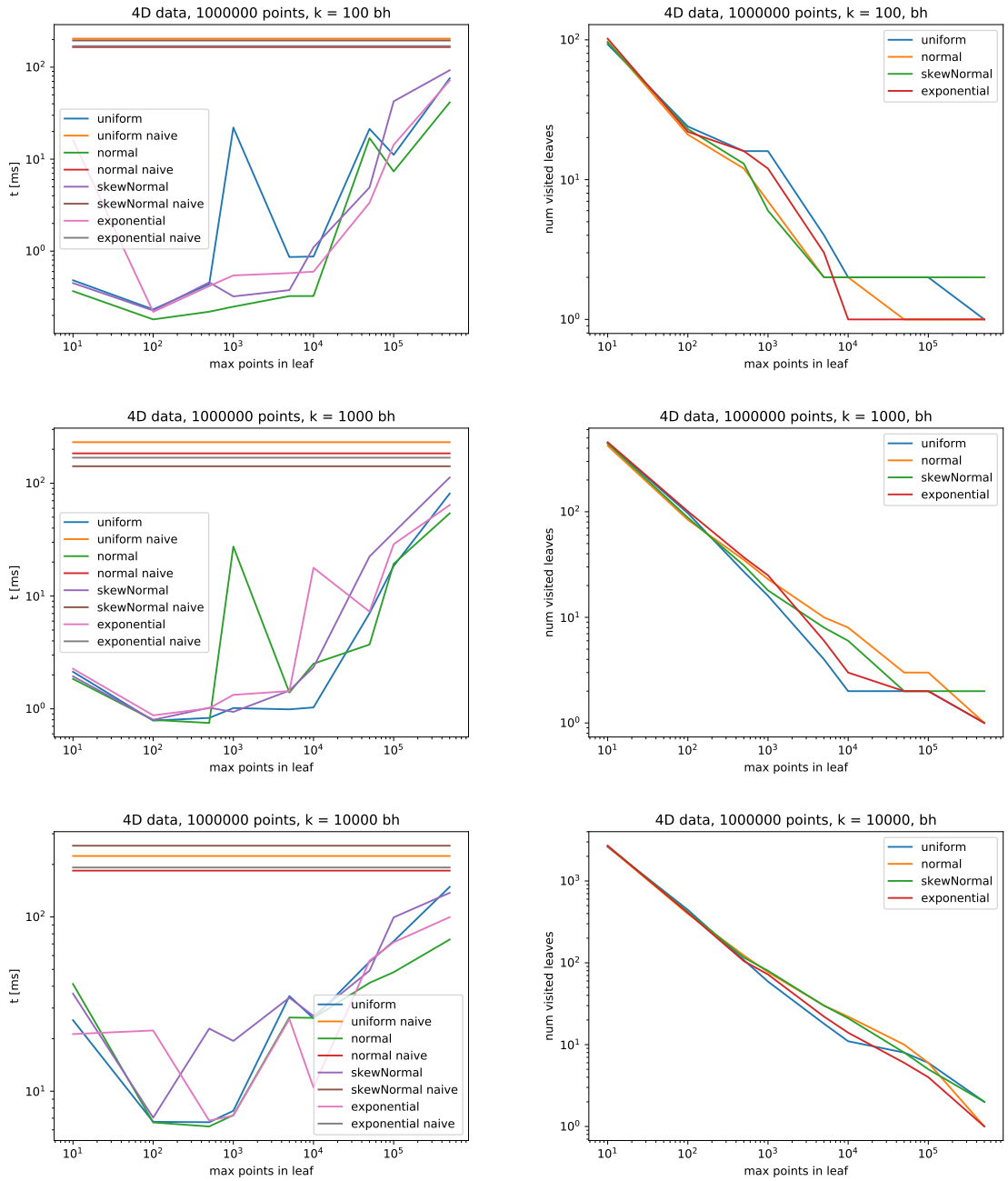
Obr. 10: Porovnanie času kNN vyhľadávania pre rôzne 2D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov. Implementované so std::priority_queue



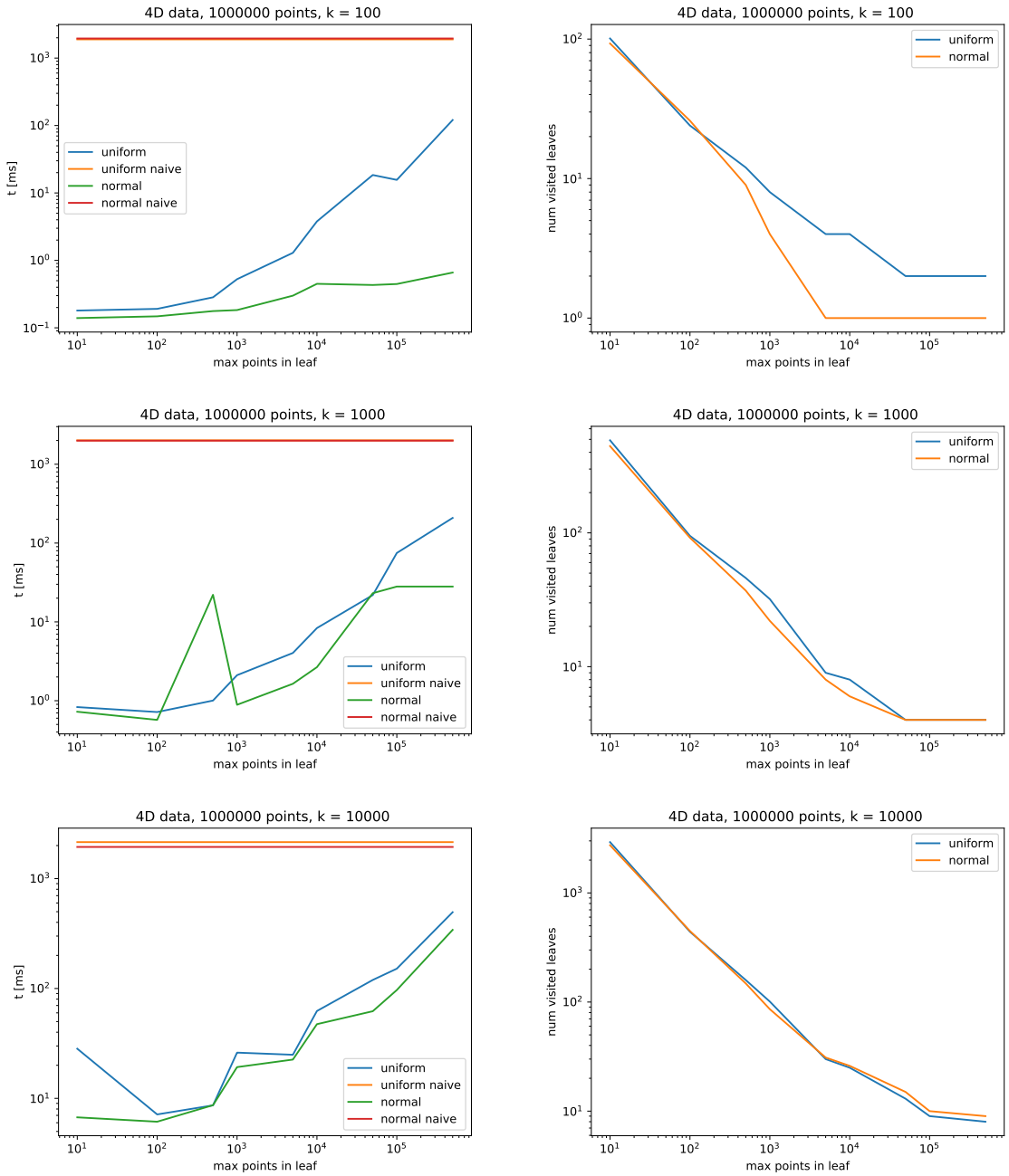
Obr. 11: Porovnanie času kNN vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov. Implementované so std::priority_queue



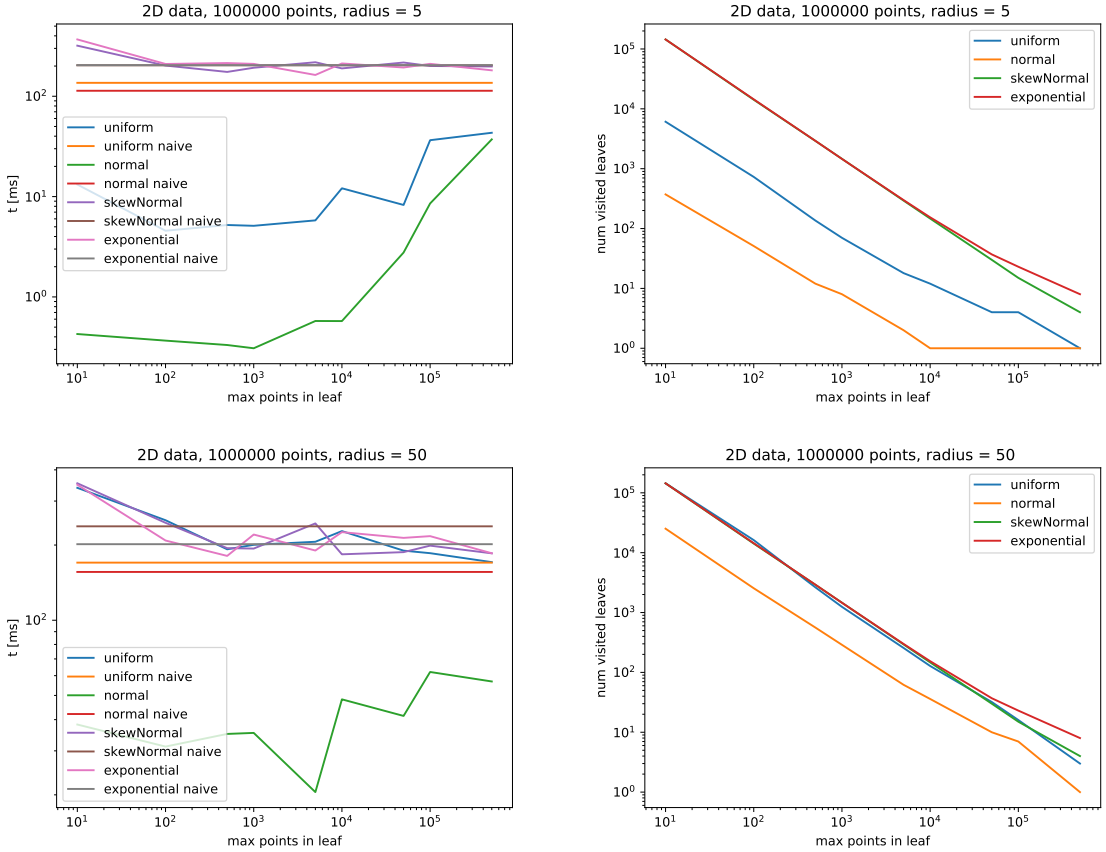
Obr. 12: Porovnanie času kNN vyhľadávania pre rôzne 2D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov. Implementované s binomialnou haldou



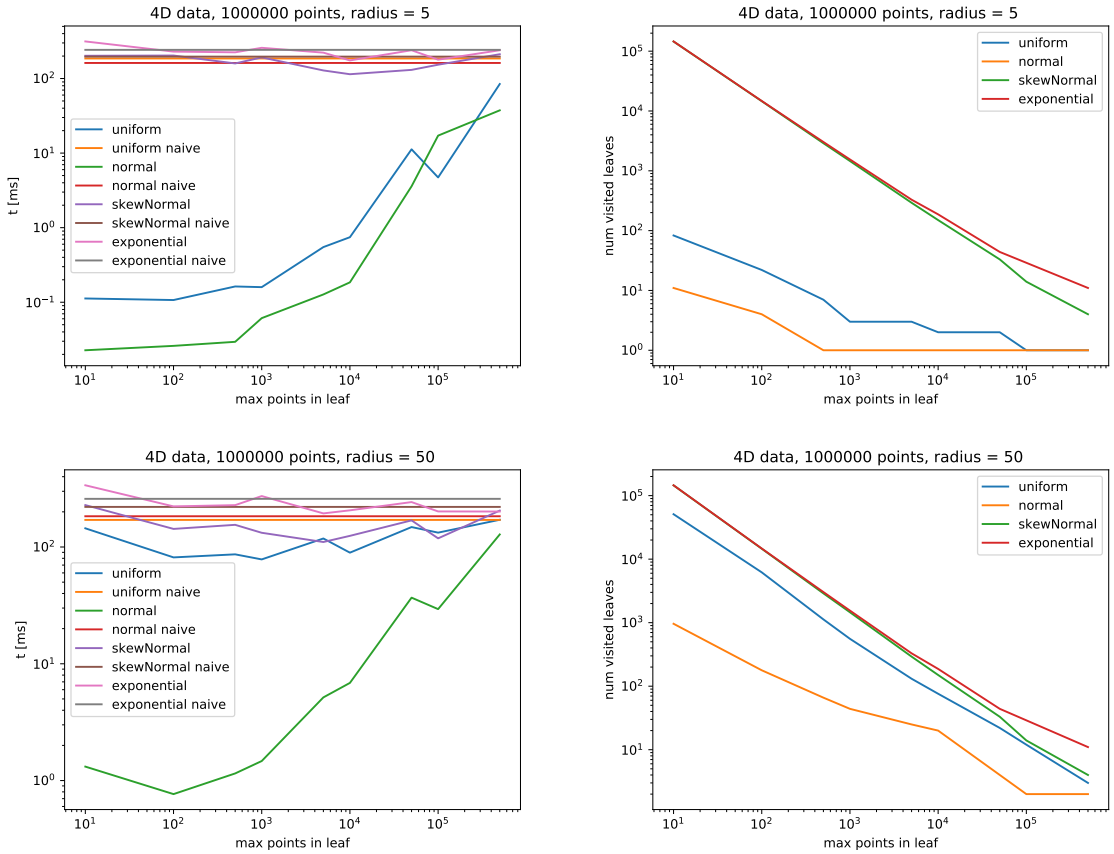
Obr. 13: Porovnanie času kNN vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov. Implementované s binomialnou haldou



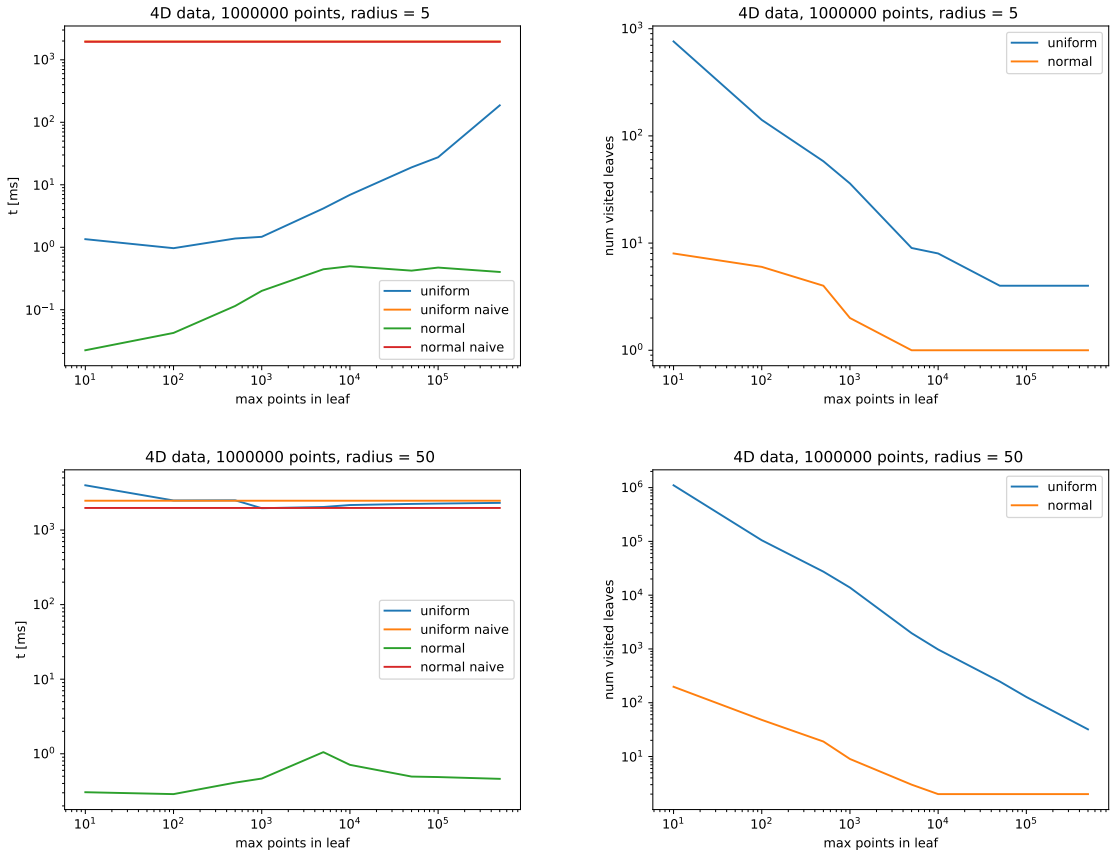
Obr. 14: Porovnanie času kNN vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^7 bodov. Implementované so std::priority_queue



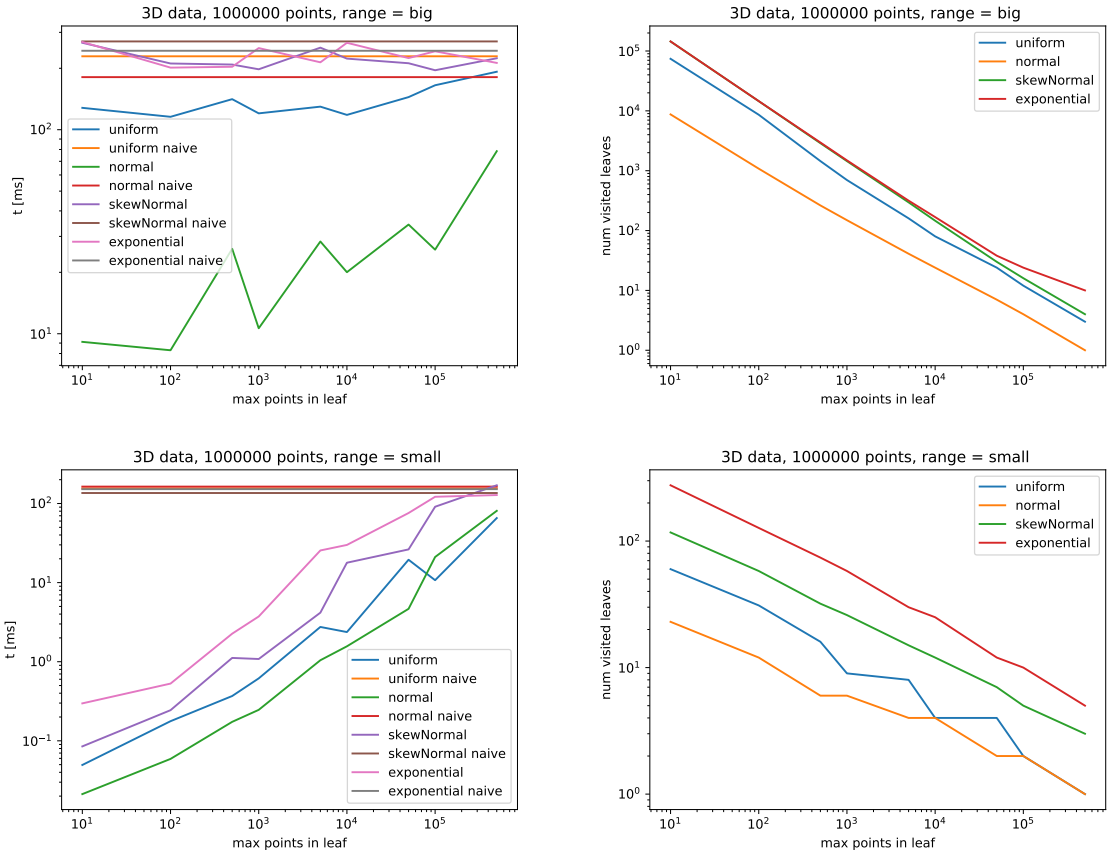
Obr. 15: Porovnanie času sférického vyhľadávania pre rôzne 2D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov



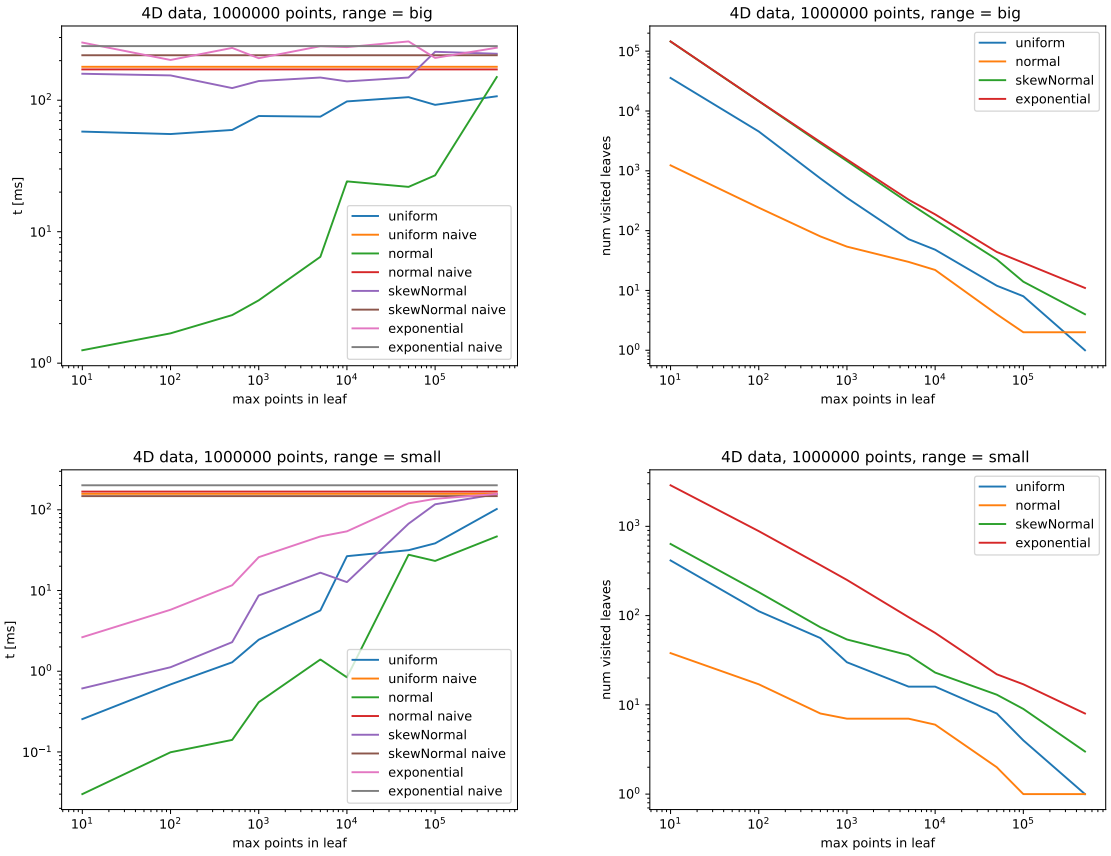
Obr. 16: Porovnanie času sférického vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov



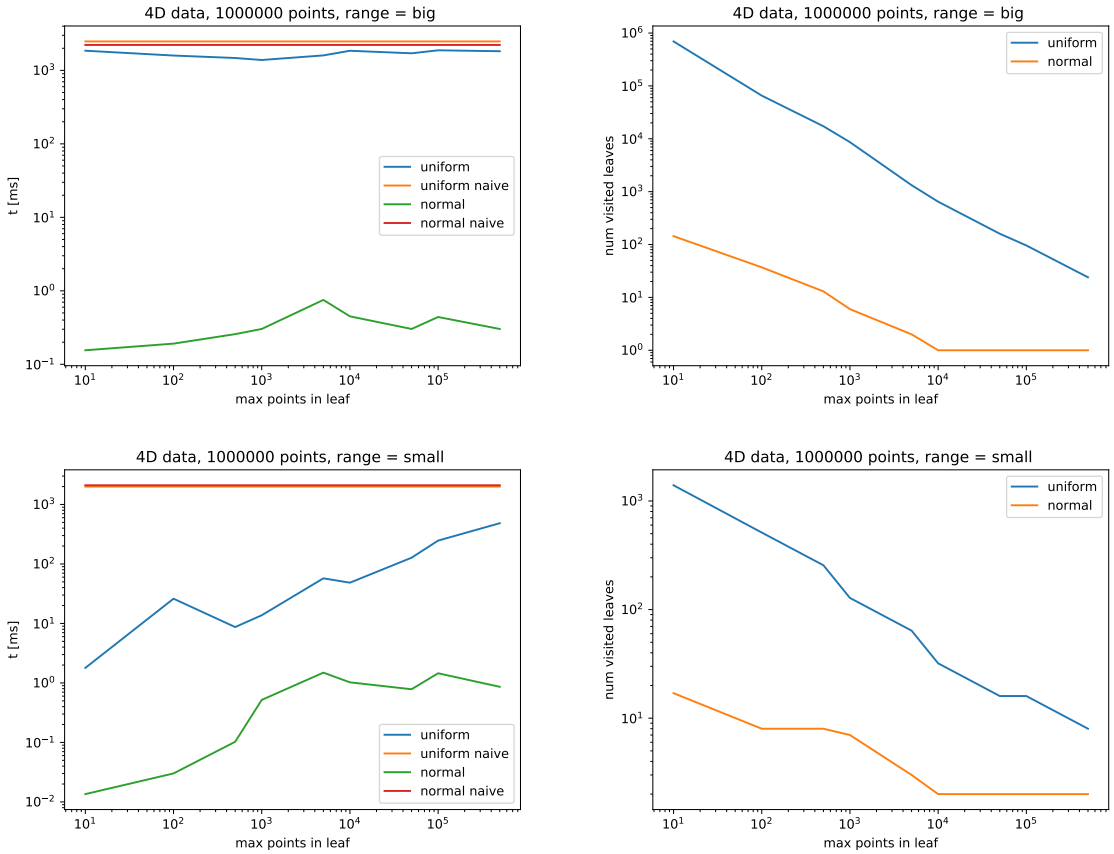
Obr. 17: Porovnanie času sférického vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^7 bodov



Obr. 18: Porovnanie času kvádrového vyhľadávania pre rôzne 3D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov



Obr. 19: Porovnanie času kvádrového vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^6 bodov



Obr. 20: Porovnanie času kvádrového vyhľadávania pre rôzne 4D distribúcie v závislosti so zvyšujúcim počtom maximálneho počtu bodov v jednom liste. 10^7 bodov

7. Záver

Môžeme povedat' že vyhľadávanie pomocou kd-stromov sa chová asymptoticky lepšie než naivný algoritmus. Vizualizáciami a grafmy som dokázala ale že to neplatí pre distribúcie so silne nerovnomerným zahustením priestoru a pre dotazy pri ktorých je výstupom väčšina bodov v priestore.

V tejto práci by sa dali ešte vylepšit' vizualizácie, hlavne chýbajúce zobrazovanie kvádrového vyhľadávania v 3D, tak ako aj zobrazovanie navštívených listov v 3D. Ďalej by šlo implementáciu zlepšit' z pamät'ovej stránky čo by mohlo docielit' schopnost' spracovávania 10^8 dát. Ako som už zmienila v sekcií tažkostí pri implementovaní, pri mojích nerekurzívnych algorimoch vyhľadávania dochádza ku kopírovaniu polí čo musí ovplyvňovat' časovú náročnost' (na druhú stranu je stále vyhľadávanie rýchlejšie než naivné). Tento problém by sa dal vyriešit' rekurzívnym vyhľadávaním. Môj zámer bolo ale vyhľadávanie naimplementovat' nerekurzívne, preto som pri tejto metóde zostala.

Literatúra

[1] Sample, Neal & Haines, Matthew Arnold, Mark & Purcell, Timothy. (2001). Optimizing Search Strategies in k-d Trees.

[2] Moore, Andrew. (2004). An Intoductory Tutorial on Kd-Trees.

[3] Maneewongvatana, Songrit & Mount, David. (2000). It's Okay to Be Skinny, If Your Friends Are Fat. Center for Geometric Computing 4th Annual Workshop on Computational Geometry.