

UNIVERZA V LJUBLJANI  
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Primož Kariž

**Iskanje podobnih primerov v  
večrazsežnih prostorih**

MAGISTRSKO DELO  
ŠTUDIJSKI PROGRAM DRUGE STOPNJE  
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: izr. prof. dr. Marko Robnik-Šikonja

Ljubljana, 2015



To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavnine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujojo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.



Izvorna koda magistrskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco GNU General Public License, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani <http://www.gnu.org/licenses/>.

*Besedilo je oblikovano z urejevalnikom besedil L<sup>A</sup>T<sub>E</sub>X.*



## IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani Primož Kariž, z vpisno številko **63070299**, sem avtor magistrskega dela z naslovom:

*Iskanje podobnih primerov v večrazsežnih prostorih*

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Robnika-Šikonje,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki ”Dela FRI”.

V Ljubljani, dne 24. marca 2015

Podpis avtorja:



*Zahvaljujem se mentorju izr. prof. dr. Marku Robniku-Šikonji za usmerjanje in pomoč pri izdelavi dela. Zahvalil bi se tudi bratu, dekletu in staršema za podporo skozi celoten študij.*



# Kazalo

<b>1 Uvod</b>	<b>1</b>
<b>2 Metode za iskanje bližnjih sosedov</b>	<b>3</b>
2.1 Eksaktne metode . . . . .	4
2.2 $\mathcal{E}$ -približne metode . . . . .	24
<b>3 Evalvacija</b>	<b>35</b>
3.1 Merjenje razpršenosti . . . . .	35
3.2 Opis podatkovnih množic . . . . .	39
3.3 Gostota podatkovnih množic . . . . .	41
3.4 Testni scenarij . . . . .	42
<b>4 Rezultati testov</b>	<b>45</b>
4.1 Analiza parametrov podatkovnih struktur . . . . .	45
4.2 Ocene porabe pomnilnika . . . . .	52
4.3 Hitrost iskanja najbližnjih sosedov . . . . .	54
<b>5 Izbira metode</b>	<b>67</b>
<b>6 Sklepne ugotovitve</b>	<b>71</b>
<b>Dodatek A</b>	<b>77</b>
<b>Dodatek B</b>	<b>95</b>
<b>Dodatek C</b>	<b>123</b>



# Seznam uporabljenih kratic

kratica	angleško	slovensko
<b>API</b>	application programming interface	programske vmesnike
<b>LSH</b>	locality-sensitive hashing	lokalno občutljivo zgoščevanje
<b>MBR</b>	minimum bounding rectangle	minimalen oklepajoč pravokotnik



# Povzetek

Iskanje najbližjih objektov se uporablja na različnih področjih in pomembno je, da jih lahko hitro poiščemo. Pri iskanju v visokodimenzionalnih prostorih ne znamo hitro poiskati eksaktnih sosedov, zato se zadovoljimo s približnimi. V magistrski nalogi opišemo najbolj uporabljane eksaktne in približne metode za iskanje najbližjih sosedov. Med eksaktnimi so to R,  $R^*$ , KD, M, PM in ball-drevo, med približnimi pa RKD-drevo, LSH, hierarhično razvrščanje z voditelji in gozd robov. Nekatere smo implementirali sami, druge smo uporabili iz že obstoječih knjižnic. Predstavimo in analiziramo rezultate testiranj hitrosti iskanja najbližjih sosedov, točnosti in porabe pomnilnika. V programskem jeziku python smo razvili knjižnico, ki vsebuje opisane metode in omogoča njihovo preprosto in enotno uporabo preko programskega vmesnika. Knjižnica omogoča tudi avtomatsko izbiro najprimernejšega algoritma za dano podatkovno množico. Algoritem izberemo na podlagi dveh odločitvenih dreves, ki smo ju sestavili s pomočjo analize rezultatov testiranj.

**Ključne besede:** algoritmi, podatkovne strukture, iskanje najbližjih sosedov, približni najbližji sosedji, visokodimenzionalni prostor, R-drevo,  $R^*$ -drevo, M-drevo, PM-drevo, ball-drevo, KD-drevo, RKD-drevo, LSH, hierarhično razvrščanje z voditelji



# Abstract

Nearest neighbours search is used in different problems, therefore it is important that we are able to find nearest neighbours fast. When searching in high-dimensional spaces we have to be satisfied with approximate nearest neighbours, because fast methods do not exist. In this master thesis we describe some well-known exact and approximate methods for searching nearest neighbours. The described exact ones are R, R\*, KD, M, PM and ball-tree, while the approximate are RKD-tree, LSH, hierarchical k-means and boundary-forest. Some of them we implemented, while others were taken from existing libraries. We present and analyze the search results in terms of speed, precision and memory requirements of methods. We developed a library in python programming language, which includes the described methods and provides a simple and consistent API. The library also allows automatic selection of the most suitable algorithm for a given dataset based on two decision trees, which were created through analysis of the results.

**Keywords:** algorithms, data structures, nearest neighbours search, approximate nearest neighbours, high-dimensional space, R-tree, R\*-tree, M-tree, PM-tree, ball-tree, KD-tree, RKD-tree, LSH, hierarchical k-means



# Poglavlje 1

## Uvod

Algoritmi za iskanje najbližjih sosedov se uporabljo za različne naloge, kot so iskanje najbližjega soseda v geometrijskem prostoru, iskanje podobnih dokumentov in napovedovanje pri strojnem učenju. Zaradi široke in pogoste uporabe je zaželeno, da se iskanje izvaja hitro. V tem delu obravnavamo algoritme za iskanje najbližjih sosedov, ki imajo manj kot linearno časovno zahtevnost, ki jo zahteva izčrpno iskanje, pri katerem preiščemo vso bazo podatkov. Na izbiro najprimernejšega algoritma vplivajo različne konfiguracije problemskega prostora, na primer število točk v prostoru, dimenzionalnost prostora, razpršenost točk, število iskanih sosedov in podobno. V nizkodimenzionalnih prostorih se za iskanje najbližjih sosedov uporabljo algoritmi, ki izvajajo strategijo razveji in omeji. Sem spadajo različne rekurzivne drevesne strukture, kjer vozlišča predstavljajo podprostor starša. Zaradi uravnoteženosti lahko v teh drevesih poiščemo list podane točke v logaritmičnem času. Slabost teh metod je, da je potrebno pri visokodimenzionalnih podatkih preveriti skoraj vsa vozlišča in je zato učinkovitost enaka kot pri izčrpnemu iskanju. Zato v teh primerih uporabljam algoritme, ki vrnejo približne najbližje sosede. Največkrat za to nalogu uporabljam prilagojene drevesne strukture in algoritme, ki temeljijo na preslikavah. Pri približnih algoritmih je poleg hitrosti iskanja pomembna tudi točnost vrnjenih objektov.

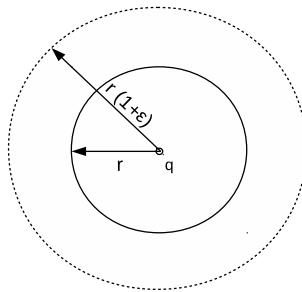
Cilj magistrskega dela je bila implementacija knjižnice v programskem jeziku python, ki omogoča uporabo obeh vrst iskanja, eksaktnega in približnega, saj običajno knjižnice v programskem jeziku python omogočajo le eno in se je zato potrebno naučiti uporabe večih knjižnic. Naša knjižnica [11] omogoča uporabo metod na preprost in enoten način. Nekatere algoritme smo implementirali sami, druge pa smo uporabili iz že obstoječih knjižnic. Hitrost in točnost algoritmov smo izmerili na različnih podatkovnih množicah.

V poglavju 2 opišemo najbolj uporabljane metode za iskanje najbližjih sosedov. Pri eksaktnih metodah opišemo strukture R [10],  $R^*$  [2], M [6], PM [19] in KD-drevo [3], pri približnih pa RKD-drevo [18], LSH [9], hierarhično razvrščanje z voditelji [1] in gozd robov [15]. V poglavju 3 opišemo uporabljeno mero razpršenosti podatkov LiNearN [21] in podatkovne množice, ki smo jih uporabili pri testiranjih. Razložimo postopek testiranja eksaktnih in približnih algoritmov ter omejitve pri testiranjih. V poglavju 4 predstavimo rezultate testiranj parametrov družin iskalnih algoritmov, nato sledijo rezultati meritev rabe pomnilnika. Opišemo rezultate testiranj eksaktnih in približnih metod, kjer smo zaradi razlike v hitrosti ločili python implementacije od C++ implementacij. Odločitveni drevesi za izbiro najprimernejše metode implementirane v jezikih python in C++ sta opisani v poglavju 5. V poglavju 6 povzamemo narejeno in podamo ideje za možne izboljšave. V dodatkih A in B vključimo podrobnejše grafe za vse podatkovne množice. V dodatku C prikažemo primer uporabe knjižnice za iskanje najbližjih sosedov z R-drevesom in tabelo z navedenimi parametri metod posameznih struktur.

## Poglavlje 2

# Metode za iskanje bližnjih sosedov

V tem poglavju so opisane metode, ki se najpogosteje uporablja za iskanje najbližjih sosedov. Med njimi je nekaj drevesnih struktur, ki najdejo eksaktne najbližje sosede, vendar imajo slabost, da je potrebno pri podatkih v visokih dimenzijah preiskati praktično celotno drevo in zato ni izboljšav v primerjavi z izčrpnim preiskovanjem. V visokih dimenzijah običajno uporabljamo metode, ki znajo poiskati  $\epsilon$ -približne najbližje sosede.



Slika 2.1: Prikaz napake pri iskanju približnih najbližjih sosedov.

Tu  $\epsilon$  predstavlja delež napake glede na dejanskega najbližjega soseda, zato je  $\epsilon > 0$ . Slika 2.1 prikaže razliko med iskanjem eksaktnih ter približnih

najbližjih sosedov, kjer lahko oddaljenost približnih sosedov nadzorujemo s parametrom  $\epsilon$ .

## 2.1 Eksaktne metode

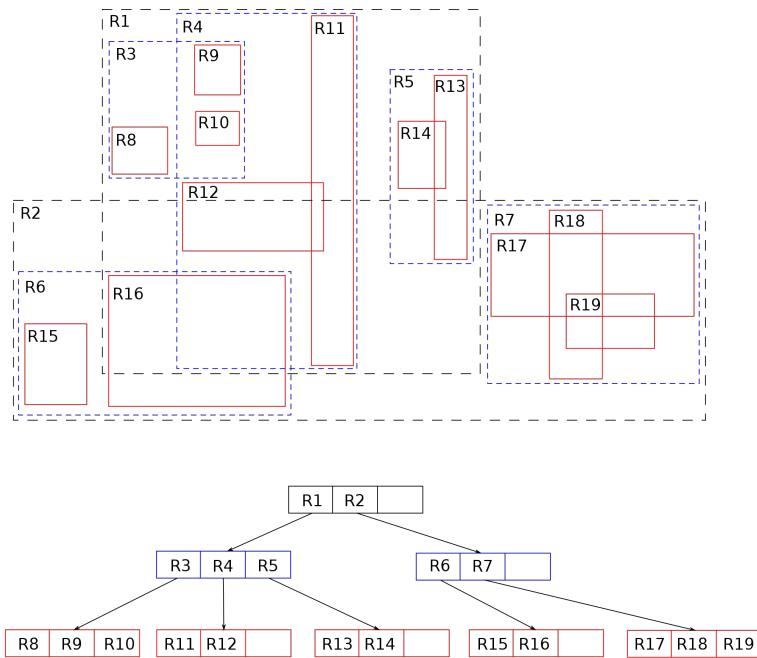
Najprej opišimo najbolj uporabljane metode za iskanje najbližjih sosedov pri nizkodimenzionalnih podatkih. Te metode so eksaktne, saj poznamo več struktur, ki nam omogočajo, da izvedemo operacije na drevesu v logaritmičnem času pri nizkodimenzionalnih podatkih. V nadaljevanju opišemo drevesa R, R\*, M, PM, ball in KD.

### 2.1.1 R-drevo

Ideja podatkovne strukture R-drevo [10] izhaja iz drevesne strukture B-drevo [7], ki je primerna za podatke v enodimenzionalnem prostoru. Gre za uravnoteženo drevo, kar nam zagotavlja hitro preiskovanje. Vsako vozlišče v drevesu ima število sinov na celoštevilskem intervalu  $[m, M]$ , kjer sta  $m$  in  $M$  parametra drevesa. Vsak izmed sinov predstavlja podprostor prostora starša. Prostori sinov se lahko med seboj prekrivajo, kar poskuša struktura minimizirati, saj je preiskovanje hitrejše, če je prekrivanja manj. Poleg tega vsako vozlišče hrani minimalen (hiper)pravokotnik, znotraj katerega so vsebovani vsi minimalni (hiper)pravokotniki sinov. Listi v drevesu vsebujejo podatke o objektih ter o minimalnem (hiper)pravokotniku, ki zajema vse te objekte. Primer dvodimenzionalnega R-drevesa z maksimalno velikostjo vozlišča 3 je predstavljen na sliki 2.2 [8]. Tu pravokotnik R3 predstavlja vozlišče v drevesu, ki vsebuje tri minimalne pravokotnike objektov in sicer R8, R9 in R10. Ti so, tako kot tudi preostali minimalni pravokotniki objektov, obarvani z rdečo barvo. Starš pravokotnika R3 je R1, ki je eden od dveh sinov korena drevesa.

Obstajajo različne izboljšave R-drevesa. Med najbolj znanimi sta  $R^+$  in  $R^*$ . Prva dodatno upošteva prekrivanje in tako zagotovi, da prekrivanja ni. Zaradi tega mora včasih določene objekte vstaviti v več različnih listov.

Hitrost iskanja je večja, saj do lista posamezne točke vodi vedno ena sama pot.  $R^*$  je v nadaljevanju bolj podrobno opisan, saj od vseh različic R-dreves zgradi najbolj učinkovito strukturo.



Slika 2.2: Primer R-drevesa v dvodimenzionalnem prostoru [8].

### 2.1.1.1 Vstavljanje

Vstavljanje objekta v drevo poteka po algoritmu 1. Najprej se določi list, v katerega se vstavi nov objekt. V primeru, da je z novim vnosom ta list postal preobsežen, se vozlišče razdeli na dva lista in oba postaneta sinova predhodnega vozlišča. Zatem se kliče metoda `AdjustTreeUpwards`, ki skrbi za pravilno strukturo drevesa po vnosu novega vozlišča tako, da popravlja minimalne pravokotnike vozlišč na poti do korena drevesa. V kolikor je neko vozlišče preobsežno se z razdelitveno metodo razdeli na 2 vozlišči. Ti dve vozlišči postaneta sinova nadrejenega vozlišča in postopek se ponovi. V primeru preobsežnosti korena drevesa se ustvari nov koren, katerega sinova sta vozlišči, ki sta nastali z razdelitvijo starega korena drevesa.

**Algoritem 1** Vstavljanje v R-drevo

---

```

1: procedure INSERT( $e$ )
2:    $\triangleright e$  is the entry object which we want to insert
3:    $\triangleright$  find the leaf in which the new entry should be put
4:    $l \leftarrow \text{ChooseLeaf}(e)$ 
5:   add  $e$  to leaf  $l$ 
6:    $ll \leftarrow \text{null}$ 
7:   if  $l$  overflowed then
8:      $\triangleright$  leaf  $l$  has too many children so it needs to be split
9:      $l, ll \leftarrow \text{SplitNode}(l)$ 
10:     $\triangleright$  adjust MBR's and perform additional splits if needed
11:    AdjustTreeUpwards( $l, ll$ )

```

---

V delu [10] so predstavljeni 3 načini razdelitve preobsežnega vozlišča na dva dela. Vsi načini sproti popravljajo minimalne pravokotnike vozlišč. Pri izbiri dela, v katerega bo vstavljen naslednji sin, se upoštevajo kriteriji v naslednjem vrstnem redu:

1. povečanje prostornine posamezne skupine,
2. velikost prostornine posamezne skupine in
3. trenutno število vozlišč v skupini.

Najpomembnejše je, da je povečanje prostornine čim manjše, saj se s tem prostor, ki ga predstavlja vozlišče najmanj poveča in je zato povečanje verjetnosti preiskave prostora pri iskanju najbližjih sosedov manjše. V kolikor je povečanje prostornine enako, je pomembno, da se sina vstavi v skupino, ki ima manjšo prostornino, saj je bolje imeti čim bolj primerljive prostornine obeh skupin. Če se tudi po tem ne da določiti skupine, izberemo tisto, ki ima manjše število sinov. S tem poskušamo izenačiti velikosti obeh skupin, saj se s tem lahko izognemo prehitrim delitvam, ki so lahko posledica slabega razmerja zapolnjenosti obeh skupin. V nadaljevanju opišemo tri pogosto uporabljene načine delitve.

### Linearna razdelitev

Pri linearni razdelitvi se najprej določita začetni vozlišči za vsako skupino. V vsaki dimenziji izberemo dve vozlišči, prvo z največjo spodnjo mejo in drugo z najmanjšo zgornjo mejo v trenutni dimenziji. Nato izračunamo oceno ločitve (2.1) izbranih vozlišč in sicer tako, da absolutno razliko med največjo spodnjo mejo in najmanjšo zgornjo mejo normaliziramo glede na zalogo vrednosti te dimenzije, pri čemer upoštevamo le vozlišča, ki so del razdelitve. Za začetni vozlišč velja, da imata največjo oceno ločitve. Vrstni red izbire naslednjega vozlišča je naključen.

$$separation_d = \frac{|highest\_low\_side_d - lowest\_high\_side_d|}{range_d} \quad (2.1)$$

### Kvadratna razdelitev

Podobno kot pri linearni razdelitvi se tudi pri kvadratni najprej določita začetni vozlišči. Naredimo vse možne kombinacije parov vozlišč in za vsako kombinacijo izračunamo minimalni pravokotnik, ki zajema ta par. Za vsaki vozlišči  $A$  in  $B$  izračunamo razdaljo, ki je definirana v enačbi (2.2), kjer  $V$  predstavlja prostornino.

$$d = V(MBR(A \cup B)) - V(MBR(A)) - V(MBR(B)) \quad (2.2)$$

Izberemo tisti par, kjer je vrednost  $d$  največja. Za izbiro naslednjega vozlišča za vsako vozlišče izračunamo povečanje prostornine vsake skupine ob dodajnju posameznega vozlišča. Izberemo tisto vozlišče, ki ima največjo absolutno razliko med obema povečavama.

### Izčrpna razdelitev

Generiramo vse možne razdelitve in izberemo najboljšo. Ta razdelitev je primerna le pri manjših velikostih vozlišč, saj obstaja eksponentno mnogo različnih kombinacij razdelitev.

### 2.1.1.2 Brisanje

Pri brisanju najprej preverimo, če drevo vsebuje podani objekt. V primeru vsebovanja ga izbrišemo iz lista in če ima sedaj list premajhno število objektov, list izbrišemo iz njegovega starša ter dodamo v seznam izbrisanih. Potem navzgor na isti način preverjamo ali ima katero vozlišče premalo sinov in če jih ima, ga izbrišemo iz drevesa ter dodamo v seznam izbrisanih. Na koncu se sprehodimo čez izbrisana vozlišča in v primeru lista njegove objekte ponovno vstavimo na enak način, kot je opisano pri postopku vstavljanja. V primeru notranjega vozlišča to vozlišče ponovno vstavimo v drevo in sicer na isto višino, na kateri je bil, preden smo ga izbrisali. S tem zagotovimo, da vozlišča ostanejo na istih višinah in ohranimo uravnoteženost drevesa.

### 2.1.1.3 Iskanje najbližjih sosedov

Iskanje najbližjih sosedov v R-drevesu [5] poteka tako, da se rekurzivno spuščamo po drevesu navzdol in preverjamo, katere od sinov je potrebno pregledati, da lahko določimo najbližje sosede. To preverjanje poteka tako, da za vsakega sina izračunamo razdaljo MINDIST (2.3), ki predstavlja razdaljo med pravokotnikom (v 2-dimenzionalnem prostoru) in točko  $p$ , katere najbližje sosede iščemo:

$$MINDIST(p, R) = \sum_{i=1}^n |p_i - r_i|^2, \quad (2.3)$$

kjer  $i$  predstavlja dimenzijo,  $r_i$  pa oddaljenost komponente  $p_i$  od intervala, ki ga zavzema pravokotnik  $R$  v  $i$ -ti dimenziji.

Če je ta razdalja večja od oddaljenosti trenutnega k-najbližjega soseda, potem vemo, da nam poddrevesa računanega sina ni potrebno preiskati. Rekurziven postopek iskanja k-najbližjih sosedov je predstavljen v algoritmu 2.

---

**Algoritem 2** Iskanje k-najbližjih sosedov v R-drevesu

---

```

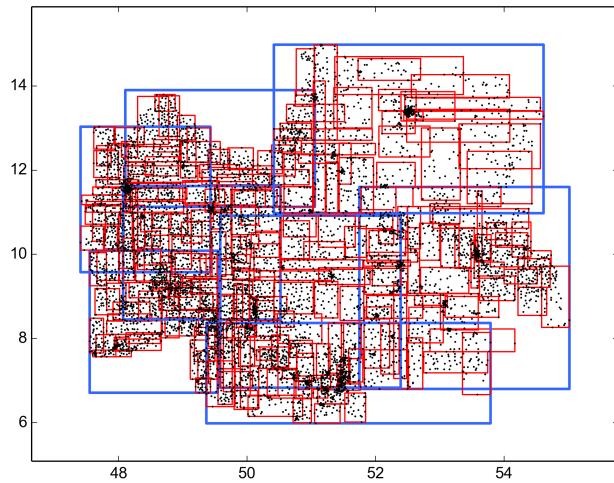
1: procedure KNN_SEARCH(node, p, k)
2:   ▷ node represents current node, p is the query point and k is the
      number of nearest neighbours to return
3:   if node is Leaf then
4:     for each entry in node do
5:       if entry is closer to p than current k-th NN then
6:         add entry in current k-NNs and remove previous k-th NN
7:     else
8:       ▷ ABL is a list of children of current node, sorted by ascending
          distance MINDIST(p, MBR(child))
9:       ABL  $\leftarrow$  Active Branch List of node
10:      for each child in ABL do
11:        d  $\leftarrow$  MINDIST(p, MBR(child))
12:        if d > distance(p, k-thNN) then
13:          ▷ because ABL is sorted by MINDIST other children are
              further away so none of them can contain closer points
14:        return current k-nearest neighbours
15:      else
16:        KNN_SEARCH(child, p, k)

```

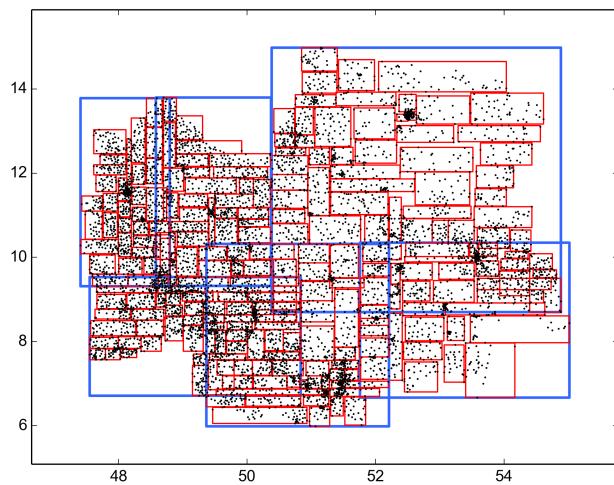
---

### 2.1.2 R\*-drevo

Gre za izboljšavo R-drevesa, kjer je spremenjen samo postopek vstavljanja novih objektov v strukturo. Za razliko od R-drevesa, ki poskuša optimizirati le prostornino pravokotnikov vozlišč drevesa, poskuša R\*-drevo optimizirati tudi prekrivanje med njimi in njihov obseg [2]. Z minimiziranjem prekrivanja se zmanjša število poti, ki jih je potrebno preiskati med postopkom vstavljanja novih točk in iskanja najbližjih sosedov. Minimizacija obsega pravokotnikov pa povzroči, da so pravokotniki bolj kvadratne oblike. Struktura R in R\*-drevesa na množici nemških poštnih številk je prikazana na slikah 2.3 in 2.4, kjer opazimo manjše prekrivanje pri R\*-drevesu.



Slika 2.3: Prikaz zgrajenega R-drevesa na množici točk, ki predstavlja lokacije nemških poštnih številk.



Slika 2.4: Prikaz zgrajenega R\*-drevesa na množici točk, ki predstavlja lokacije nemških poštnih številk.

To je zaželjeno, saj se s tem zmanjša prekrivanje in je bolj primerno za računanje MINDIST razdalje. Postopek uporablja tudi strategijo ponovnega vstavljanja. Zaradi močne odvisnosti strukture od zaporedja vstavljanja točk običajno ne dobimo optimalne strukture. Ponovno vstavljanje določenih poddreves oziroma objektov popravlja drevesno strukturo in s tem povzroči, da je višina drevesa manjša, kar omogoča hitrejše iskanje po drevesu.

### 2.1.2.1 Vstavljanje

Postopek vstavljanja je podoben kot pri R-drevesu. Razlika je v iskanju lista, v katerega se nov objekt vstavi in v obravnavanju preobsežnih vozlišč. Pri iskanju lista iščemo še vedno najmanjše povečanje prostornine, razen v primeru, ko so sinovi trenutnega vozlišča listi. Takrat izbere list, pri katerem vnos objekta povzroči najmanjše povečanje prekrivanja med izbranim listom in ostalimi sinovi trenutnega vozlišča. V primeru preobsežnega vozlišča lahko pride do razdelitve preobsežnega vozlišča ali pa do ponovnega vnosa nekaterih sinov tega vozlišča. V kolikor med vnosom objekta še ni prišlo do obravnavanja preobsežnega vozlišča na istem nivoju, izvedemo ponovno vstavljanje in nekaj sinov preobsežnega vozlišča izbrišemo in ponovno vstavimo v drevo. V drugem primeru pride do razdelitve preobsežnega vozlišča. Delež ponovnega vstavljanja je parameter, običajno najboljše rezultate dobimo, če ponovno vstavimo 30 odstotkov sinov. Pri ponovnem vstavljanju moramo biti pozorni, da vstavimo poddrevesa tako, da ostanejo vsi listi na isti višini. S tem zagotovimo, da je drevo tudi po ponovnem vstavljanju poddreves še vedno uravnoteženo.

### Razdelitev v $R^*$ -drevesu

Pri razdelitvi se najprej določi razdelitvena dimenzija, kar je prikazano v algoritmu 3. Sprehodimo se čez vsako dimenzijo in računamo vsoto obsegov minimalnih pravokotnikov pri različnih razvrstitvah in razdelitvah. To storimo tako, da razvrstimo sinove preobsežnega vozlišča enkrat padajoče in enkrat naraščajoče, glede na vrednost njihovega minimalnega pravokotnika

v izbrani dimenziji. Nato za vsako razvrstitev generiramo vse možne razdelitve sinov v dve disjunktni množici, pri čemer ima vsaka množica najmanj  $m$  in največ  $M$  elementov, saj moramo obdržati lastnost drevesa, da je število sinov na intervalu  $[m, M]$ . Takih razdelitev je točno  $M - 2m + 2$ . Za  $k$ -to razdelitev velja, da prva množica vsebuje prvih  $(m - 1) + k$  sinov, druga pa preostale. Tako se določita minimalna pravokotnika za obe množici in njun obseg. Obseg se doda k vsoti obsegov za trenutno dimenzijo. Na koncu izberemo kot razdelitveno dimenzijo tisto dimenzijo, ki ima največjo vsoto obsegov. Zatem izberemo najboljšo razdelitev sinov preobsežnega vozlišča v dve disjunktni množici. Postopek je podoben, kot pri iskanju razdelitvene dimenzije, saj zopet razvrstimo sinove na oba načina in generiramo razdelitve. Razlika je v tem, da namesto skupnega obsega računamo skupno prekrivanje in skupno prostornino. Na koncu izberemo razdelitev, ki ima najmanjše skupno prekrivanje. V primeru, da je skupno prekrivanje isto za več različnih razdelitev, izberemo tisto razdelitev, ki ima manjšo skupno prostornino.

---

**Algoritem 3** Izbor razdelitvene dimenzijske

---

```

1: procedure CHOOSE_SPLIT_AXIS(node)
2:   split_axis  $\leftarrow 0$ 
3:   highest_margin  $\leftarrow 0$ 
4:   for each dimension d do
5:     sl  $\leftarrow$  children sorted by their lower value in dimension d
6:     sh  $\leftarrow$  children sorted by their higher value in dimension d
7:     margin  $\leftarrow 0$ 
8:      $\triangleright$  m and M are parameters of tree
9:     for each k in  $[0, M - 2m + 2)$  do
10:       $\triangleright$  calculate margin for sl sorting
11:      group1l  $\leftarrow$  first  $(m - 1) + k$  elements in sl
12:      group2l  $\leftarrow$  all remaining elements in sl
13:      margin  $\leftarrow$  margin + margin(group1l) + margin(group2l)
14:       $\triangleright$  calculate margin for sh sorting
15:      group1h  $\leftarrow$  first  $(m - 1) + k$  elements in sh
16:      group2h  $\leftarrow$  all remaining elements in sh
17:      margin  $\leftarrow$  margin + margin(group1h) + margin(group2h)
18:      if margin  $>$  highest_margin then
19:         $\triangleright$  found a better split axis
20:        highest_margin  $\leftarrow$  margin
21:        split_axis  $\leftarrow d$ 
22:   return split_axis

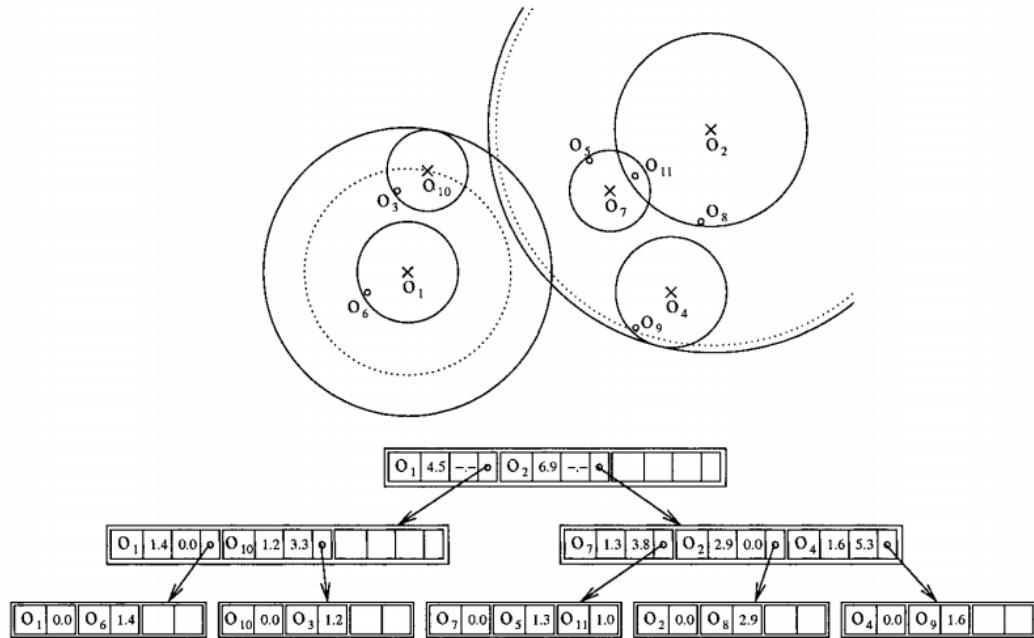
```

---

### 2.1.3 M-drevo

M-drevo je drevesna struktura, pri kateri so podprostori definirani s hiperkroglami [6]. Struktura je podobna R-drevesu, le da vozlišče v npr. 2-dimenzionalnem prostoru namesto minimalnega pravokotnika definira krog, ki vsebuje celoten podprostor sinov. Podobno kot v R-drevesu imamo dve vrsti vozlišč, notranja vozlišča in liste, vsako izmed vozlišč lahko vsebuje

največ  $m$  sinov. Notranja vozlišča so predstavljena kot seznam terk oblike  $(O_r, r, d(O_r, P(O_r)), ptr)$ , kjer  $O_r$  predstavlja objekt, ki definira center kroga, in  $r$  njegov polmer. Skupaj tako definirata podprostor, ki vsebuje vse podprostore sinov, kar pomeni, da so sinovi za največ  $r$  oddaljeni od centra starša. Polmer  $r$  ni nujno najmanjši, saj struktura ne računa najmanjšega polmera ampak se zadovolji s polmerom, ki zagotovljeno zajame vse sinove. S tem se izognemo pogostemu računanju razdalj in pospešimo postopek vstavljanja. Poleg tega hranimo  $d(O_r, P(O_r))$ , ki predstavlja razdaljo centra vozlišča do centra njegovega starša. S tem podatkom si pomagamo pri iskanju najbližjih sosedov, kjer lahko določene veje izpustimo, saj v prostoru velja trikotniška neenakost. Kazalec  $ptr$  kaže na seznam terk sinov. Listi drevesa vsebujejo le seznam terk oblike  $(O_i, d(O_i, P(O_i)))$ . V listih se nahajajo objekti, ki so predstavljeni z  $O_i$ . Poleg objekta  $O_i$  terka vsebuje tudi razdaljo objekta do centra njegovega starša, kar je definirano z  $d(O_i, P(O_i))$ . Za vse sinove poljubnega vozlišča velja, da so za največ  $r$  oddaljeni od starša. Tako kot R-drevo je tudi M-drevo dinamična struktura, kar pomeni, da lahko po iskanju najbližjih sosedov dodatno razširimo drevo z novimi objekti in se nato ponovno osredotočimo na iskanje najbližjih sosedov, kar pri statičnih strukturah ni mogoče. Običajno pride pri M-drevesu do večjega prekrivanja kot pri R-drevesu, saj je težje izvesti dobro razdelitev preobsežnih vozlišč. Primer M-drevesa z 10 točkami je prikazan na sliki 2.5. Koren drevesa vsebuje dva sinova. Prvi vsebuje objekt  $O_1$  kot center kroga s polmerom 4.5 in dvema sinovoma, ki sta lista v drevesu. Prvi list vsebuje objekta  $O_1$  in  $O_6$  in ima razdaljo do starša enako 0, ker je center kroga tega lista ista točka, kot center kroga starša, torej objekta  $O_1$ . Prostor drugega lista je predstavljen s krogom s središčem v točki objekta  $O_{10}$  in s polmerom 1.2. Znotraj tega prostora se nahajata objekta  $O_{10}$  in  $O_3$ . Iz slike je razvidno, da polmeri krogov niso minimalni, saj se v nasprotnem primeru sinova korena ne bi prekrivala. Njuna minimalna kroga sta predstavljena s pikčasto obrobo.



Slika 2.5: Primer M-drevesa v dvodimenzionalnem prostoru z 10 točkami [22].

### 2.1.3.1 Vstavljanje

Pri vstavljanju v M-drevo najprej izberemo najbolj primeren list. Postopek se začne v korenju drevesa in postopoma se spuščamo po drevesu navzdol do lista. Pri tem vedno izberemo sina, pri katerem je povečanje polmera čim manjše. V kolikor je več takih sinov izberemo tistega, pri katerem je razdalja od centra sina do objekta najkrajša. Z iskanjem minimalnega povečanja polmera poskušamo doseči, da so prostori vozlišč čimmanjši, medtem ko izbor sina, ki je čimmanj oddaljen od centra koristi pri razdelitvah vozlišč. Kot pri ostalih podobnih strukturah lahko tudi pri M-drevesu pri vstavljanju pride do preobsežnih vozlišč, kar rešujemo z razdelitvijo vozlišč.

### Razdelitev vozlišč v M-drevesu

Razdelitev je sestavljena iz dveh delov, povišanja in razbitja. Pri povišanju se določita centra vozlišč, ki sta posledica razdelitve preobsežnega vozlišča.

Obstajajo različne metode določitve centrov. Med najbolj znanima sta izbira naključnih centrov ter metoda imenovana mM\_RAD. Ta metoda poskuša optimizirati izbiro pivotov iz skupine  $N$  tako, da za pivota izbere točki  $p_1$  in  $p_2$ , kjer velja kar je prikazano v enačbi 2.4 [22]. Metoda razbitja sprejme centra skupin ter razdeli ostale sinove v eno izmed dveh skupin. Najpogosteje uporabljeni metoda se sprehodi čez sinove ter jih dodeli v skupino, katere center je bližji centru sina. Iz obeh skupin se ustvarita novi vozlišči in obe se dodelita staršu prej preobsežnega vozlišča. V kolikor starš postane preobsežen se postopek rekurzivno ponovi. Če postane preobsežen koren drevesa, se ustvari nov koren in se tako višina drevesa poveča za ena.

$$\max(\text{radij}(p_1), \text{radij}(p_2)) \leq \max(\text{radij}(p_i), \text{radij}(p_j)) \forall p_i, p_j \in N \quad (2.4)$$

### 2.1.3.2 Iskanje najbližjih sosedov

Iskanje najbližjih sosedov v M-drevesu poteka podobno kot pri R-drevesu. Razlika je v tem, da v primeru notranjega vozlišča uredimo sinove naraščajoče glede na DMIN razdaljo namesto glede na MINDIST (2.3). Razdalja DMIN (2.5) sprejme kot parametra vozlišče  $n$  ter točko  $q$  in je definirana kot:

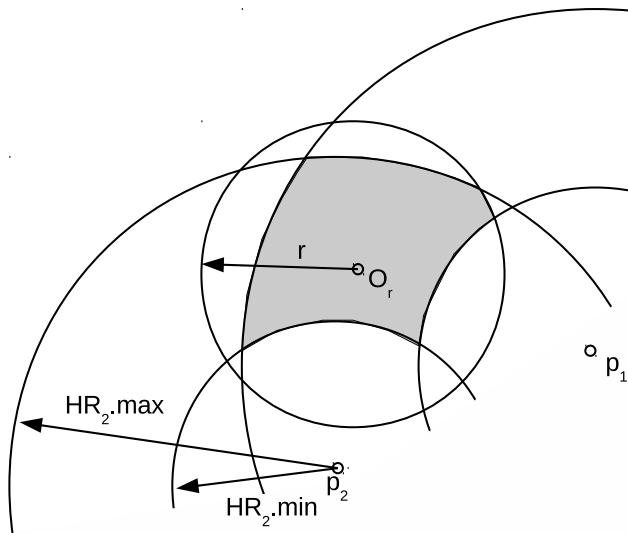
$$DMIN(n, q) = \max(0, d(O_r(n), q) - r_n), \quad (2.5)$$

kjer  $d(O(n), O(q))$  predstavlja razdaljo med centrom vozlišča  $n$  in točko  $q$ ,  $r_n$  pa polmer vozlišča  $n$ . Vse ostalo, razen kriterija ureditve sinov, je identično postopku iskanja najbližjih sosedov v R-drevesu.

### 2.1.4 PM-drevo

PM-drevo je implementacija M-drevesa s pivoti [19]. Pri tem drevesu je prostor vozlišča poleg hiperkrogle definiran tudi s  $p$  hiperobroči, ki dodatno krčijo prostor. Pri tem  $p$  predstavlja število pivotov in je parameter drevesa. Namen pivotov je dodatno zmanjšati prostor, ki ga definira vozlišče, kar je razvidno iz slike 2.6. Pivote predstavljajo objekti, pri čemer boljša izbira pivotov privede do manjših prostorov vozlišč in posledično do

hitrejšega preiskovanja. Zaradi dodatnega preverjanja vsebovanosti točke v vseh hiperobročih je potrebno biti pazljiv, da ne izberemo prevelikega  $p$ , saj s tem upočasnimo preiskovanje. Zaradi manjših prostorov se pregleda manj objektov in je zato ta struktura dobra izbira pri razdaljah, ki so računsko zahtevnejše. Ob izgradnji drevesa se izmed podanih objektov izbere  $p$  takih, ki bodo predstavljeni pivote drevesa. Obstajajo različni načini izbire pivotov, med najbolj uporabljenimi sta naključna metoda ter metoda, ki sestavi  $g$  skupin s po  $p$  naključnimi pivoti in izbere tisto skupino, pri kateri je vsota razdalj med pivoti največja. Vozlišča v drevesu morajo imeti dodatno informacijo, ki definira njihove hiperobroče. Tako terke notranjih vozlišč vsebujejo dodatno podatek  $HR$ , ki je seznam velikosti  $p_{hr}$ , kjer velja  $p_{hr} \leq p$ . V tem seznamu je  $i$ -ti element terka  $(min_i, max_i)$ , ki definira svoj hiperbroč. Ta se nanaša na  $i$ -ti pivot in zavzema prostor hiperkrogla s centrom v  $i$ -tem pivotu ter polmerom  $max_i$  brez prostora, ki ga zavzema hiperkrogla z istim centrom ter polmerom  $min_i$ . Na ta način vsako notranje vozlišče vsebuje informacijo o vseh svojih  $p_{hr}$  hiperobročih. Lastnost posameznega hiperobroča je, da je minimalen hiperbroč s centrom v posameznem pivotu, ki zajema vse objekte vozlišča. Torej za vsako notranje vozlišče  $n$  velja  $HR_i.min = min(d(O_j, p_i)) \forall O_j \in n$  in  $HR_i.max = max(d(O_j, p_i)) \forall O_j \in n$ . Tako je prostor notranjega vozlišča definiran kot presek njegove hiperkrogle in njegovih hiperobročev. Podobno tudi terke v listih vsebujejo dodatno informacijo, vezano na pivote, ki se hrani v seznamu imenovanem  $PD$  in so zato oblike  $(O_i, d(O_i, P(O_i), PD))$ . Seznam  $PD$  vsebuje  $p_{pd}$  elementov, kjer  $i$ -ti predstavlja razdaljo med objektom terke ter  $i$ -tim pivotom. Poleg tega mora veljati  $p_{pd} \leq p_{hr}$ . Slika 2.6 prikazuje primer prostora dvodimensionalnega notranjega vozlišča, ki je predstavljen s terko  $(O_r, r, d(O_r, P(O_r), HR))$ , pri dveh pivotih označenih s  $p_1$  in  $p_2$ . Presek obročev obeh pivotov ter kroga, ki ga definira to vozlišče predstavlja prostor, ki ga zajema vozlišče in je na sliki obarvan sivo. Struktura PM-drevo deluje na poljubni metriki, torej ne zahteva postavitve točk v koordinatni sistem kot to zahteva R-drevo. Primer take metrike je Levenshteinova razdalja.



Slika 2.6: Primer PM-drevesa v dvodimenzionalnem prostoru z dvema pivottoma.

#### 2.1.4.1 Vstavljanje

Vstavljanje poteka na isti način kot pri M-drevesu z razliko, da je sproti potrebno skrbeti za  $HR$  in  $PD$  sezname. Preden vstavimo nov objekt v list moramo zgraditi  $PD$  seznam za terko, ki vsebuje ta objekt. Ta seznam vsebuje razdalje  $d(O_i, P_j) \forall j \leq p_{pd}$ . Zatem moramo popravljati  $HR$  sezname vozlišč, ki so na poti od izbranega lista do korena, saj mora držati, da vsak hiperobroč vsebuje vse objekte svojega poddrevesa, torej mora vsebovati tudi na novo vstavljen objekt. Metoda razdelitve je enaka, vendar moramo biti pozorni, da pravilno izračunamo nove  $HR$  sezname za obe novi vozlišči. Za posamezno vozlišče  $n$  jih izračunamo, kot

$$HR(n) = HR(c_1) \cup HR(c_2) \dots \cup HR(c_k),$$

kjer  $c_i$  predstavlja  $i$ -tega sina vozlišča  $n$  in  $k$  število sinov tega vozlišča.

### 2.1.4.2 Iskanje najbližjih sosedov

Algoritem iskanja najbližjih sosedov je podoben kot pri M-drevesu. Dodatno je potrebno pred začetkom iskanja najbližjih sosedov točke  $q$  izračunati razdalje  $d(q, P_t) \forall t \leq \max(p_{hr}, p_{pd})$  in si jih shraniti. Posamezno notranje vozlišče  $n$  lahko izpustimo, če ne velja  $|d(P(n), q) - d(n, P(n))| \leq nn_k + r_n$ . Tu  $nn_k$  predstavlja trenutno razdaljo do  $k$ -tega najbližjega soseda. Ta pogoj je enak kot pri M-drevesu, vendar lahko pri PM-drevesu dodatno izločimo vsa notranja vozlišča  $n$ , kjer pri vsaj enim izmed hiperobročev ne velja  $d(q, P(t)) - nn_k \leq HR_t.\max \wedge d(q, P(t)) + nn_k \geq HR_t.\min$ . Podobno moramo tudi v listih drevesa preverjati, katere razdalje do objektov je potrebno računati. Izkaže se, da je potrebno preverjati le objekte  $O_i$ , kjer velja  $|d(P(O_i), q) - d(O_i, P(O_i))| \leq nn_k$ . Razdalje do posameznega objekta pa ni potrebno izračunati tudi v primeru, ko za vsaj enega izmed  $p_{pd}$  pivotov ne velja  $absd(q, P_t) - d(O_i, P_t) \leq nn_k$ . Ti dodatni pogoji so razlog, da pri PM-drevesu izračunamo manj razdalj. Zaradi dodatnega preverjanja je celoten postopek hitrejši le, če je funkcija razdalje zahtevna za računanje, saj le v takem primeru s PM-drevesom pridobimo v primerjavi z navadnim M-drevesom. Prav zaradi tega običajno velja  $p_{pd} < p_{hr}$ , ker je bolj pomembno, da z dodatnim preverjanjem preprečimo preiskovanje poddreves, kot računanje razdalj. DMIN se izračuna, kot je prikazano s formulo 2.6, kjer  $O_r(n)$  predstavlja center vozlišča  $n$ ,  $r_n$  pa njegov polmer. Za razliko od M-drevesa moramo pri PM-drevesu izračunati dodatno  $d_{HR.\max}^{\text{low}}$ , ki preveri minimalno oddaljenost z zgornjimi mejami hiperobročev in  $d_{HR.\min}^{\text{low}}$ , naredi enako s spodnjimi mejami.

$$\begin{aligned}
 DMIN(n, q) &= \max(0, d(O_r(n), q) - r_n, d_{HR.\max}^{\text{low}}, d_{HR.\min}^{\text{low}}) \\
 d_{HR.\max}^{\text{low}} &= \max \bigcup_{t=1}^{p_{hr}} (d(q, P_t) - HR_t.\max) \\
 d_{HR.\min}^{\text{low}} &= \max \bigcup_{t=1}^{p_{hr}} (HR_t.\min - d(q, P_t))
 \end{aligned} \tag{2.6}$$

## 2.1.5 Ball-drevo

Ball-drevo je drevesna struktura, ki je podobna strukturi M-drevo, saj so podprostori prav tako definirani s hiperkroglami [17]. Gre za uravnoteženo binarno drevo, kjer notranja vozlišča vsebujejo podatke o centru in radiju, listi pa vstavljeni točki. Za razliko od M-drevesa, pri ball-drevesu za center vozlišča ne izberemo ene izmed točk, ki se nahaja v strukturi, ampak ga izračunamo. Zato struktura ne deluje na poljubni metriki, ker potrebuje postavitev točk v koordinatnem sistemu. Prostor vozlišča je vedno minimalen prostor, ki vsebuje vse prostore sinov.

### 2.1.5.1 Vstavljanje

Najbolj znan algoritem zgradi drevo na podoben način kot KD-drevo, ki je opisano v nadaljevanju. Algoritem zahteva, da so vse točke podane vnaprej. Drevo gradimo od zgoraj navzdol tako, da najprej določimo center in radij korena drevesa. Center izračunamo kot povprečje vrednosti koordinat točk v vsaki dimenziji, razdalja med centrom korena in najbolj oddaljeno točko pa predstavlja radij. Nato izberemo dimenzijo  $d$ , ki ima največji razpon, in poiščemo njen mediano  $m$ . Točke razdelimo v dve skupini glede na njihovo vrednost v izbrani dimenziji. Če je vrednost  $\leq m$  potem točko uvrstimo v prvo skupino, drugače v drugo. Vsaka izmed skupin predstavlja vozlišče, katerega starš je koren drevesa. Postopek rekurzivno ponavljamo na točkah obeh vozlišč in se ustavimo, ko zmanjka točk.

### 2.1.5.2 Iskanje najbližjih sosedov

Iskanje najbližjih sosedov točke  $q$  poteka na enak način, kot pri M-drevesu. Rekurzivno se spuščamo po drevesu navzdol in najprej obiščemo sina, ki ima razdaljo od centra do točke  $q$  manjšo. Tako se sprehodimo do lista, izračunamo njegovo razdaljo do točke  $q$ . V kolikor je manjša od trenutnega  $k$ -tega najbližjega sosedja  $p$ , potem popravimo trenutne najbližje sosedе. Zatem se rekurzivno vračamo po drevesu navzgor in preverjamo ali moramo preiskati

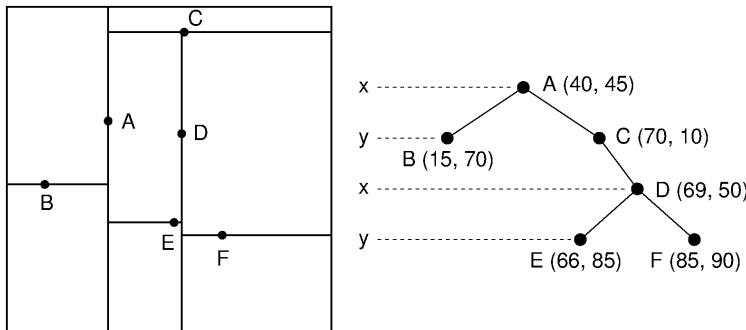
tudi preostala vozlišča. Vozlišča nam ni potrebno preiskati, če je razdalja med  $q$  in najbližjo točko vozlišča večja od razdalje med točko  $q$  in  $p$ .

### 2.1.6 KD-drevo

KD-drevo [3] je najbolj razširjena struktura za iskanje najbližjih sosedov pri nizko-dimenzionalnih podatkih. Gre za binarno drevo, kjer je vsako vozlišče predstavljeno z d-dimenzionalno točko  $V$ . V osnovni načini se na vsaki višini določi dimenzija, glede na katero se prostor razdeli na dva manjša podprostora in sicer je ta dimenzija enaka  $d = \text{height} \mod K$ . Tako vsako notranje vozlišče v drevesu razdeli prostor na dva dela in sicer levi sin predstavlja prostor, kjer je  $V_d$  manjši od starša, desni sin pa preostali prostor. Boljše načice prostor razdelijo boljše, npr. dimenzija, glede na katero se prostor razdeli, se lahko izbere pametnejše, če vnaprej poznamo podatke, v ločitveni dimenziji za delitev prostora izberemo mediano vrednosti sinov ipd. Vsebovanost točke v drevesu zlahka preverimo, saj je zaradi strukture drevesa potrebno preiskati eno samo vejo drevesa. Slika 2.7 prikazuje primer 2-dimenzionalnega drevesa, kjer točka A predstavlja koren drevesa in deli glede na os  $x$ , kar pomeni, da njen levi sin predstavlja vse točke, ki imajo vrednost v dimenziji  $x$  manjšo kot točka A, desni sin pa vse preostale. Vozlišča na drugem nivoju razdelijo prostor glede na  $y$  os, na tretjem nivoju zopet glede na  $x$  os itd. Iz slike je razvidno, da lahko pri KD-drevesu pride do slabše uravnovezenosti drevesa. V najslabšem primeru dobimo eno samo verigo, kar pomeni, da je potreben linearen čas za vstavljanje, brisanje ter iskanje.

#### 2.1.6.1 Vstavljanje

Točko  $p$  v drevo vstavimo tako, da se spuščamo po drevesu v sinove, katerih prostor vsebuje točko  $p$ . To počnemo, dokler ne pridemo do lista  $V$ , takrat listu dodamo točko  $p$  kot levega sina, če je vrednost  $p_d$  manjša od  $V_d$ , v nasprotnem primeru pa postane točka  $p$  desni sin lista  $V$ . Zaradi načina vstavljanje je uravnovezenost drevesa odvisna od razpršenosti podatkov ter vrstnega reda vstavljanja točk.



Slika 2.7: Primer k-d drevesa v dvodimensionalnem prostoru.

### 2.1.6.2 Brisanje

Brisanje poteka na podoben način kot vstavljanje, le da med spuščanjem po drevesu sproti pregledujemo, če smo že prišli do željene točke. Ko pridemo do točke, jo izbrišemo in po potrebi iz poddrevesa izbrisane točke določimo točko  $q$ , ki bo omenjeno točko nadomestila. Nato rekurzivno izbrišemo točko  $q$  iz poddrevesa izbrisane točke in s tem se postopek brisanja konča.

### 2.1.6.3 Iskanje najbližjih sosedov

Postopek iskanja najbližjih sosedov je opisan v algoritmu 4. Najprej se spustimo do lista drevesa ter točko v tem listu dodamo med najbližje sosede. Nato se rekurzivno vračamo navzgor po drevesu in na vsakem vozlišču preverimo:

1. ali je točka v trenutnem vozlišču med trenutnimi  $k$ -najbližjimi sosedi in
2. ali obstaja možnost, da je v drugem poddrevesu točka, ki je bližje, kot je trenutna  $k$ -ta najbližja točka.

Drugi pogoj preverimo tako, da izračunamo, ali hiperkrogla s središčem v točki vozlišča in z radijem oddaljenosti iskane točke do trenutnega  $k$ -najbližjega soseda seka prostor, ki ga predstavlja drugi sin. Če ga, je potrebno preiskati tudi prostor tega sina, saj se znotraj njega lahko nahaja

---

točka, ki je bližje kot trenutni k-najbližji sosed.

---

**Algoritem 4** Iskanje k-najbližjih sosedov v k-d drevesu
 

---

```

1: procedure KNN_SEARCH(node, p, k)
2:   ▷ node represents current node, p is the query point and k is the
      number of nearest neighbours to return
3:   if node is Leaf then
4:     if node is closer to p than current k-th NN then
5:       add node to current NNs and remove previous k-th NN
6:   else
7:     ▷ d is splitting dimension of current node
8:     d  $\leftarrow$  node.d
9:     if pd < noded then
10:    KNN_SEARCH(node.left, p, k)
11:    if node is closer to p than current k-th NN then
12:      add node to current NNs and remove previous k-th NN
13:      if hypersphere(node, k-thNN)  $\cap$  hyperplane(node.right)  $\neq \emptyset$ 
        then
14:        ▷ the right subtree might contain a point, which is closer
           to p than the current k-th NN
15:        KNN_SEARCH(node.right, p, k)
16:   else
17:     KNN_SEARCH(node.right, p, k)
18:     if node is closer to p than current k-th NN then
19:       add node to current NNs and remove previous k-th NN
20:       if hypersphere(node, k-thNN)  $\cap$  hyperplane(node.left)  $\neq \emptyset$ 
        then
21:         ▷ the left subtree might contain a point, which is closer
           to p than the current k-th NN
22:         KNN_SEARCH(node.left, p, k)
  
```

---

## 2.2 $\epsilon$ -približne metode

Pri večdimenzionalnih podatkih se za hitro iskanje najbližjih sosedov uporabljajo metode, ki iščejo  $\epsilon$ -približne najbližje sosede, saj eksaktne ne delujejo dobro na takih podatkih. Razlog je, da z večanjem dimenzij prostor hitro narašča, število točk pa ostaja približno isto in so zato točke na redko razporejene v prostoru. To povzroči, da so razdalje med točkami velike in je zato potrebno preiskati večino točk. Obstajajo različne metode iskanja približnih najbližjih sosedov. Najpogosteje so to prilagojene drevesne strukture in lokalno-občutljive funkcije razprševanja. Pri drevesnih strukturah običajno skrčimo podatke in sosede iščemo na novih podatkih, medtem ko z lokalno-občutljivimi funkcijami razprševanja poskušamo uvrstiti objekte, ki so si med seboj blizu, v isti koš in tam iščemo sosede. V nadaljevanju opišemo metode RKD-drevo, LSH, večrazlično LSH razprševanje, hierarhično razvrščanje z voditelji in gozd robov.

### 2.2.1 RKD-drevo

RKD-drevo predstavlja različico KD-drevesa, ki je primerna za iskanje najbližjih sosedov v visoko dimenzionalnih podatkih [18]. Princip je podoben kot pri KD-drevesu, le da namesto enega drevesa zgradimo  $m$  dreves. Drevesa se med seboj razlikujejo po dimenzijah, glede na katere se prostori delijo v posameznih nivojih dreves. Preden zgradimo drevesa, je potrebno ugotoviti varianco v posameznih dimenzijah. Izmed vseh dimenzij izberemo  $n$  dimenzij z največjo varianco. Te so v množici potencialnih kandidatov za deljenje, ostalih ne upoštevamo. Tako za posamezno drevo na vsakem nivoju naključno izberemo dimenzijo iz omenjene množice. Če imamo  $m$  neodvisnih dreves in je verjetnost, da ne najdemo najbližjega soseda v posameznem drevesu  $p$ , potem je pri iskanju v  $m$  neodvisnih drevesih verjetnost enaka  $p^m$ . V naših poskusih smo uporabili implementacijo RKD-drevesa iz FLANN [16] C++ knjižnice, ki je povezana s programskim jezikom python. Ta implementacija izbere 5 dimenzij z največjo varianco in na njih gradi drevesa. Na

vsakem nivoju približno izračuna povprečno vrednost v izbrani dimenziji, tako da upošteva le 100 vektorjev, z namenom da pospeši gradnjo. Podobno uporabimo tudi python vezavo na C++ implementacijo KD-drevesa knjižnice Annoy [4]. Ta za razliko od FLANN implementacije ne upošteva variance dimenzij, ampak na vsakem nivoju poskuša najti naključno hiperravnino, ki razdeli točke na dva dela. Pri tem se zadovolji s hiperravnino, ki razdeli točke tako, da je v vsakem delu vsaj kakšna točka.

### 2.2.1.1 Iskanje najbližjih sosedov

Ker uporabljam le  $n$  dimenzij, se za računanje razdalj med točkama izračuna razdalja le glede na te dimenzije. S tem se hitrost računanja razdalje precej poveča, saj večino dimenzij zanemarimo in glede na to, da so drevesa zgrajena samo na teh dimenzijah, ničesar ne izgubimo. Drevesa preiščemo vzporedno (z več nitmi), hranimo pa eno skupno množico najbližjih sosedov za vsa drevesa. S tem so preiskave poddreves odvisne od preiskovanja preostalih dreves in se zmanjša število vozlišč, ki jih je potrebno preiskati.

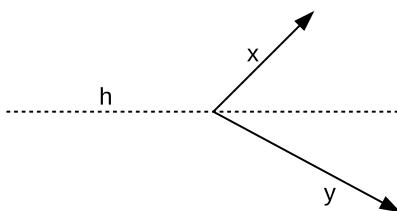
### 2.2.2 Lokalno-občutljive funkcije razprševanja

Med najuspešnejšimi metodami približnega iskanja najbližjih sosedov so metode z lokalno-občutljivimi funkcijami razprševanja (LSH) [9]. Za družino lokalno-občutljivih funkcij razprševanja  $H$  s parametri  $(r_1, r_2, P_1, P_2)$  velja:

$$\begin{aligned} \text{if } \|p - q\| \leq r_1 &\implies \Pr_H[h(p) = h(q)] \geq P_1 \\ \text{if } \|p - q\| \geq r_2 &\implies \Pr_H[h(p) = h(q)] \leq P_2 \end{aligned} \tag{2.7}$$

kjer sta  $p$  in  $q$  poljubna objekta,  $r_1$  in  $r_2$  dve različni razdalji,  $P_1$  in  $P_2$  verjetnosti ter  $h$  poljubna funkcija znotraj te družine. Za vsak poljuben par objektov  $p$  in  $q$  velja, da če je njuna oddaljenost  $\leq r_1$  potem mora biti verjetnost, da sta oba objekta razpršena v isto vrednost vsaj  $P_1$ . V primeru, da je njuna oddaljenost  $\geq r_2$  pa mora biti taka verjetnost največ  $P_2$ . Če za vsako funkcijo znotraj družine  $H$  držita oba pogoja, potem tako družino funkcij  $H$

imenljeno  $(r_1, r_2, P_1, P_2)$ -občutljiva. Družina takih funkcij je uporabna le, ko velja  $P_1 > P_2$  in  $r_1 < r_2$ . Pri implementaciji generiramo  $l$  tabel, vsaka izmed njih pa vsebuje  $m$  naključnih funkcij iz družine funkcij  $H$ . Te funkcije na poljuben način povežemo med seboj, tako da na koncu vsaka tabela generira eno razpršeno vrednost za podano točko. Za dovolj majhno verjetnost, da imata dve sosednji točki isto razpršeno vrednost, mora obstajati dovolj teh tabel. Poznamo različne družine lokalno-občutljivih funkcij. Med najbolj znanimi je generiranje naključnih hiperravnin [20]. Hiperravnino določa normalni vektor  $n$ , to je vektor, ki je pravokoten nanjo. Če ima skalarni produkt vektorjev  $x$  in  $n$  drugačen predznak kot skalarni produkt vektorjev  $y$  in  $n$ , potem vektorja  $x$  in  $y$  ne ležita na isti strani glede na hiperravnino, ki je definirana z vektorjem  $n$ . Na sliki 2.8 sta prikazana vektorja  $x$  in  $y$  ter hiperravnina  $h$ . Kot med vektorjema je označen s  $\theta$  in določa kosinusno razdaljo med njima. Skalarna produkta  $x \cdot n$  in  $y \cdot n$  imata različen predznak, ker sta vektorja  $x$  in  $y$  na različni strani hiperravnine. Verjetnost, da naključna hiperravnina razdeli vektorja na različni strani, je  $\frac{\theta}{180}$ . Tako lahko sestavimo družino lokalno-občutljivih funkcij za kosinusno razdaljo. Vsaka funkcija  $f \in H$  temelji na svoji hiperravnini in velja  $f(x) = f(y)$  natanko tedaj, ko imata  $x \cdot n$  in  $y \cdot n$  isti predznak. Ta družina funkcij je  $(r_1, r_2, \frac{180-r_1}{180}, \frac{180-r_2}{180})$ -občutljiva.

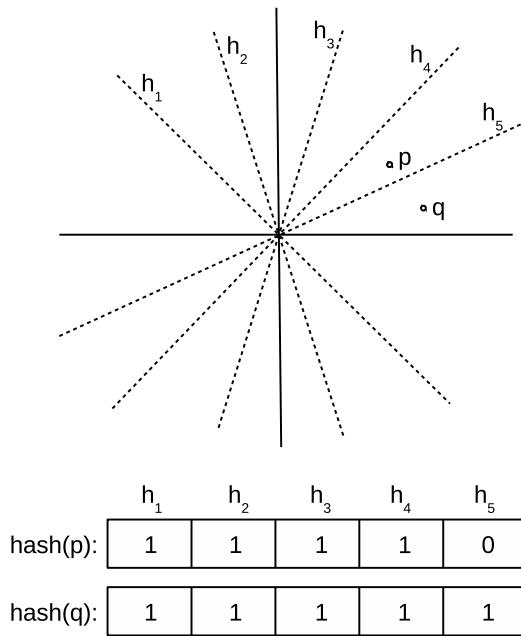


Slika 2.8: Prikaz dveh vektorjev in kota med njima ter hiperravnine.

#### 2.2.2.1 Vstavljanje

Pri vstavljanju izračunamo razpršeno vrednost točke ter vstavimo točko v koš, ki predstavlja dobljeno razpršeno vrednost. To naredimo za vsako ta-

belo in tako vstavimo točko v  $l$  košev. Vsak par tabele in razpršene vrednosti  $(t, v)$  definira svoj koš, ki vsebuje točke, ki so do bile vrednost  $v$  pri razprševanju s tabelo  $t$ . Slika 2.9 prikazuje primer vstavljanja dveh točk v posamezno tabelo, ki definira eno izmed lokalno-občutljivih funkcij družine s kosinusno razdaljo. Tabela, imenujmo jo  $t$ , je definirana s 5 notranjimi funkcijami. Vsaka izmed notranjih funkcij ima podatek o normalnem vektorju. Ta definira hiperravnino  $h_i$ , ki razdeli prostor na 2 podprostora. Razpršeno vrednost točke  $p$  dobimo tako, da ugotovimo vrednost, ki nam jo za to točko vrne posamezna funkcija. Ta je odvisna od predznaka, ki ga dobimo pri produktu  $p \cdot n$ , kjer  $n$  predstavlja izbrani normalni vektor funkcije. Rezultate funkcij združimo po vrsti in dobimo binarno razpršeno vrednost. Na koncu točko  $p$  vstavimo v koš, ki se nanaša na tabelo  $t$  in je namenjen točkam z dobljeno razpršeno vrednostjo. Uporabili smo implementacijo družine funkcij s kosinusno razdaljo iz knjižnice NearPy, ki je spisana v pythonu in vsebuje še nekaj drugih implementacij lokalno-občutljivih funkcij.



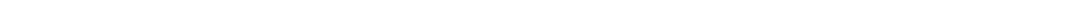
Slika 2.9: Primer vstavljanja točk v eno izmed tabel.

### 2.2.2.2 Iskanje najbližjih sosedov

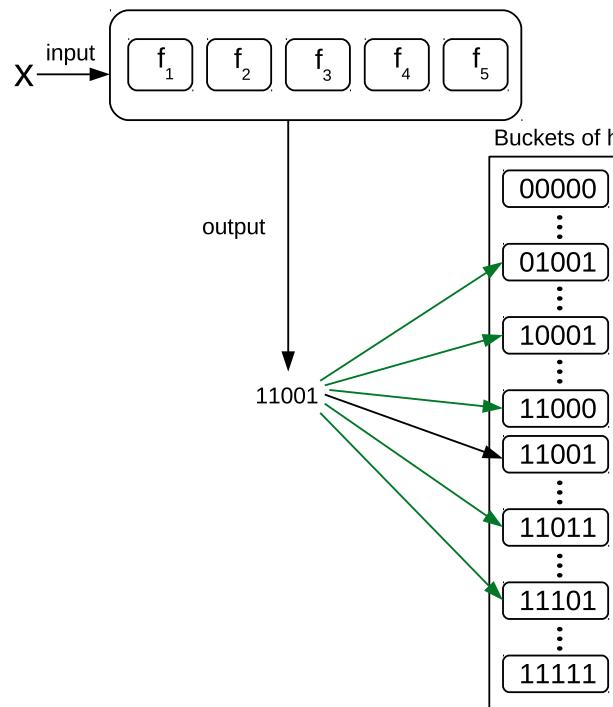
Iskanje najbližjih sosedov poteka tako, da se sprehodimo čez vse funkcije razprševanja in izračunamo razpršene vrednosti danega objekta s funkcijami. Nato se sprehodimo čez vse objekte, ki imajo enako razpršeno vrednost, izračunamo razdalje in po potrebi spremenimo seznam trenutno najbližjih sosedov. Na koncu vrnemo najbližje sosede, ki smo jih na ta način dobili. Slabost metode je, da ne zagotavlja, da bomo dejansko dobili vseh najbližjih sosedov, saj je število pregledanih točk odvisno od števila objektov, ki so imeli razpršenost v vsaj eni izmed funkcij enako razpršenosti iskanega objekta.

### 2.2.2.3 Večrazlično LSH razprševanje

Za razliko od običajnega lokalno-občutljivega razprševanja različica večrazlično razprševanje LSH (*multi-probe LSH*) algoritma [14] upošteva, da če sta si dve točki sosednji in nimata iste razpršene vrednosti, potem je visoka verjetnost, da sta si razpršeni vrednosti precej podobni. Tako se pri izgradnji indeksa poda tudi razlika, ki pove, na koliko mestih se lahko dve razpršeni vrednosti razlikujeta, da ju obravnavamo kot sosednji. S tem se zmanjša število tabel, ki jih je potrebno generirati pri izgradnji indeksa, da pridemo do dobrih rezultatov pri iskanju najbližjih sosedov. Če je podana razlika enaka 0, se za sosede štejejo le tisti z enako razpršeno vrednostjo. V tem primeru se izvede običajna implementacija lokalno-občutljivih funkcij, ki je opisana predhodno. Pri vstavljanju novega vektorja je postopek podoben kot pri običajni implementaciji. Razlika je le v tem, da moramo vektor shraniti v več košev za posamezno dobljeno razpršeno vrednost. Število teh košev je odvisno od podane razlike. Na sliki 2.10 je prikazan primer vstavljanja vektorja  $x$  v večrazlični LSH algoritem z razliko 1. Najprej se izračuna razpršena vrednost vektorja, ki je rezultat 5 notranjih funkcij. V danem primeru je razpršena vrednost vektorja  $x$  enaka 11001. Nato se vektor vstavi v koš, ki predstavlja dobljeno razpršeno vrednost (označen s črno puščico) in koše, ki so mu, glede na razpršeno vrednost, sosednji (označeni z zelenimi puščicami).



primeru so sosednji vsi koši, ki se razlikujejo v natanko enem bitu. Če bi bila razlika 2, bi bili sosednji koši tisti, ki se razlikujejo v enem ali dveh bitih. Iskanje najbližjih sosedov se od običajne implementacije razlikuje na isti način kot vstavljanje. Namesto le koša z identično razpršeno vrednostjo v trenutni tabeli je potrebno pregledati tudi vse sosednje koše dobljene razpršene vrednosti. Uporabili smo implementacijo večrazličnega LSH algoritma iz C++ knjižnice FLANN [16].



Slika 2.10: Prikaz vstavljanja vektorja pri večrazličnem LSH algoritmu z razliko 1.

### 2.2.3 Hierarhično razvrščanje z voditelji

Metoda [1] zgradi drevesno strukturo s pomočjo algoritma razvrščanje z voditelji. Ta vrne  $k$  gruč in vsako izmed točk dodeli gruči, kjer je razdalja od

centra gruče do te točke najmanjša. Postopek razvrščanja z voditelji je opisan v algoritmu 5. Najprej določimo začetne centre gruč, nato vsako točko dodelimo gruči, ki ima razdaljo med centrom gruče in točko najmanjšo. Zatem izračunamo nove centre gruč tako, da vzamemo povprečje točk, ki so v gruči. Postopek dodeljevanja točk gručam in posodobitve centrov ponavljamo, dokler prihaja do sprememb pri dodelitvi točk v gruče oziroma pri vrednostih centrov gruč. Ker je razvrščanje z voditelji hevrističen algoritem nam ne zagotavlja optimalne rešitve, saj lahko obtičimo v lokalnem minimumu. Kvaliteta rezultata algoritma je odvisna tudi od začetne izbire centrov. Običajno za začetne centre naključno izberemo  $k$  točk, vendar obstajajo tudi boljši načini izbire centrov, ki zagotavljajo boljšo rešitev in hitrejšo skonvergenco. Večina implementacij razvrščanja z voditelji omogoča omejitve števila iteracij, ki se izvedejo.

---

**Algoritem 5** K-means

---

```

1: procedure K-MEANS(points,  $k$ )
2:   clusters  $\leftarrow$  ChooseCenters(points,  $k$ )
3:   while centers have not converged do
4:     for each p in points do
5:       c  $\leftarrow$  null
6:       mindist  $\leftarrow \infty$ 
7:       for each cluster in clusters do
8:         d  $\leftarrow$  distance(p, cluster)
9:         if d  $<$  mindist then
10:           mindist  $\leftarrow$  d
11:           c  $\leftarrow$  cluster
12:         add point p to cluster c
13:       for each i = 1 ..  $k$  do
14:         clusters[i] = mean of points in cluster i
15:   return clusters

```

---

Pri hierarhičnem razvrščanju z voditelji algoritmu najprej določimo ko-

ren drevesa, ki vsebuje vse točke. Zatem z metodo razvrščanja z voditelji dobimo  $k$  gruč. Te gruče so sinovi korena drevesa. Nato postopek rekurzivno ponavljamo na sinovih, dokler je število točk v gruči večje od  $k$ . Uporabili smo implementacijo hierarhičnega k-means algoritma iz C++ knjižnice FLANN [16].

### 2.2.3.1 Iskanje najbližjih sosedov

Pri iskanju najbližjih sosedov uporabljamo kopico, v kateri hranimo poddrevesa, ki jih še nismo preiskali in njihove razdalje od centra poddrevesa do iskane točke. Na začetku kopica vsebuje koren drevesa. Algoritem sprejme število  $m$ , ki predstavlja maksimalno število točk, ki jih lahko pregledamo med iskanjem najbližjih sosedov. Če je  $m$  dovolj velik, iščemo eksaktne najbližje sosede, v nasprotnem primeru iščemo približne. Dokler kopica ni prazna in lahko pregledamo še kakšno točko, vzamemo prvi element iz kopice in iščemo sosede v njegovem poddrevesu. Pri iskanju sosedov posameznega poddrevesa najprej preverimo, ali je gruča preveč oddaljena, da bi lahko dobili bližje sosede. To najlažje storimo tako, da si za vsako gručo hranimo radij, ki predstavlja maksimalno razdaljo od centra gruče do točke, ki je vsebovana v gruči. V primeru, da je poddrevo list izračunamo razdalje vseh točk v listu in jih primerjamo s trenutnimi k-najbližjimi sosedi. V nasprotnem se premaknemo v sina, ki ima center najbližje iskani točki. Vse ostale sinove vstavimo v kopico skupaj z njihovimi razdaljami do iskane točke.

### 2.2.4 Gozd robov

Struktura gozd robov (*Boundary Forest*) [15] vsebuje  $t$  dreves robov, ki so medseboj neodvisna. Z njo se da uspešno reševati klasifikacijske in regresijske probleme ter problem iskanja najbližjega soseda. Osredotočili smo se predvsem na postopek iskanja najbližjih sosedov, ki je v nadaljevanju podrobneje razložen. Strukturo zgradimo tako, da ji podamo  $y$  točk in njihovih razredov  $c_y$ , kjer mora veljati  $y \geq t$ , saj se za koren  $i$ -tega drevesa izbere  $i$ -ta točka. Tako zgradimo  $t$  dreves, ki vsebujejo le koren. Zatem v vsako

drevo vstavimo še preostalih  $n - 1$  točk skupaj z njihovimi vrednostmi razreda in tako dobimo  $t$  dreves velikosti  $n$ . Vsako vozlišče v drevesu vsebuje podatke  $(y, c_y, children)$ , kjer  $y$  predstavlja točko vozlišča,  $c_y$  razred točke  $y$  in  $children$  seznam sinov vozlišča. Maksimalno število sinov vozlišča, označeno z  $m$ , je parameter strukture. Učenje dreves je različno v primerih reševanja klasifikacijskih, regresijskih in problemov iskanja najbližjega soseda. Pri gradnji strukture za iskanje najbližjih sosedov razredne vrednosti niso pomembne in jih nadomestijo kar sami primeri. Funkcija razdalje je parameter drevesa in ni potrebno, da ima lastnosti metrike, saj jih struktura ne uporablja. Vsa drevesa so uravnotežena in so zato primerna za hitro preiskovanje. Strukturo smo implementirali v programskejem jeziku python. Za hitrejše iskanje ter gradnjo smo vsakemu drevesu določili  $j$  dimenzij, ki jih upošteva pri računanju razdalj. Ker python ne omogoča delovanja z več nitmi in je vzporedno izvajanje metod mogoče doseči le s kreiranjem novih procesov, sočasno izvajamo le vstavljanje in iskanje več različnih primerov.

#### 2.2.4.1 Vstavljanje

Vstavljanje primera v gozd robov poteka tako, da se primer  $(y, c_y)$  vstavi v vsako drevo. Zaradi neodvisnosti dreves lahko ta proces pospešimo z istočasnim vstavljanjem primera v različna drevesa. V algoritmu 6 je prikazan algoritem učenja drevesa robov, namenjenega iskanju najbližjih sosedov, na novem primeru oziroma točki. Pri učenju drevesa za namen iskanja najbližjih sosedov vedno vstavimo primer v vsa drevesa. Vozlišče, ki mu kot sina dodamo novo vozlišče z novo točko, dobimo tako, da začnemo preiskovati pri korenju drevesa in se postopoma spuščamo po drevesu navzdol. Spustimo se v sina  $x$ , katerega razdalja  $d(x, y)$  je najmanjša izmed vseh sinov. Če ima trenutno vozlišče  $v$  manj kot  $m$  sinov, potem primerjamo tudi njegovo razdaljo  $d(v, y)$ . V primeru, da je prav njegova razdalja najmanjša, se postopek spuščanja po drevesu konča in se v vozlišče  $v$  doda nov primer.

---

**Algoritem 6** Učenje drevesa robov

---

```

1: procedure TRAIN( $y, c_y$ )
2:    $v_{min} \leftarrow \text{query}(y)$ 
3:    $\triangleright$  always add child to the closest node for knn purpose
4:    $v_{min}.add\_child(y, c_y)$ 

5: procedure QUERY( $y$ )
6:    $v \leftarrow \text{root}$ 
7:   while true do
8:      $candidates \leftarrow \text{set of the children of node } v$ 
9:     if  $\text{size}(v) < m$  then
10:      add  $v$  to  $candidates$ 
11:       $v_{min} \leftarrow \min d(x, y) \forall x \in candidates$ 
12:      if  $v_{min} = v$  then
13:        break
14:       $v \leftarrow v_{min}$ 
15:   return  $v$ 

```

---

**2.2.4.2 Iskanje najbližjih sosedov**

Pri iskanju najbližjih sosedov v gozdu robov  $k$ -najbližjih sosedov iščemo v vseh drevesih robov in nato dobljene rezultate združimo ter na ta način dobimo najbližje sosede. Pri tem moramo biti pozorni, da med združevanjem rezultatov vsak primer le enkrat upoštevamo. Preiskovanje po posameznem drevesu nam, zaradi načina iskanja, ne zagotavlja, da bomo dobili vseh  $k$  sosedov. Iskanje v drevesu robov poteka na podoben način, kot iskanje najbližje točke med vstavljanjem novega primera in je prikazano na algoritmu 7. Med iskanjem si hranimo najbližje sosede v kopici  $nns$ . Kopica  $q$  hrani podatke o vozliščih, katerih razdaljo do iskane točke smo izračunali, vendar jih še nismo obiskali. Ob koncu iteracije si izberemo naslednje vozlišče, to je vozlišče iz kopice  $q$ , ki je najmanj oddaljeno od iskane točke. Če je kopica  $q$  prazna ali če je razdalja vozlišča iz kopice do iskane točke večja, kot razdalja do  $k$ -tega najbližnjega sosedja, se iskanje zaključi. Zatem je potrebno najbližje

sosede iz vseh dreves združiti.

---

**Algoritem 7** Iskanje k-najbližjih sosedov v drevesu robov

---

```

1: procedure KNN_SEARCH( $y, k$ )
2:    $\triangleright nns$  is a heap queue containing  $k$  nearest neighbours as pairs
      (distance, neighbour),  $q$  is a heap queue containing candidate nodes
      as pairs (distance, node)
3:    $nns \leftarrow [(-\infty, \_)$  for  $i = 1 \dots k - 1]$ 
4:    $v \leftarrow$  root
5:    $nns.push((-distance(v.point, y), v.point))$ 
6:    $q \leftarrow []$ 
7:   while true do
8:     for each  $child$  in  $v.children$  do
9:        $cur_d \leftarrow dist(child.point, y)$ 
10:       $q.push((cur_d, child))$ 
11:      if  $cur_d < kth\_NN$  then
12:         $nns.pop()$ 
13:         $nns.push((-cur_d, child.point))$ 
14:      if  $q$  is empty or  $q.first.dist > kth\_NN$  then
15:        break
16:       $v \leftarrow q.first.node$ 
17:       $q.pop()$ 
18:   return  $nns$ 

```

---

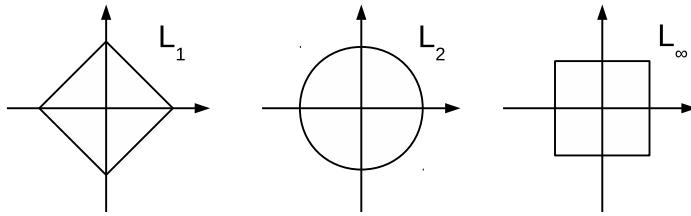
# Poglavlje 3

## Evalvacija

V tem poglavju predstavimo podatke, na katerih smo testirali iskanje najbližjih sosedov. Ti so razdeljeni v 2 skupini, na nizkodimenzionalne in visokodimenzionalne podatke. Prve smo uporabili za testiranje eksaktnih metod in druge za testiranje približnih. Najprej predstavimo mero za gostoto podatkovne množice. Na koncu poglavja razložimo oznake metod in njihov pomen.

### 3.1 Merjenje razpršenosti

Razpršenost podatkovne množice smo merili z algoritmom LiNearN [21]. Algoritmom nam za podano točko in podatkovno množico poda oceno gostote točk v okolini podane točke. Za razdaljo se lahko izbere poljubna  $L_p$ -norma, v kateri je razdalja definirana, kot je prikazano v enačbi 3.1, kjer  $x$  in  $y$  predstavlja točki,  $d$  pa njuno dimenzijo. Izbor norme definira obliko lika s točkami, ki so enako oddaljene. Slika 3.1 prikazuje oblike treh najbolj znanih  $L_p$ -norm. V algoritmu 9 je prikazan LiNearN algoritmom, ki uporablja  $L_\infty$ -normo. Izberemo si število iteracij  $t$ , število hiperkock  $a$  in število točk  $b$ , ki jih bomo upoštevali pri ocenjevanju gostote. Z večanjem vrednosti parametrov se ocena gostote izboljša, vendar se izvajanje algoritma upočasni. V vsaki iteraciji izberemo podmnožico  $D \subset data$ , kjer  $data$  predstavlja množico



Slika 3.1: Prikaz oblik likov, ki jih tvorijo enako oddaljene točke v dvodimensionalnem prostoru za  $L_1$ ,  $L_2$  in  $L_\infty$ .

točk velikosti  $a$ , na podlagi katere zgradimo hiperkocke za to iteracijo. Vsaka točka  $x \in D$  definira svojo hiperkocco s centrom v tej točki, maso enako 0 in radijem, ki je enak  $\frac{1}{2} \cdot \|x - nn\|_\infty$ , kjer  $nn$  predstavlja njen najbližjo sosednjo točko v množici točk  $D \setminus x$ . Masa hiperkocke je definirana kot število točk, ki so vsebovane hiperkocki.

$$\|x - y\|_p = \sqrt[p]{\sum_{i=1}^d |x_i - y_i|^p} \quad (3.1)$$

$$\|x - y\|_\infty = \max(|x_1 - y_1|, |x_2 - y_2|, \dots, |x_d - y_d|)$$

Za hiperkocke v posameznih iteracijah ocenimo gostoto tako, da znova izberemo  $D \subset data$  in za  $\forall x \in D$  dobimo hiperkocco, ki vsebuje  $x$  in ji maso povečamo za 1. Če nobena hiperkocka ne vsebuje točke  $x$ , potem ta točka ne spremeni mase nobeni hiperkocki. Izračun gostote točk okrog točke  $x$  z uporabo LiNearN algoritma je prikazan v algoritmu 8. Ta za iskanje najbližjega soseda uporabi izčrpno iskanje zaradi majhnega števila točk množice, v kateri išče. Algoritem vrne povprečno gostoto  $t$  hiperkock, ki vsebujejo  $x$ , kjer gostoto hiperkocke predstavlja razmerje med maso in radijem hiperkocke. Gostoto celotne podatkovne množice  $data$  ocenimo tako, da povprečimo gostoto posameznih točk v tej množici, kot prikazuje enačba 3.2, kjer je  $hs$

rezultat funkcije LiNearN na podatkovni množici  $data$ .

$$density(data) = \frac{\sum_{x \in data} density(x, hs)}{length\ of\ data} \quad (3.2)$$

Množice smo najprej normalizirali tako, da smo poiskali minimalno in maksimalno vrednost v vsaki dimenziji  $min$  in  $max$  in nato spremenili vrednost  $x$  v  $x = \frac{x-min}{max-min}$ . S tem smo vse dimenzije postavili na interval  $[0,1]$ , saj je pomembno, da imajo množice podatkov podoben razpon v vseh dimenzijah, če želimo primerjati njihovo gostoto.

---

**Algoritem 8** Ocenjevanje gostote okrog točke  $x$  z LiNearN algoritmom

---

```

1: procedure DENSITY( $x, hs$ )
2:    $\triangleright x$  - data point,  $hs$  - output of LiNearN algorithm
3:    $p \leftarrow 0$ 
4:   for each hypercubes in  $hs$  do
5:      $h \leftarrow$  get hypercube from hypercubes that covers point  $x$ 
6:     if  $h$  is not null then
7:        $p \leftarrow p + \frac{h.mass}{h.radius}$ 
8:   return  $\frac{p}{length\ of\ hs}$ 

```

---

---

**Algoritem 9** LiNearN algoritem za  $L_\infty$ 

---

```

1: procedure LINEARN(data, t, a, b)
2:    $\triangleright$  data - data points, t - number of subsamples, a - number of
   hypercubes, b - size of subsample used to estimate density
3:   hs  $\leftarrow$  empty list
4:   for each i in 1..t do
5:     D  $\leftarrow$  choose a points from data without replacement
6:     hypercubes  $\leftarrow$  BUILD_HYPERCUBES(D)
7:     insert hypercubes in hs[i - 1]
8:   ASSIGN_SAMPLE_MASS(hs, data, b)
9:   return hs
10: procedure BUILD_HYPERCUBES(D)
11:   hypercubes  $\leftarrow$  empty list
12:   for each x in D do
13:     create new Hypercube h with center at x
14:      $\triangleright$  find nearest neighbour of x using brute-force algorithm
15:     nn  $\leftarrow$  nearest neighbour of x in D \ x using  $L_\infty$ 
16:     h.radius  $\leftarrow$   $\frac{1}{2} \cdot \|x - nn\|_\infty$ 
17:     h.mass  $\leftarrow$  0
18:     insert hypercubes in hs[i]
19:     insert h in hypercubes
20:   return hypercubes
21: procedure ASSIGN_SAMPLE_MASS(hs, data, b)
22:   hypercubes  $\leftarrow$  empty list
23:   for each hypercubes in hs do
24:     D  $\leftarrow$  choose b points from data without replacement
25:     for each i in 1..b do
26:       h  $\leftarrow$  get hypercube from hypercubes that covers point Di
27:       if h is not null then
28:         increase mass of h by 1

```

---

## 3.2 Opis podatkovnih množic

Za testiranje algoritmov smo uporabili testne množice z različnim številom dimenzij. Nekatere smo pridobili iz zbirke podatkovnih množic za strojno učenje UCI [12], večino ostalih smo generirali sami, da smo tako dobili različne lastnosti (npr. razpršenost). Število podatkovnih množic, ki smo jih uporabili za testiranje, je 15. Med njimi je 10 takih, ki imajo število dimenzij manjše od 10 ter 5 z večjim številom dimenzij. Med tistimi z nizkimi dimenzijami prevladujejo 2 in 3 dimenzionalni podatki, ki so primerni za primerjavo eksaktnih metod. Ker določene strukture ne dovolijo vnosa dveh identičnih točk, smo take odstranili iz množic.

### 3.2.1 Nizkodimensonalne podatkovne množice

Ker smo želeli ugotoviti, kako razpršenost podatkov vpliva na delovanje posameznih eksaktnih algoritmov smo generirali 3 vrste razpršenosti in sicer v krogu, v elipsi in v gručah. Nizkodimensonalne množice podatkov, na katerih smo izvajali teste, so:

- nemške poštne številke (NPŠ): podatki predstavljajo lokacije poštnih številk v Nemčiji,
- krog2d: podatki so enakomerno razpršeni v obliki kroga,
- krog3d: podatki so razpršeni z enakomerno porazdelitvijo v obliki krogla,
- krog5d: podatki so enakomerno razpršeni v 5-dimensionalni hiperkrogli,
- krog10d: podatki so enakomerno razpršeni v 10-dimensionalni hiperkrogli,
- gruče2d: podatki so generirani z normalno porazdelitvijo okrog 10 naključnih centrov, kjer je standardni odklon 1 pri podatkih na intervalu

[-15, 15]

- gruče3d: podatki so generirani okrog 10 naključnih centrov, kjer je standardni odklon 1 pri podatkih na intervalu [-15, 15],
- elipsa2d: podatki so v obliki elipse,
- elipsa3d: podatki so v obliki elipse v 3 dimenzionalnem prostoru in
- uniform1d: podatki so uniformno porazdeljeni na intervalu [-1.000.000, 1.000.000].

### 3.2.2 Visokodimenzionalne podatkovne množice

Pri visokodimenzionalnih podatkih smo 2 množici dobili iz zbirke množic UCI (amazon in gisette), 3 so bile že v preteklosti uporabljene za primerjanje približnih najbližjih sosedov (aerial, disk trace in wiki) [13], eno pa smo generirali sami (dim1000). Te množice so:

- aerial: podatki predstavljajo teksture zračnih posnetkov,
- wiki: podatki predstavljajo wiki dokumente, kjer je vsak predstavljen s 500 dimenzijami,
- dim1000: podatki so razpršeni z normalno porazdelitvijo na intervalu celih števil [0, 255],
- disk trace: podatki predstavljajo zapise na disk računalnika; zapisi so velikosti 1kB,
- gisette: podatek predstavlja opis slike, na kateri je ročno napisana številka 4 ali 9 in
- amazon: podatek predstavlja informacije o komentarju nekega uporabnika.

### 3.3 Gostota podatkovnih množic

Razpršenost podatkovnih množic smo merili z LiNearN algoritmom. Za parametre smo izbrali  $t = 50, a = 4, b = 64$  in  $L_\infty$ -normo. Iz tabele 3.1 je razvidno, da imajo podatkovne množice večjih dimenzij manjšo povprečno gostoto okrog točk, kar je pričakovano zaradi večjega prostora. Pri podatkovnih množicah gisette in amazon je gostota zelo nizka zaradi velikega prostora in majhnega števila točk.

Množica	N	D	Povprečna G	Standardni odklon G
aerial	275.465	60	731.49	837.81
amazon	1.480	10.000	0.01	0.14
krog10d	100.000	10	3.35	6.63
krog2d	100.000	2	1660.29	963.26
krog3d	100.000	3	513.53	314.10
krog5d	100.000	5	66.51	74.73
gruče2d	100.000	2	1483.42	690.01
gruče3d	100.000	3	801.31	297.65
dim1000	10.000	1.000	0.00	0.02
disk trace	30.840	1.024	104.14	150.88
elipsa2d	100.000	2	1653.64	1015.36
elipsa3d	100.000	3	732.43	515.91
NPŠ	7.955	2	1000.48	478.36
gisette	6.000	5.000	0.00	0.03
uniform1d	100.000	1	3001.28	789.53
wiki	100.000	500	88.02	169.70

Tabela 3.1: Prikaz lastnosti podatkovnih množic in njihove gostote, izračunane z LiNearN pri  $t = 50, a = 4, b = 64$  in  $L_\infty$ . G predstavlja gostoto, N število točk in D njihovo dimenzionalnost.

### 3.4 Testni scenarij

Opisan testni scenarij smo uporabili pri testiranju hitrosti iskanja najbližjih sosedov v razdelku 4.3. Števila najbližjih sosedov, ki smo jih iskali so bila na intervalih  $[1, 10]$  in  $[12, 50]$ , pri čemer smo zaradi hitrejšega testiranja na intervalu  $[12, 50]$  vzeli samo soda števila. Vsi testi so uporabljali evklidsko razdaljo. Čas izgradnje strukture smo omejili na 20 minut. Pri eksaktnih metodah smo najbliže sosede testirali na 100 naključnih točkah iz množic podatkov, pri približnih metodah pa smo zaradi počasnejšega testiranja najbliže sosede testirali na 50 naključnih točkah. Zaradi precejšnje razlike v hitrosti izvajanja implementacij v programskejem jeziku C++ v primerjavi z implementacijami v programskejem jeziku python smo implementacije primerjali ločeno glede na programski jezik. Ta primerjava nam omogoča, da na rezultate ne vpliva implementacija, ampak kakovost in primernost podatkovne strukture. Testi so potekali na 4-jedrnem procesorju Intel Xeon X3460 2.80GHz. Zaradi odvisnosti struktur od parametrov smo testirali strukture z različnimi parametri. Zaradi napake pri vezavi knjižnice FLANN za programski jezik python nam ni uspelo testirati te LSH implementacije. Zgrajene strukture smo shranili v datoteke, da nam jih v prihodnje ni bilo potrebno ponovno graditi. Žal določene strukture nimajo implementirane možnosti shranjevanja ali imajo druge težave. FLANN implementacije imajo napako pri shranjevanju, saj nam nalaganje shranjene strukture ne vrne iste strukture, gozd robov pa ima težave pri shranjevanju zaradi omejitve števila rekurzivnih klicev, ki je del programskega jezika python. Zaradi omenjenih težav je bilo testiranje počasnejše in smo zato vzporedna testiranja vseh iskanih točk izvedli samo enkrat.

Med eksaktnimi metodami smo uporabili strukture KD-drevo, R-drevo, R\*-drevo in PM-drevo. Implementaciji KD-drevesa smo uporabili iz knjižnic scikit-learn in FLANN, ostale strukture smo sami implementirali v programskejem jeziku python. Pri R in R\*-drevesih smo vedno uporabili kvadratno razdelitev, oznake struktur so oblike R-tree\_x, kjer x predstavlja maksimalno

število sinov vozlišč. Pri PM-drevesih imajo imena struktur obliko PM-tree- $x\_nhr\_npd$ , kjer  $x$  predstavlja maksimalno število sinov vozlišč,  $nhr$  predstavlja število pivotov, ki se upoštevajo v notranjih vozliščih in  $npd$  število pivotov, ki se upoštevajo v listih. Pri vseh PM-drevesih smo generirali 10 naključnih skupin pivotov, med katerimi smo izbrali tisto, ki je imela vsoto razdalj med pivoti največjo. Vse implementacije iz knjižnice FLANN imajo na koncu imen še število uporabljenih jeder za iskanje najbližjih sosedov.

Med približnimi metodami smo uporabili RKD-drevo, hierarhični k-means, FlannAuto, LSH in Gozd robov. Implementacije RKD-drevo, k-means in FlannAuto smo uporabili iz knjižnice FLANN. Vse tri metode smo testirali s privzetimi parametri, razen s parametrom željene točnosti  $\epsilon$ . Metode smo poimenovali RKD- $x$ , K-means- $x$  in Flann- $x$ , kjer je  $x$  željena točnost. FlannAuto poskuša s prečnim preverjanjem ugotoviti najprimernejši algoritmom in parametre za podano množico podatkov. Pri tem izbira med strukturami RKD-drevo, k-means in kombinacijo obeh. Implementacijo RKD-drevesa smo uporabili tudi iz C++ knjižnice Annoy. Strukture smo poimenovali v obliki Annoy- $x$ , pri čemer  $x$  predstavlja število naključnih KD-dreves. Implementacijo lokalno-občutljivih funkcij razprševanja smo uporabili iz python knjižnice NearPy. Testirali smo z različnim številom naključnih binarnih projekcij in smo zato algoritmom dali imena oblike LSH- $x$ , kjer  $x$  predstavlja število binarnih projekcij. Ker smo gozd robov implementirali sami, smo uporabili našo implementacijo. Pri tem smo število sinov vozlišč omejili na 10 in strukturam dali imena oblike BF- $x\_y$ , kjer  $x$  predstavlja število generiranih dreves,  $y$  pa število naključnih dimenzij, ki jih posamezno drevo upošteva.

Testiranje parametrov je potekalo na isti način, vendar smo strukturo R-drevo poimenovali drugače, ker smo primerjali tudi način razdelitve. R-tree-q- $X$  predstavlja R-drevo s kvadratno razdelitvijo pri maksimalni velikosti vozlišča  $X$ , R-tree\_l- $X$  pa isto drevo z linearno razdelitvijo.



# Poglavlje 4

## Rezultati testov

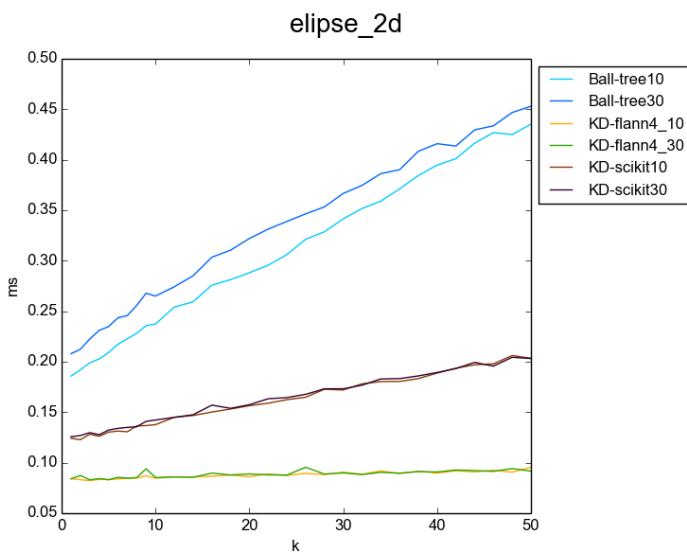
Pri testiranju hitrosti iskanja najbližjih sosedov predstavimo rezultate na podatkovnih množicah, ki so tipične za iskanje z neko podatkovno strukturo. Poleg tipičnih prikažemo tudi grafe, ki odstopajo in razložimo razloge za odstopanja. Analiziramo tudi porabo pomnilnika posameznih eksaktnih in približnih metod in rezultate glede na gostoto podatkovnih množic ocenjenih z LiNearN algoritmom. Najprej analiziramo posamezne parametre v 4.1, nato porabo pomnilnika v 4.2 in na koncu še hitrost iskanja najbližjih sosedov v 4.3.

### 4.1 Analiza parametrov podatkovnih struktur

Navajamo najpomembnejše ugotovitve o vplivu parametrov na hitrost iskanja. Za vsako družino metod prikažemo graf, ki prikazuje tipično obnašanje struktur pri danih parametrih. Nekatere strukture nimajo pomembnih parametrov, oziroma so ti predvidljivi, kot npr. število dreves pri metodi Annoy, ki z naraščanjem veča točnost in upočasni iskanje. Večina grafov prikazuje obnašanje za različno število iskanih sosedov ( $k$ ), ki je podano na abscisni osi.

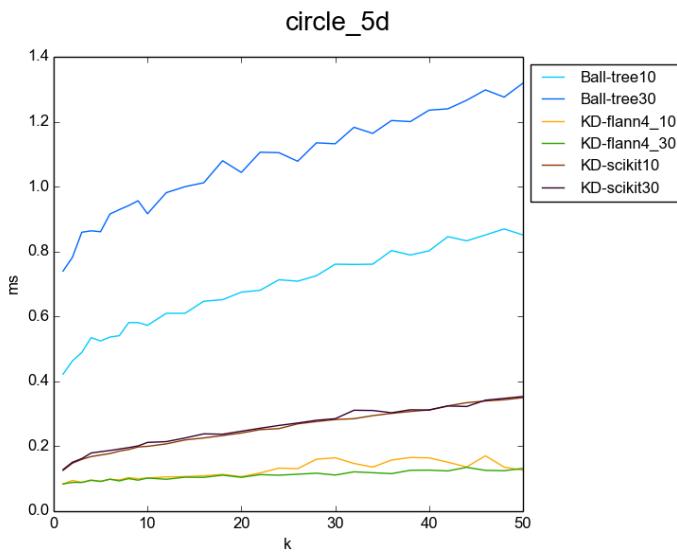
### 4.1.1 KD-drevo in ball-drevo

Obe implementaciji KD-drevesa in implementacijo ball-drevesa testiramo z istim parametrom in sicer maksimalnim številom objektov v listih. Z večanjem tega parametra se zmanjšuje višina drevesa in veča velikost listov. Na sliki 4.1 opazimo, da sprememba tega parametra pri nobeni od implementacij ne vpliva bistveno na hitrost iskanja najbližjih sosedov na podatkovni množici elipsa2d.



Slika 4.1: Rezultati časovnih meritev KD in ball dreves na podatkovni množici elipsa2d pri spremenjanju števila iskanih elementov k. Krivulje se razlikujejo glede števila elementov v listih.

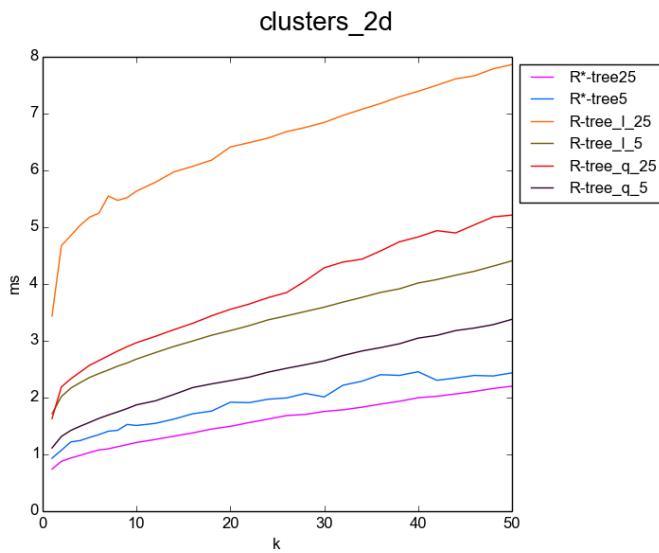
Opazili smo, da je pri večjih dimenzijah podatkov ball-drevo z manjšim parametrom precej hitrejše pri iskanju, kar je razvidno iz slike 4.2. Podobne rezultate smo dobili tudi na drugih množicah in jih navajamo v dodatku na slikah od A.1 do A.8.



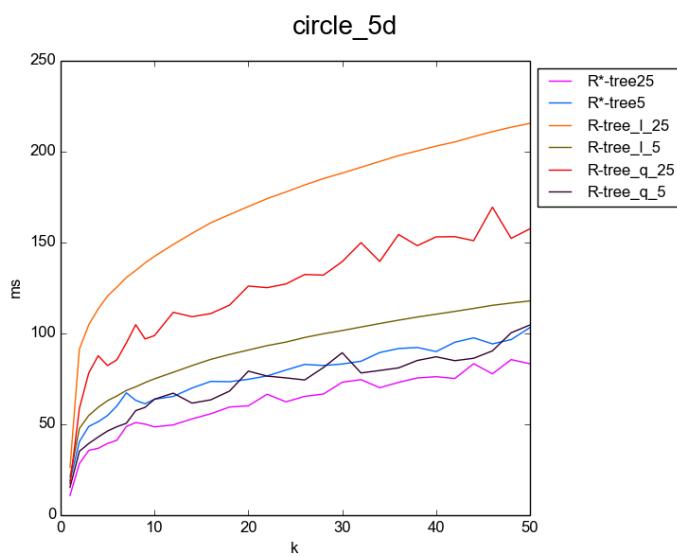
Slika 4.2: Rezultati časovnih meritev KD in ball dreves na podatkovni množici krog5d v odvisnosti od števila iskanih sosedov. Krivulje se razlikujejo glede števila elementov v listih.

### 4.1.2 R in R\*-drevo

Pri R-drevesu smo primerjali, kako močno tip delitve vozlišč vpliva na kakovost strukture. Pri tem smo testirali linearne in kvadratne razdelitev. Testirali smo maksimalno število sinov, ki jih lahko imajo vozlišča v drevesu. Ta parameter smo testirali tudi pri R\*-drevesih. Pri R-drevesih nam povečanje velikosti vozlišč poslabša rezultate, medtem ko nam isti parameter pri R\*-drevesih rezultate malenkost izboljša, kar je razvidno iz slike 4.3, ki prikazuje testiranje na podatkovni množici gruče2d. To je najverjetnejše posledica tega, da pride pri R-drevesu z večjo velikostjo vozlišč do več pravokotnikov, ki niso kvadratne oblike, kar povzroči pregledovanje večih vozlišč. R\*-drevo ta problem rešuje z boljšim vnosom, ki povzroči le majhno prekrivanje listov drevesa in z razdelitvijo, ki se trudi za čim manjše prekrivanje in izberi dimenzije razdelitve, ki upošteva obseg pravokotnikov.



Slika 4.3: Rezultati časovnih meritev R in R\* dreves na podatkovni množici gruče2d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.

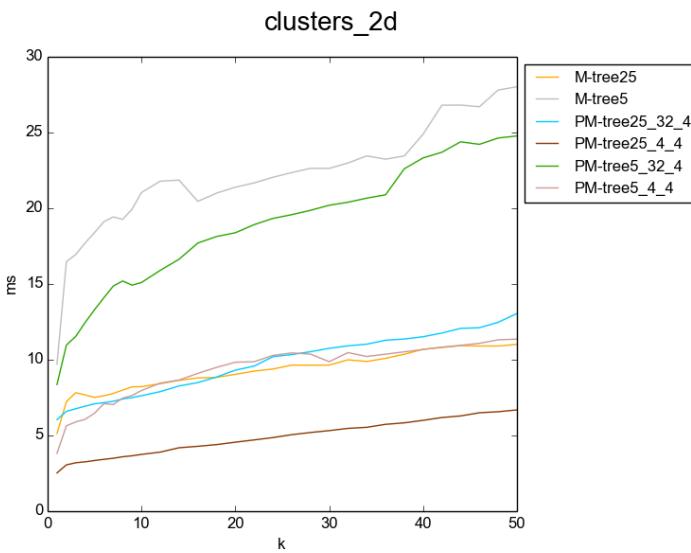


Slika 4.4: Rezultati časovnih meritev R in R\* dreves na podatkovni množici krog5d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.

Opazimo, da R-drevo z linearno razdelitvijo daje slabše rezultate od R-drevesa s kvadratno razdelitvijo. Z večanjem dimenzij podatkov postanejo strukture manj učinkovite in ponekod se vrstni red kvalitete struktur spremeni. Tak primer je prikazan na sliki 4.4. Ostali grafi so prikazani v dodatku na slikah od A.9 do A.16.

### 4.1.3 PM-drevo

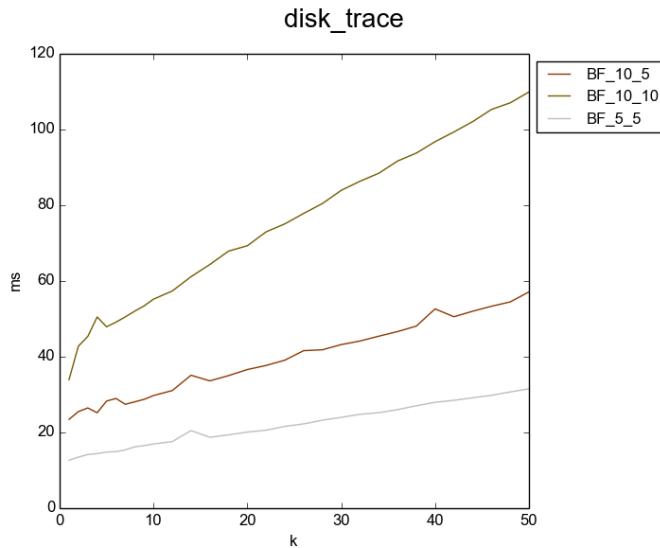
Pri PM-drevesih smo analizirali parametre maksimalno število sinov vozlišč, število pivotov, ki se upoštevajo v vozliščih in število pivotov, ki se upoštevajo v listih. Iz slike 4.5 vidimo, da so pomembni vsi parametri. Najpomembnejši je maksimalno število sinov v vozliščih, saj nam povečanje tega parametra močno zmanjša čas iskanja. Za razliko od R-dreves, kjer lahko dobimo podolgovate pravokotnike, pri PM-drevesu tega problema ni, saj je prostor definiran s krogom. V splošnem ima PM-drevo večje težave pri majhni velikosti vozlišč, ker pri vsebovanosti dveh oddaljenih točk v istem vozlišču njegov krog pokrije precej več prostora, kot bi ga pravokotnik v R-drevesu. Tudi število upoštevanih pivotov v vozliščih precej vpliva na hitrost iskanja. Če jih je malo, se hitrost iskanja poveča, če pa jih je več, se iskanje upočasni zaradi povečanja števila preverjanj vsebovanosti točke v obročih. Ostali rezultati so prikazani v dodatku na slikah od A.17 do A.25.



Slika 4.5: Rezultati časovnih meritev PM dreves na podatkovni množici gruče2d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.

#### 4.1.4 Gozd robov

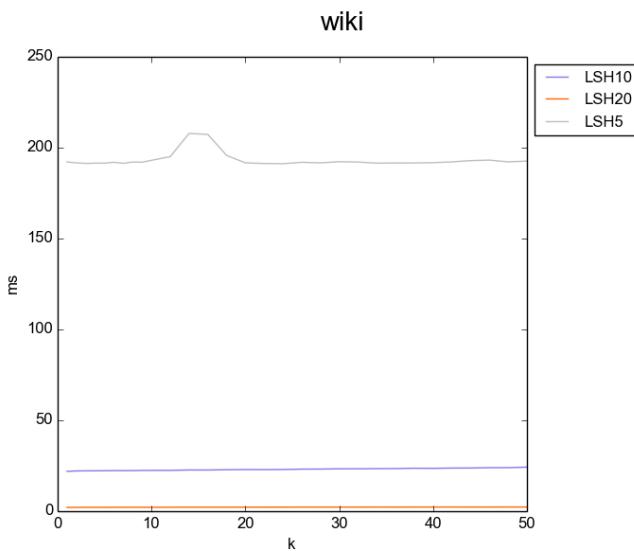
Pri gozdu robov smo opazovali, kako na hitrost vpliva število dreves v gozdu in število naključnih dimenzij, ki se upoštevajo za računanje razdalje v drevesih. Pričakovano se z večanjem števila upoštevanih dimenzij veča čas iskanja. Isto velja tudi za število dreves, kar je razvidno iz slike 4.6. Ker naša implementacija ne uporablja več jeder, se čas potreben za iskanje linearno povečuje z višanjem obeh parametrov. Rezultatov testiranj na množici aerial ni vključenih, ker se strukture niso zgradile v zahtevanem času. Ostali grafi so vključeni v dodatku na slikah od A.26 do A.29. Iz slike A.28 vidimo, da je iskanje pri upoštevanju desetih dimenzij hitrejše, kot pri upoštevanju petih. Iskanje najbližjih točk se je pri prvem zaključilo na višjem nivoju in zato ni bilo potrebnih toliko računanj, razlog za to je naključje.



Slika 4.6: Rezultati časovnih meritev gozda robov na podatkovni množici disk trace v odvisnosti od števila iskanih sosedov, dreves in upoštevanih dimenzij.

#### 4.1.5 LSH

Pri lokalno-občutljivih funkcijah razprševanja smo analizirali število naključnih binarnih projekcij. Če je ta parameter premajhen, koši vsebujejo veliko število točk in jih zato pregledamo več, kar upočasni iskanje. Če je število projekcij veliko se lahko zgodi, da je koš prazen in ne najdemo nobenega soseda. Število naključnih binarnih projekcij je zato pomemben parameter in je odvisen od podatkovne množice, gostote in števila točk. Običajno želimo pri iskanju  $k$  sosedov imeti koše velikosti  $k$ , saj je tako iskanje hitro in dobimo vseh  $k$  sosedov. Na sliki 4.7 vidimo, da 5 binarnih projekcij ni dovolj, ker so koši preveliki. Hitrost iskanja sosedov v LSH strukturi ni odvisna od števila sosedov, ki jih iščemo (vsaj za zmerno število sosedov), saj vedno preiščemo celoten koš, v katerega se uvrsti iskana točka. Ostali rezultati so prikazani na slikah od A.30 do A.34.



Slika 4.7: Rezultati časovnih meritev LSH na podatkovni množici wiki v odvisnosti od števila iskanih sosedov in binarnih projekcij.

## 4.2 Ocene porabe pomnilnika

S pomočjo python knjižnice *memory profiler* smo ocenili pomnilniško zahetvost struktur. Pri strukturah smo uporabili različne parametre, če smo menili, da vplivajo na rabo pomnilnika. Pri eksaktnih metodah smo izpustili nekatere podatkovne množice, saj bi bila raba pomnilnika podobna testiranim. Iz tabele 4.1 je razvidno, da gozd robov pri 10 naključnih dimenzijah porabi največ pomnilnika. Razlog je, da si vsako drevo shrani celotno tabelo z dimenzijami, ki so bile izbrane za to drevo. To je posledica naše implementacije, ker nismo želeli krčiti točk med iskanjem sosedov, saj bi to upočasnilo iskanje. Pričakovano najmanj pomnilnika porabita LSH metodi, saj si hranita le podatke o posameznih koših. V tabeli 4.1 KM predstavlja k-means strukturo. Ker se BF5 in BF10 na podatkovni množici aerial nista zgradili v času 20 minut, njune porabe nismo izmerili. Tabela 4.2 prikazuje rabo pomnilnika eksaktnih struktur. Metode M, PM5 in PM32 imajo maksimalno

Množica	Annøy1	Annøy	RKD	KM	LSH5	LSH10	BF5	BF10
aerial	128.1	259.2	272.8	31.9	55.5	55.4	/	/
amazon	78.8	313.2	1.4	29.3	0.5	0.9	61.9	65.7
dim1000	63.3	126.7	10.1	11.3	2.0	2.3	43.1	71.3
disk trace	128.6	257.4	30.8	61.9	6.2	6.9	125.7	227.4
gisette	157.0	313.4	6.0	30.9	1.3	1.7	43.1	71.3
dim1000	63.3	126.7	10.1	11.3	2.0	2.3	134.3	154.1
wiki	253.5	505.5	99.5	75.2	20.0	22.0	514.4	811.0

Tabela 4.1: Prikaz rabe pomnilnika v MB nekaterih struktur za iskanje pri- bližnjih sosedov.

Množica	Ball	KDS	KDF	R5	R*5	M	PM4_4	PM32_4
krog2d	1.3	1.2	4.1	156.2	156.1	123.9	161.4	365.9
krog3d	1.3	1.2	4.9	159.7	158.5	127.0	165.0	376.5
krog10d	1.8	1.7	10.5	209.9	211.8	131.1	171.1	397.5
NPŠ	0.2	0.3	0.5	12.6	12.6	10.3	13.2	30.6
uniform1d	1.2	1.1	3.4	152.1	152.8	119.5	155.2	348.2

Tabela 4.2: Prikaz rabe pomnilnika v MB nekaterih struktur za iskanje eksaktnih sosedov.

število sinov v vozlišču omejeno na 5, PM5 in PM32 predstavlja strukturi PM-tree\_5\_5\_4 in PM-tree\_5\_32\_4. R-drevesa smo testirali z linearno razdelitvijo in omejitvijo na maksimalno 5 sinov v vozlišču, ista omejitev je veljala pri R\*-drevesu. KDS predstavlja KD-drevo iz scikit-learn knjižnice, KDF pa iz knjižnice FLANN. R in PM drevesa zasedajo veliko več pomnilnika, kot ostala. Razlog je v tem, da smo te strukture implementirali tako, da hranijo celotne točke in ne le njihovih indeksov. S tem dovolimo uporabniku spremenjati podatkovno množico brez vpliva na že zgrajeno strukturo, saj ne hranimo referenc na točke. PM-drevesa s pivoti zasedejo več prostora, ker vsako vozlišče vsebuje dodatne informacije o oddaljenosti centra vozlišča do vseh pivotov.

## 4.3 Hitrost iskanja najbližjih sosedov

V tem razdelku primerjamo hitrost iskanja najbližjih sosedov eksaktnih in približnih metod. Pri približnih metodah merimo še točnost iskanja, delež vrnjenih objektov in izrabo večjedrnih procesorjev. Zaradi razlike v hitrosti izvajanja primerjamo C++ in python implementacije posebej.

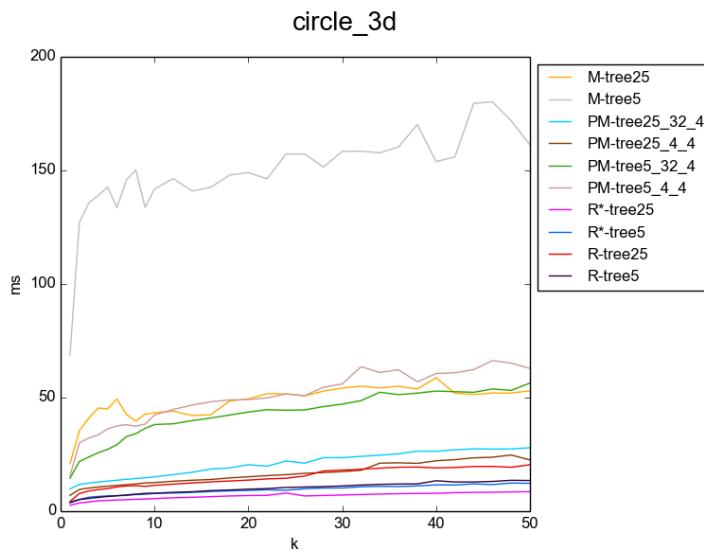
### 4.3.1 Eksaktne metode

Pri eksaktnih metodah nas je zanimal čas iskanja najbližjih sosedov. Ker smo sami implementirali R, R\* in PM-drevo smo lahko primerjali tudi število izračunanih razdalj in število objektov, ki jih moramo pregledati v listih.

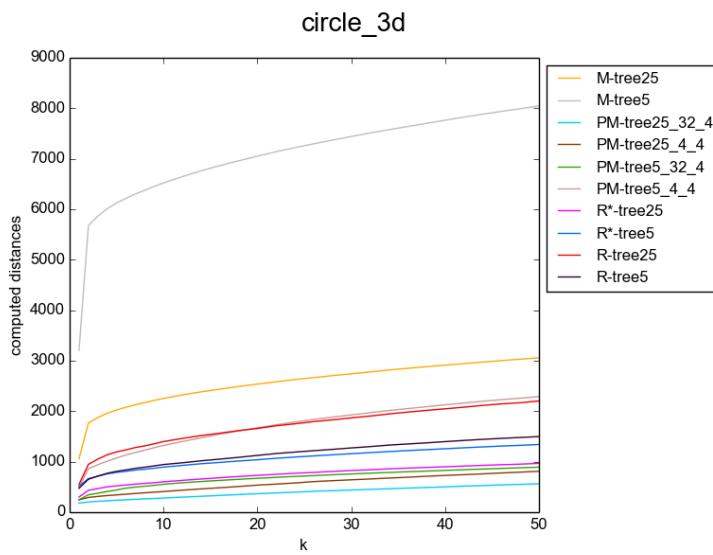
#### 4.3.1.1 Python implementacije

Iz slike 4.8 vidimo, kako pomembni so parametri struktur, posebno pri M-drevesu. Zanimivo je, da će M-drevesu, kjer ima vozlišče maksimalno 5 sinov, dodamo le 4 pivote, se čas iskanja zmanjša za več kot 50 odstotkov. Iz grafa je razvidno, da se z večanjem velikosti vozlišč PM-drevo izboljša, kar velja le do neke meje, ki pa je nismo iskali zaradi počasnosti testiranja.

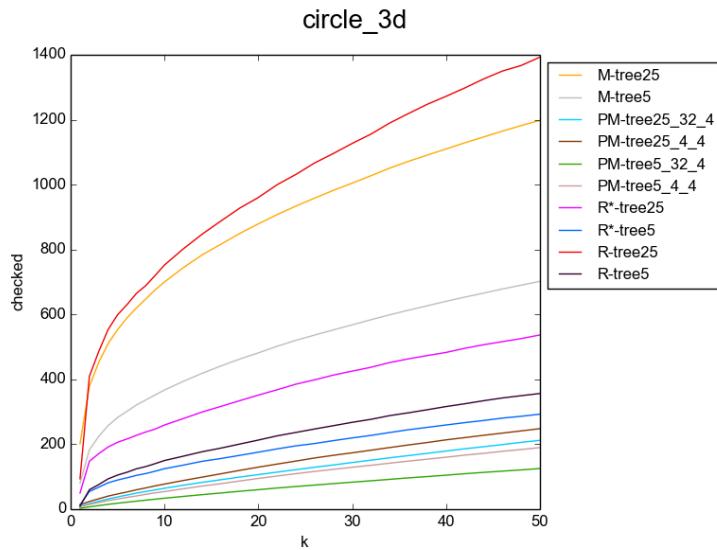
Najpočasnejša metoda je PM-drevo brez pivotov in z omejitvijo števila sinov v vozliščih na 5, najhitrejša pa R\*-drevo z omejitvijo števila sinov v vozliščih na 25. To je razvidno tudi iz testiranj na preostalih množicah, ki so prikazane v dodatku na slikah od B.1 do B.25. Razvrstitev metod v preostalih testiranjih je bila podobna. PM-tree25\_32\_4 manjkrat izračuna razdaljo med dvema točkama, kot PM-tree25\_4\_4 (slika 4.9). Kljub temu je PM-tree25\_32\_4 počasnejši v iskanju. Razlog za to je, da je časovno dodatno preverjanje vsebovanosti v hiperobročih bolj potratno, kot dodatno računanje razdalj. Zato je pomembno, da izberemo optimalno število pivotov. Čeprav je število izračunanih razdalj pri PM-tree5\_32\_4 podobno kot pri PM-tree25\_32\_4 je razlika v času iskanja precejšnja. Iskanje je počasnejše, ker pri PM-tree25\_32\_4 damo v kopico skupno precej manj vozlišč. Pri PM-tree5\_32\_4 nam ta dodatna vozlišča ne povzročijo veliko dodatnih računanj razdalj, ker se pri večini ugotovi nevsebovanost v zahtevanih prostorih (hiperobročih in hiperkrogli). Zaradi večjega prekrivanja prostorov vozlišč PM-dreves v primerjavi z R in R\*-drevesi so, kljub večjemu številu izračunanih razdalj, R in R\*-drevesa hitrejša. Na sliki 4.10 vidimo, da PM-drevesa v povprečju pregledajo najmanj točk v listih dreves. To bi bilo pomembno, če bi bilo branje objektov v listih počasnejše (npr. branje iz podatkovne baze) in bi objekte, ki so centri vozlišč ali pivoti, hranili v strukturi. Python implementacije se prav tako ne izkažejo za dobre v 10 dimenzionalnem prostoru, kar prikazuje slika 4.11, saj npr. R\*-tree25 potrebuje v povprečju za 50 najbljižjih sosedov skoraj 100 krat več časa, kot v 3-dimenzionalnem prostoru z isto porazdelitvijo. Povprečno število izračunanih razdalj v množici krog10 je pričakovano najmanjše pri PM-drevesih s pivoti, a je vseeno okrog 30 odstotkov števila vseh točk v množici (slika 4.12). Večina metod preseže 50 odstotkov vseh točk in zato niso znatno hitrejše od izčrpnegra iskanja.



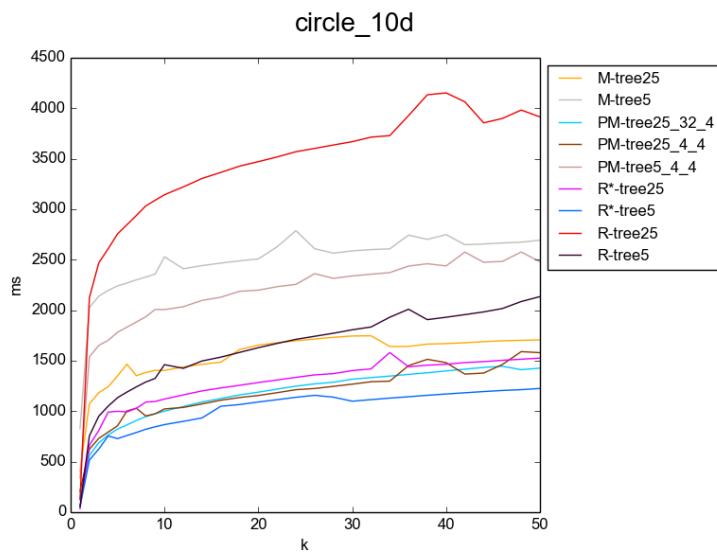
Slika 4.8: Rezultati časovnih meritev python implementacij na podatkovni množici krog3d v odvisnosti od števila iskanih elementov k.



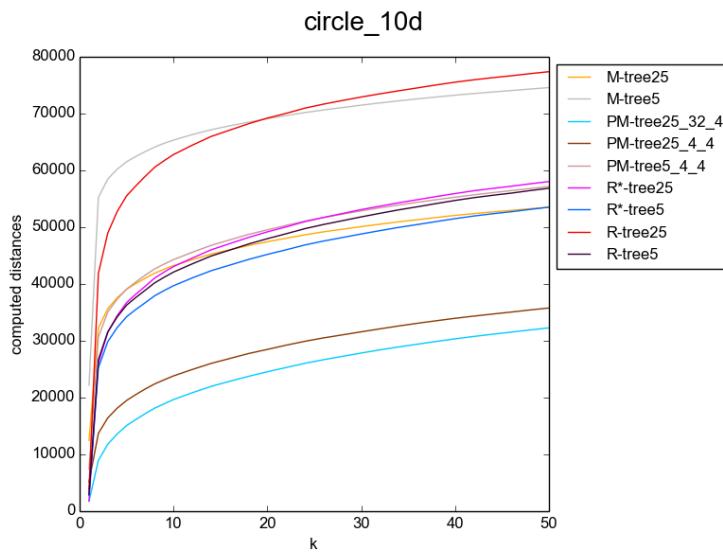
Slika 4.9: Povprečno število računanj razdalj na podatkovni množici krog3d v odvisnosti od števila iskanih elementov k.



Slika 4.10: Povprečno število pregledanih točk v listih na podatkovni množici krog3d v odvisnosti od števila iskanih elementov k.



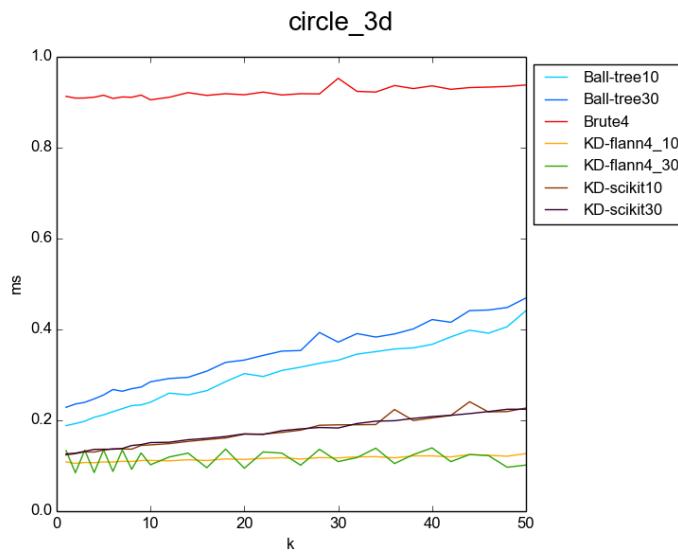
Slika 4.11: Rezultati časovnih meritev python implementacij na podatkovni množici krog10d v odvisnosti od števila iskanih elementov k.



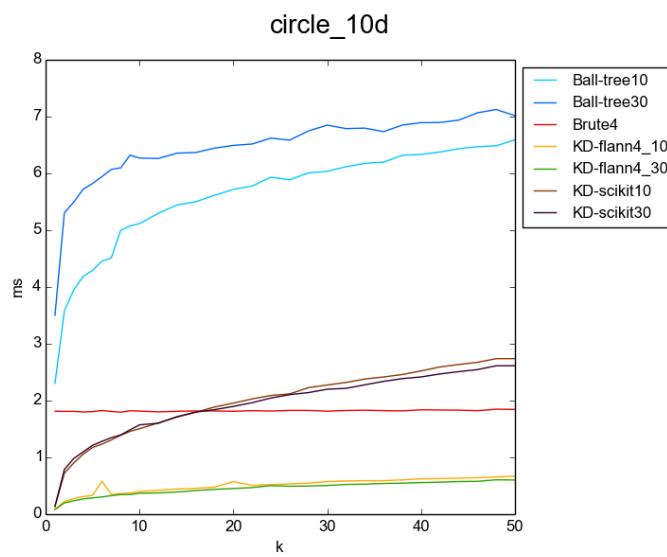
Slika 4.12: Povprečno število računanj razdalj na podatkovni množici krog10d v odvisnosti od števila iskanih elementov k.

#### 4.3.1.2 C++ implementacije

Slika 4.13 prikazuje, da je implementacija KD-drevesa iz knjižnice FLANN hitrejša kot implementacija v scikit-learn, ne glede na izbiro parametrov in da je implementacija ball-drevesa počasnejša od kd-dreves. Te meritve se pokažejo na vseh množicah, kar je prikazano v dodatku na slikah od B.26 do B.33. Pri scikit implementaciji KD-drevesa se z večanjem števila sosedov  $k$  hitreje slabša učinkovitost in pri iskanju v 10 dimenzionalnem prostoru postane počasnejše od izrčpnega iskanja, ki je označeno z Brute4 in implementirano v knjižnici FLANN (slika 4.14). Tudi implementacija KD-drevesa v FLANN knjižnici se precej upočasni in za razliko od iskanja v nižjih dimenzijah postane čas, ki je potreben za iskanje, odvisen od števila iskanih sosedov.



Slika 4.13: Rezultati časovnih meritev C++ implementacij na podatkovni množici krog3d v odvisnosti od števila iskanih elementov  $k$ .



Slika 4.14: Rezultati časovnih meritev C++ implementacij na podatkovni množici krog10d v odvisnosti od števila iskanih elementov  $k$ .

### 4.3.2 Približne metode

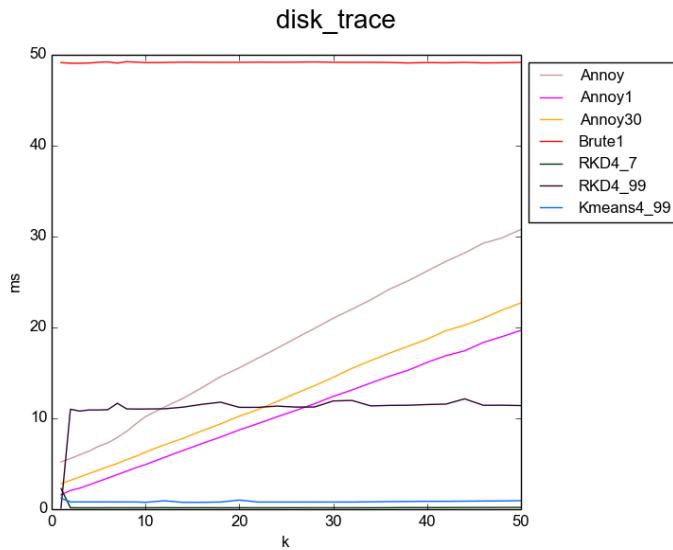
Pri testiranju približnih metod smo merili štiri količine in sicer hitrost iskanja sosedov, točnosti vrnjenih rezultatov, število vrnjenih rezultatov ter kvaliteto izkoriščanja večjedrinih procesorjev, ko podamo več iskanih točk naenkrat. Točnost je definirana z enačbo(4.1), kjer  $NN_r(X)$  predstavlja množico razdalj najbližjih sosedov dobljenih z modelom  $X$ .

$$\text{točnost(model)} = \frac{NN_r(\text{model}) \cap NN_r(\text{izčrpno})}{\text{število iskanih sosedov}} \quad (4.1)$$

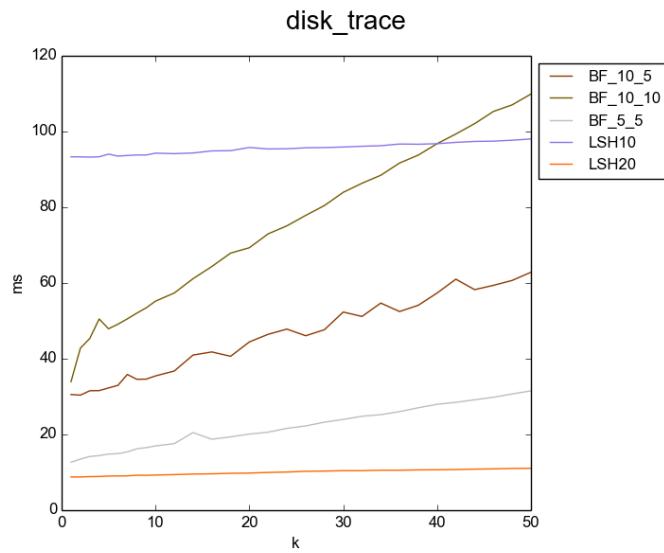
Iz slike 4.15, ki prikazuje časovne meritve C++ implementacij, je razvidno, da so metode implementirane v knjižnici FLANN hitrejše od Annoy-evih. Poleg tega vidimo, da čas iskanja pri FLANN implementacijah ni odvisen od števila iskanih sosedov, razen ko iščemo samo enega sosedja. Ker so iskane točke del množice, to točko hitro najdemo in se preiskovanje ustavi, ker je razdalja do  $k$ -tega sosedja enaka 0. Glede na to, da za vsa višja števila iskanih sosedov metoda potrebuje približno enako časa, lahko sklepamo, da pregleda približno enako število točk, ki je enako FLANN-ovemu parametru *checks*, ki predstavlja maksimalno število preiskanih točk. Vidimo, da se pri metodi RKD4\_99 iskanje precej upočasni, ko zahtevamo točnost 0.99. Za večjo točnost je potrebno pregledati večje število točk. Rezultati testiranj na preostalih množicah so prikazani v dodatku na slikah od B.34 do B.53.

Na sliki 4.16, ki prikazuje časovne meritve python implementacij, vidimo, da na hitrost iskanja pri metodah LSH število iskanih sosedov ne vpliva. Razlog je, da metoda preišče vedno vse tiste točke, ki imajo razpršeno vrednost enako razpršeni vrednosti iskane točke. Iz slike razberemo, da se z naraščanjem števila dreves in upoštevanih dimenzij veča čas iskanja.

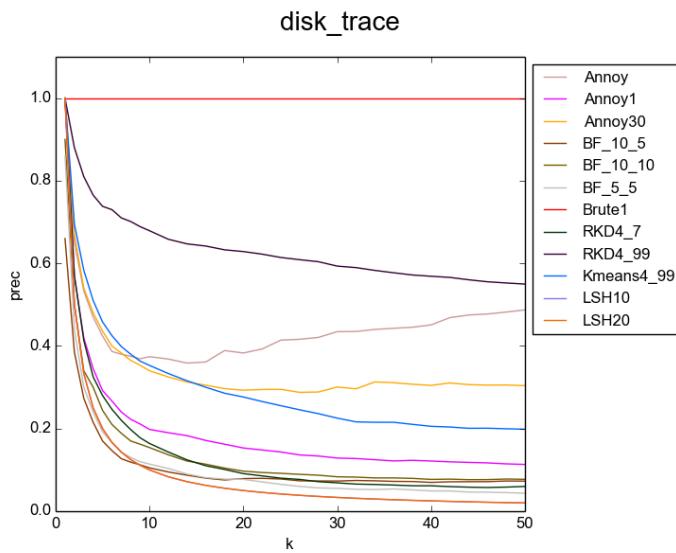
Rezultati meritev točnosti vrnjenih rezultatov so prikazani na sliki 4.17. Vidimo, da na množici disk trace nobena izmed metod ni doseglja točnosti večje kot 50 odstotkov pri  $k = 50$ . Glede točnosti se na tej podatkovni množici za najboljše izkažejo implementacije RKD-drevesa. Tudi na ostalih



Slika 4.15: Rezultati časovnih meritev C++ implementacij na podatkovni množici disk trace v odvisnosti od števila iskanih elementov k.



Slika 4.16: Rezultati časovnih meritev implementacij v programskejem jeziku python na podatkovni množici disk trace v odvisnosti od števila iskanih elementov k.

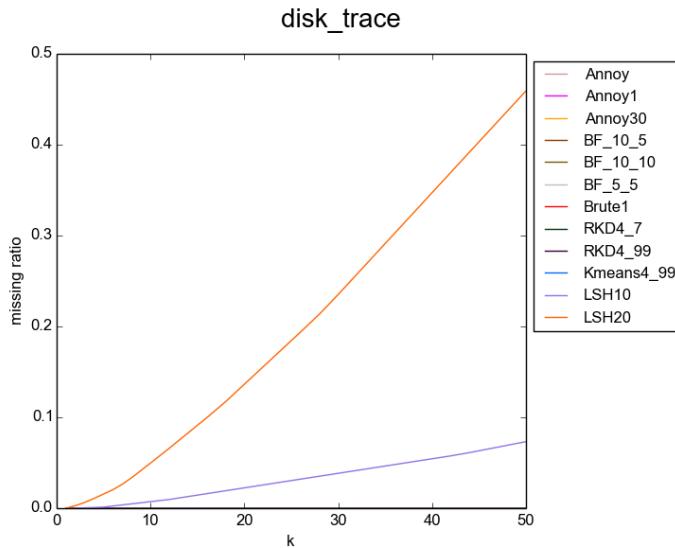


Slika 4.17: Rezultati točnosti metod na podatkovni množici disk trace v odvisnosti od števila iskanih elementov k.

množicah dobimo visoko točnost, vendar pri nekaterih visoka željena točnost povzroči, da iskanje ni hitrejše od izčrpnega iskanja (slika 4.20). V splošnem so metode iz knjižnice FLANN hitrejše od metod iz knjižnice Annoy.

Delež manjkajočih podatkov pri iskanju  $k$  najbližjih sosedov predstavlja slika 4.18. Opazimo, da ima le LSH težave pri vračanju vseh sosedov, to pa zato, ker je preveč odvisen od števila točk z isto razpršeno vrednostjo. Da bi LSH vračal vse zahtevane sosede, je potrebno imeti večje število podatkov, zmanjšati število projekcij ali pa spremeniti implementacijo v npr. večrazlično LSH. Pričakovano dobimo več sosedov pri 10 projekcijah, saj je različnih razpršenih vrednosti manj, zato koši v povprečju vsebujejo večje število točk. Ostale metode vrnejo vse zahtevane sosede in imajo zato delež manjkajočih podatkov enak 0.

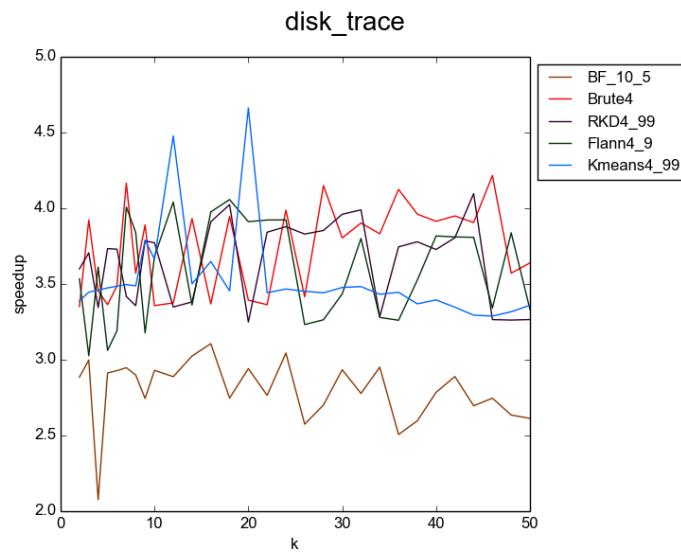
Primerjali smo tudi faktor povečanja hitrosti pri vzporednem iskanju najbližjih sosedov večih točk. Rezultati testiranja so prikazani na sliki 4.19. Ker smo vzporedno iskanje testirali le enkrat, pride do večjega odstopanja pri



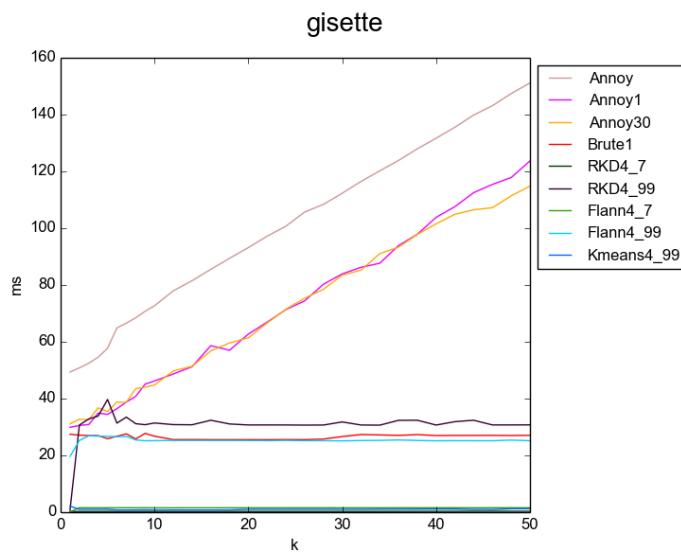
Slika 4.18: Graf deleža manjkajočih vrnjenih sosedov metod na množici disk trace v odvisnosti od števila iskanih elementov  $k$ . Večina metod vedno vrne zahtevano število sosedov in imajo zato delež enak 0.

meritvi časov. Iz grafa je razvidno, da implementacije iz knjižnice FLANN uporabijo vse 4 procesorje za istočasno iskanje najbljižjih sosedov večih točk. Implementacija gozda robov zna uporabiti enega manj, ker se iskanje sosedov ne izvaja v glavnem programu in zato en procesor čaka na rezultate preostalih. Nekonsistentne krivulje so najbrž posledica testiranja na manjšem številu točk in sklepamo, da bi se pri večjem številu krivulje bolj izravnale. Na sliki 4.20 za podatkovno množico gisette opazimo, da imajo krivulje podobno obliko kot pri prejšnji podatkovni množici s to razliko, da je požrešno iskanje hitrejše od Annoy metod in FLANN-ovega RKD-drevesa z zahtevano točnostjo 99 odstotkov. To, da mora RKD-drevo preiskati praktično vse sosede, če želi doseči točnost okrog 99 odstotkov, je pri tako visoko dimenzionalnih podatkih pričakovano. Annoy deluje dobro do približno 1000 dimenzijs, kar so omenili že avtorji implementacije v svoji dokumentaciji [4].

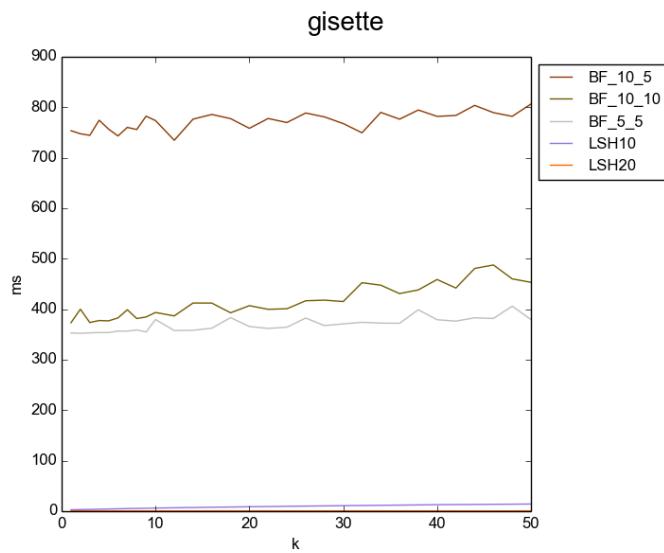
Slika 4.21, ki vsebuje python implementacije, potrdi, da je LSH hitrejši



Slika 4.19: Graf pohitritve metod pri uporabi več jeder na množici disk trace v odvisnosti od števila iskanih elementov k.



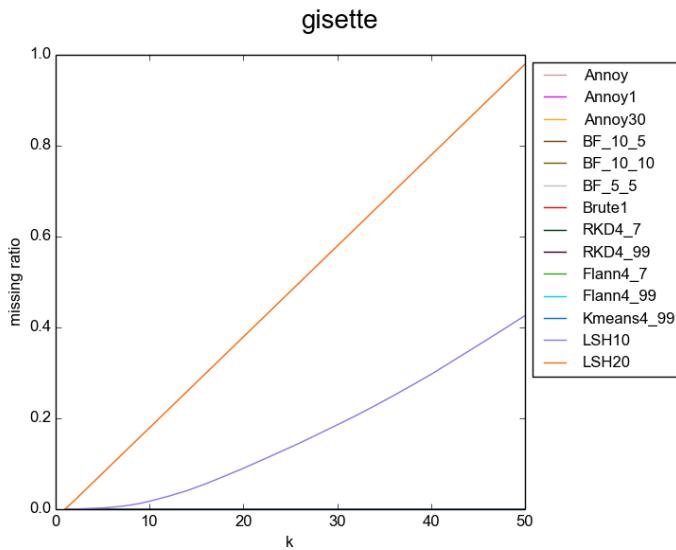
Slika 4.20: Rezultati časovnih meritev C++ implementacij pri iskanju najbližjih sosedov na podatkovni množici gisette.



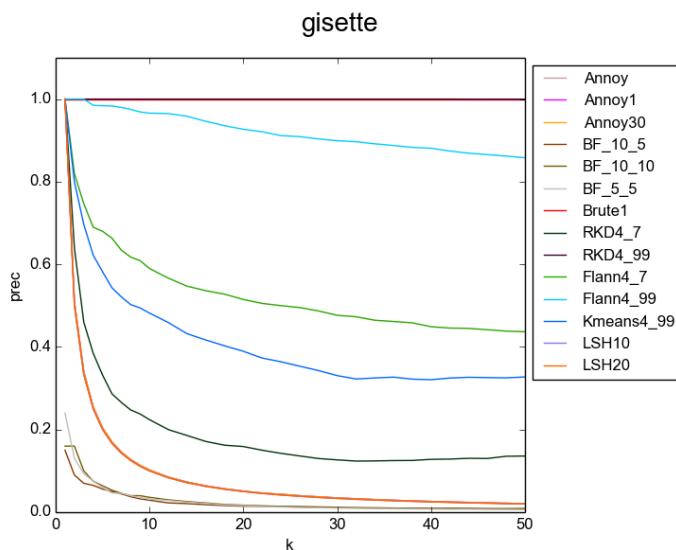
Slika 4.21: Rezultati časovnih meritev implementacij v programskej jeziku python pri iskanju najbližjih sosedov na podatkovni množici gisette.

od gozda robov, vendar je v tem primeru hitrejši tudi zato, ker množica gisette zaradi premajhnega števila podatkov ni primerna za LSH metode. To povzroči, da so koši slabo zasedeni in metoda velikokrat vrne premajhno število sosedov, kar je razvidno iz slike 4.22.

Točnost (slika 4.23) je pri RKD4\_99 metodi pričakovano 1, saj za iskanje sosedov potrebuje več časa, kot izčrpno iskanje. Glede na točnost ostalih metod sklepamo, da na tej množici nobena izmed metod ne zmore vrniti dobrih rezultatov v sprejemljivem času in so odvisne od števila pregledanih točk. To je pričakovano, saj metode težko vračajo dobre rezultate pri majhnem številu visokodimenzionalnih podatkov.



Slika 4.22: Graf deleža manjkajočih vrnjenih sosedov metod na množici gisette v odvisnosti od števila iskanih elementov  $k$ . Večina metod vedno vrne zahtevano število sosedov in imajo zato delež enak 0.



Slika 4.23: Rezultati točnosti metod na podatkovni množici gisette v odvisnosti od števila iskanih elementov  $k$ .

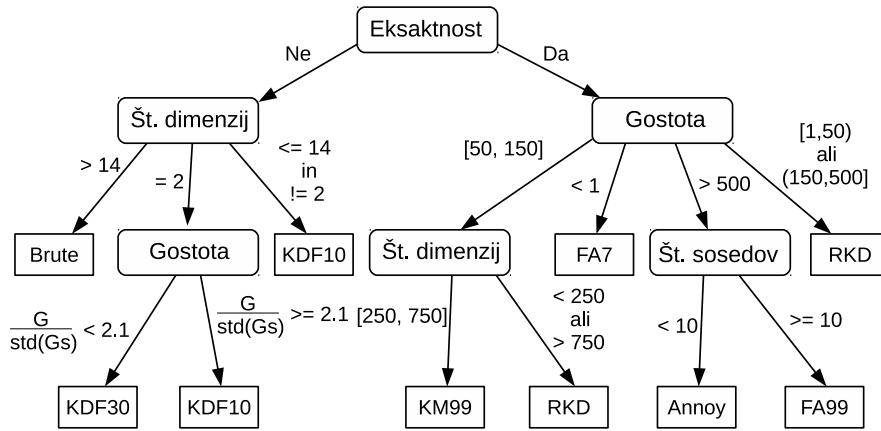
# Poglavlje 5

## Izbira metode

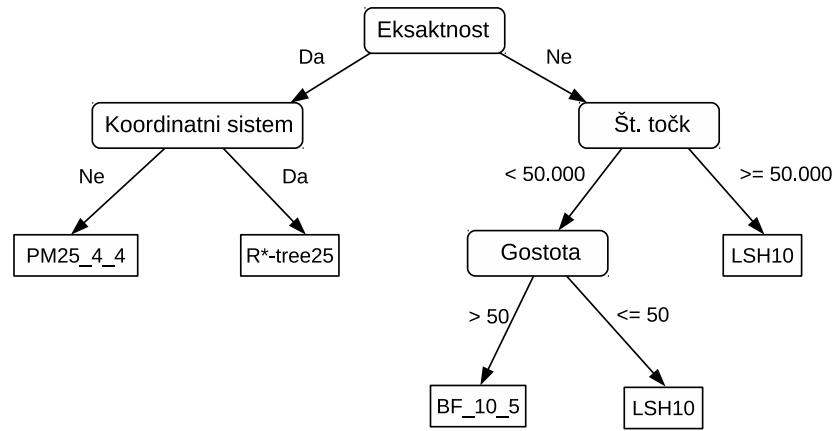
Kot povzetek ugotovitev predstavljamo dve odločitveni drevesi za avtomatско izbiro najprimernejše podatkovne strukture, eno za C++ in eno za python implementacije. Za dve drevesi smo se odločili, ker se strukture spisane v programskem jeziku python po hitrosti ne morejo primerjati s tistimi implementiranimi v C++. Pri C++ implementacijah je očitno, da se tiste iz knjižnice scikit-learn po hitrosti ne morejo primerjati s tistimi iz knjižnice FLANN. Takšna delitev je smiselna, saj bi najboljše metode iz python implementacij bilo v prihodnje smiselno implementirati tudi v C++ izvedbi. Na podlagi analize rezultatov smo za C++ metode zgradili drevo izbire primerrega algoritma, ki je prikazano na sliki 5.1. Kratica G pomeni gostoto točk, K predstavlja število sosedov, D število dimenzij, str(Gs) standardno deviacijo gostot točk množice, FA7 in FA99 FlannAuto algoritem pri točnosti 0.7 in 0.99, KM99 k-means pri točnosti 0.99, KDF10 KD-drevo iz knjižnice FLANN z maksimalnim številom točk v listih 10 in RKD-drevo predstavlja implementacijo RKD-drevesa iz knjižnice FLANN. Pri izbiri eksaktnega C++ algoritma uporabimo izčrpno iskanje iz knjižnice FLANN (označeno z imenom Brute) pri podatkih v 15 ali več dimenzijah, ker postanejo takrat ostale implementacije v C++ počasnejše. Za najprimernejši algoritem smo v ostalih primerih izbrali KDF10, razen v dvodimenzionalnem prostoru, kjer smo opazili, da pri množici gruče2 bolje deluje KD-drevo z maksimalno ve-

likostjo listov 10, pri množici krog2d pa z velikostjo 30. Ugotovili smo, da je pri omenjenih množicah razmerje med povprečno gostoto točk in njenim standardnim odklonom drugačno in smo to uporabili kot delitveni pogoj. V večini primerov pogoj pravilno razdeli metode, za boljšo delitev bi morali gostoto računati natančneje, vendar bi bilo to, zaradi implementacije v jeziku python, prepočasno. Pri izbiri približnega algoritma smo želeli, da je točnost vsaj okrog 0.4. Iz testiranj smo razbrali, da v primerih, ko je gostota manjša od 1, najbolje deluje FA7. RKD-drevo smo izbrali v primerih, ko iz testiranj nismo ugotovili zakonitosti.

Avtomatska izbira primerenega python algoritma je prikazana na sliki 5.2. Pri eksaktnih iskanjih skoraj vedno izberemo R\*-drevo z omejitvijo velikosti vozlišča 25, ker je bilo najhitrejše na vseh testiranjih. Ne izberemo ga le takrat, ko objekti niso postavljeni v koordinatnem sistemu in ker R\*-drevo na takih razdaljah ne deluje, izberemo PM-drevo, ki je bilo na testiranjih najboljše. Pri izbiri približnih algoritmov v primeru manjšega števila točk in gostote vsaj 50 izberemo BF\_10\_5, ki predstavlja gozd robov z 10 drevesi in 5 upoštevanimi dimenzijami, v ostalih primerih pa LSH10, ki je implementacija LSH algoritma iz knjižnice NearPy z 10 naključnimi binarnimi projekcijami.



Slika 5.1: Drevo izbire primerne implementacije C++ algoritmov.



Slika 5.2: Drevo izbire primerne implementacije python algoritmov.



# Poglavlje 6

## Skllepne ugotovitve

V nalogi smo opisali najbolj uporabljane algoritme za iskanje najbližjih sosedov. Te smo ločili na tiste, ki delujejo dobro v nizkodimenzionalnih prostorih in tiste, ki se uporablajo za iskanje približnih najbližjih sosedov. Nekaj implementacij smo vzeli iz obstoječih knjižnic, ostale smo implementirali sami. Algoritme smo testirali in beležili porabo pomnilnika, hitrost iskanja različnega števila sosedov in uspešnost izkoriščanja večjedrnih procesorjev. Pri približnih algoritmih smo testirali tudi njihovo točnost. Rezultate testiranj implementacij v jeziku C++ smo prikazali ločeno od tistih v jeziku python zaradi prevelike razlike v hitrosti izvajanja. Algoritme smo združili v python knjižnico in naredili programski vmesnik, ki omogoča njihovo uporabo na preprost in enoten način, ne glede na algoritmom. Knjižnica omogoča programerju uporabo tako eksaktnih, kot tudi približnih algoritmov. Knjižnice običajno ponujajo le eno izmed obeh vrst iskanja in se je zato običajno potrebno naučiti uporabljati vsaj dve knjižnici, morda pa celo več. Naša knjižnica omogoča tudi izris R in R\*-dreves, kar lahko pomaga programerjem pri razumevanju delovanja strukture. V algoritmu 10, ki se nahaja v dodatku C, je prikazan primer uporabe naše knjižnice na primeru R-drevesa. Na enak način lahko uporabimo poljubno strukturo, saj so imena metod enaka, v dodatek C pa smo vključili tudi tabelo z navedenimi parametri metod posameznih struktur. Zaradi počasnosti programskega jezika

python se naše implementacije ne morejo primerjati s tistimi spisanimi v jeziku C++, vendar menimo, da bi se implementaciji R\* in PM-drevesa v C oziroma C++ lahko po hitrosti primerjali z implementacijami KD-drevesa v knjižnici FLANN, saj je znano, da se v nekaterih primerih R\*-drevesa izkažejo za hitrejšo strukturo pri iskanju najbližjih sosedov. Poleg tega se KD-drevesa običajno zgradijo na že znanih podatkih, saj lahko zaporedno vstavljanje pripelje do neuravnoteženosti strukture, kar povzroči počasnejše iskanje. Ta problem lahko odpravi tako, da KD-drevo po potrebi ponovno zgradimo. Pri R\*-drevesih to ni potrebno, saj struktura z razdelitvijo in ponovnim vstavljanjem sama poskrbi, da ostane drevo uravnoteženo in hitro. PM-drevo je primerno za iskanje najbližjih sosedov tudi pri metrikah, kjer točk ne moremo razvrstiti v koordinatnem sistemu, kar je pogoj za delovanje pri KD in R-drevesih.

Naša analiza ima tudi nekaj pomanjkljivosti. Testiranje bi bilo bolj informativno in natančno, če bi iskali sosede še več točk in povečali število iskanih sosedov. Pri nizkodimenzionalnih podatkovnih množicah bi bilo dobro testirati tudi množice, ki vsebujejo večje število točk (npr. 1.000.000), pri visokodimenzionalnih pa bi za bolj natančno ugotovitev parametra najprimernejših metod morali testirati na več različnih množicah z različnimi dimenzijami. Tudi merjenje gostote bi lahko izboljšali, kar bi dalo bolj natančno gostoto, vendar bi zaradi počasnosti morali izračun spisati v C ali C++. Algoritem iskanja  $k$ -najbližjih sosedov pri gozdu robov smo sami preuredili iz algoritma za iskanje enega najbližjega soseda in morda obstaja boljša rešitev.

V prihodnosti bi bilo dobro implementirati gradnjo drevesa na vseh že znanih podatkih tudi za strukturo PM-drevo, saj je znano, da M-drevesa na ta način zgradijo boljšo strukturo, ker na ta način minimiziramo prekrivanje vozlišč. Tudi večrazlične lokalno-občutljive funkcije bi bilo vredno implementirati ter ga primerjati z ostalimi približnimi algoritmi.

# Literatura

- [1] Kohei Arai and Ali Ridho Barakbah. Hierarchical k-means: an algorithm for centroids initialization for k-means. *Reports of the Faculty of Science and Engineering*, 36(1):25–31, 2007.
- [2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331. ACM, 1990.
- [3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [4] Erik Bernhardsson. Annoy. <https://github.com/spotify/annoy>. Accessed: 26-03-2015.
- [5] King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the R-tree. *SIGMOD Rec.*, 27(3):16–21, September 1998.
- [6] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435. Morgan Kaufmann Publishers Inc., 1997.
- [7] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

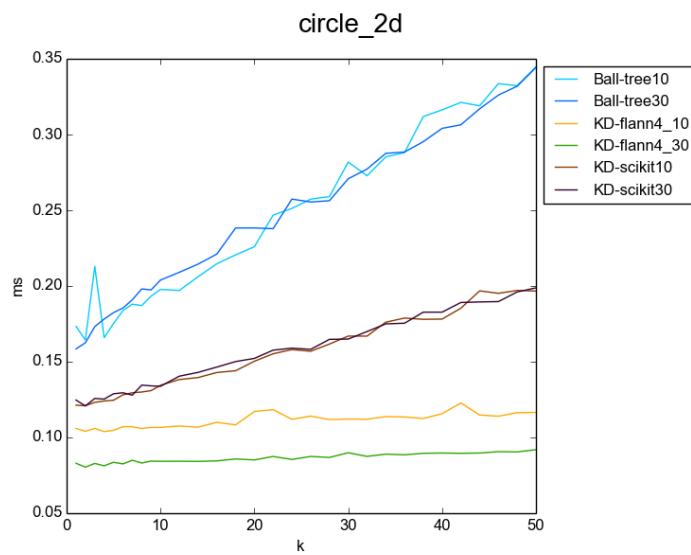
- [8] Stefan de Konink and Radim Baca. R-tree example. <http://commons.wikimedia.org/wiki/File:R-tree.svg>. Accessed: 11-01-2015.
- [9] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, volume 99, pages 518–529, 1999.
- [10] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
- [11] Primož Kariž. Nearest neighbour search library. <https://github.com/pkariz/nnssearch>. Accessed: 26-03-2015.
- [12] M. Lichman. UCI machine learning repository. <http://archive.ics.uci.edu/ml>. Accessed: 26-03-2015.
- [13] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, pages 825–832, 2004.
- [14] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very Large Data Bases*, pages 950–961. VLDB Endowment, 2007.
- [15] Charles Mathy, Nate Derbinsky, Jose Bento, Jonathan Rosenthal, and Jonathan Samuel Yedidia. The boundary forest algorithm for online supervised and unsupervised learning. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, jan 2015.
- [16] Marius Muja and David G. Lowe. FLANN - fast library for approximate nearest neighbors. <https://github.com/mariusmuja/flann>. Accessed: 26-03-2015.

- [17] Stephen Malvern Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.
- [18] Chanop Silpa-Anan and Richard Hartley. Optimised KD-trees for fast image descriptor matching. In *IEEE Conference on Computer Vision and Pattern Recognition, 2008. CVPR 2008.*, pages 1–8. IEEE, 2008.
- [19] Tomás Skopal. Pivoting M-tree: A metric access method for efficient similarity search. In *Databases, Texts, Specifications, Objects*, volume 4, pages 27–37, 2004.
- [20] Jeffrey David Ullman, Jure Leskovec, and Anand Rajaraman. *Mining of Massive Datasets*. Cambridge University Press, 2011.
- [21] Jonathan R Wells, Kai Ming Ting, and Takashi Washio. LiNearN: A new approach to nearest neighbour density estimator. *Pattern Recognition*, 47(8):2702–2720, 2014.
- [22] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity search: the metric space approach*, volume 32. Springer Science & Business Media, 2006.

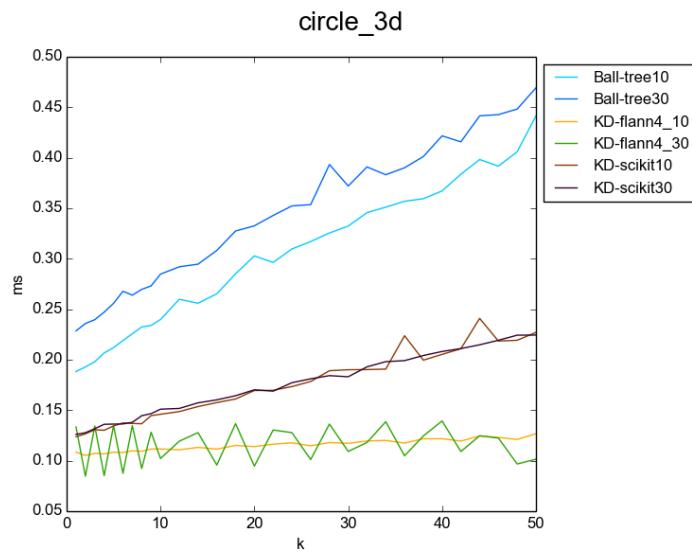


# Dodatek A

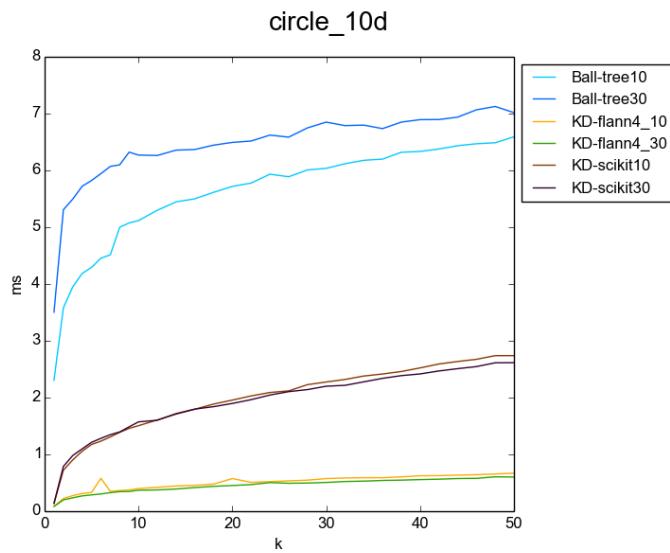
Prikaz preostalih rezultatov testiranja parametrov družin algoritmov KD-drevo in ball-drevo, R in R\*-drevo, PM-drevo, gozd robov in LSH na preostalih množicah.



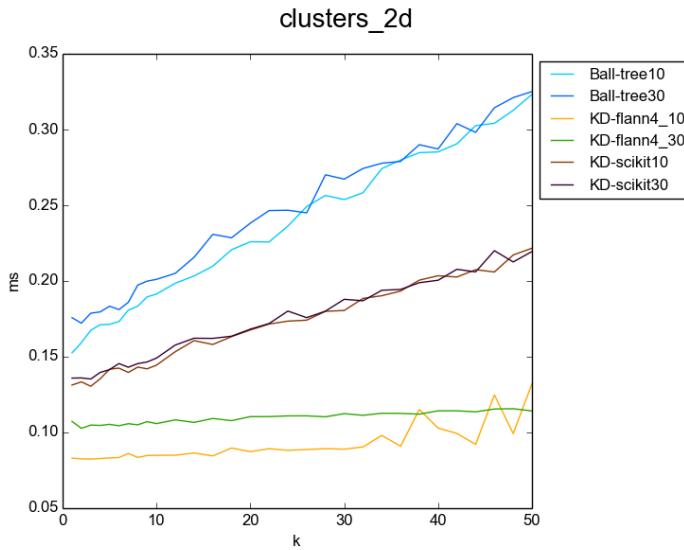
Slika A.2: Rezultati časovnih meritev KD in ball dreves na množici krog2d pri različnem števila iskanih elementov k in velikosti listov.



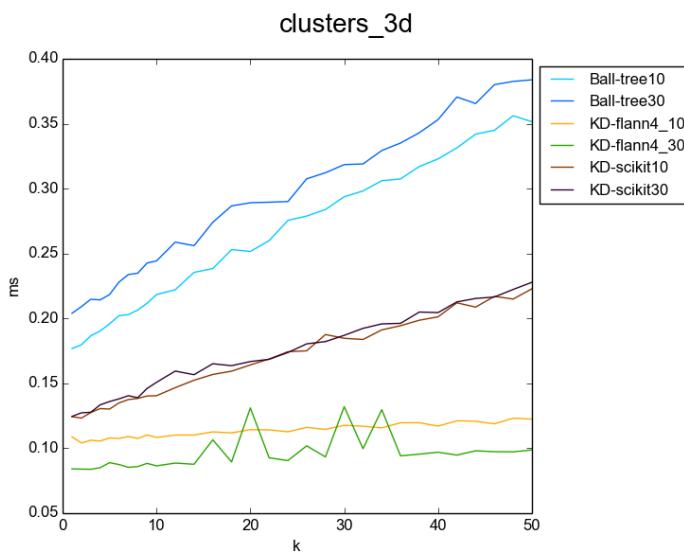
Slika A.1: Rezultati časovnih meritev KD in ball dreves na množici krog3d pri različnem števila iskanih elementov k in velikosti listov.



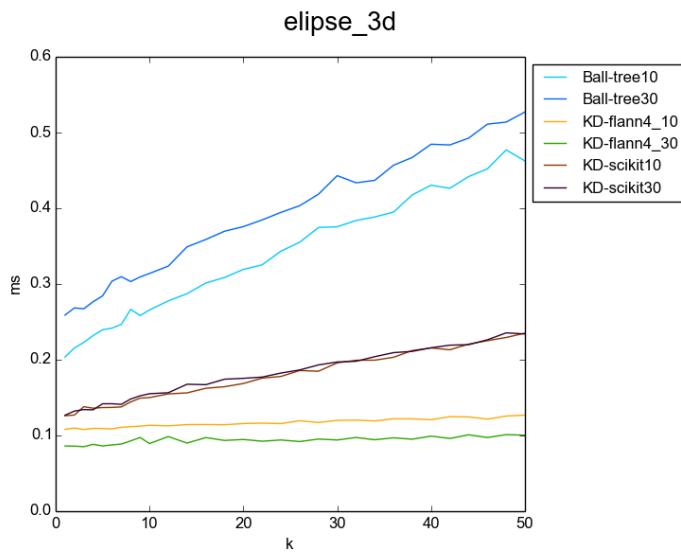
Slika A.3: Rezultati časovnih meritev KD in ball dreves na množici krog10d pri različnem števila iskanih elementov k in velikosti listov.



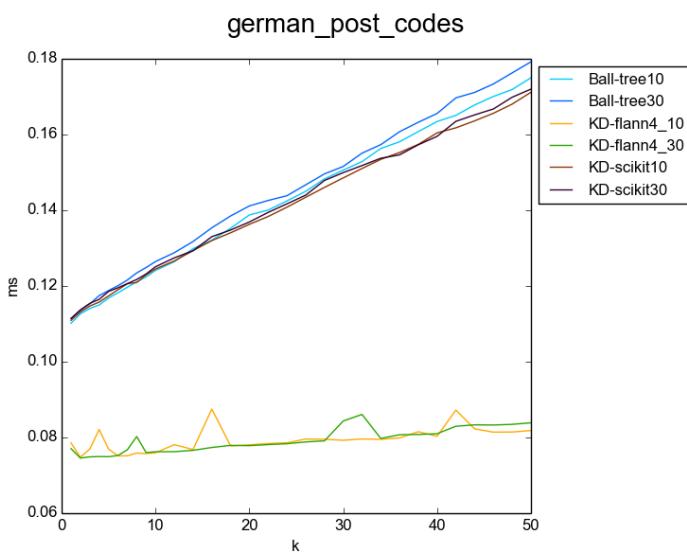
Slika A.4: Rezultati časovnih meritev KD in ball dreves na množici gruče2d pri različnem števila iskanih elementov k in velikosti listov.



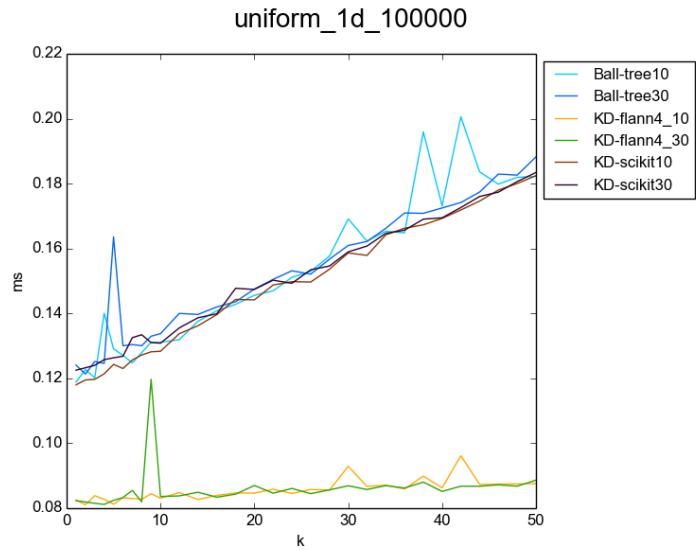
Slika A.5: Rezultati časovnih meritev KD in ball dreves na množici gruče3d pri različnem števila iskanih elementov k in velikosti listov.



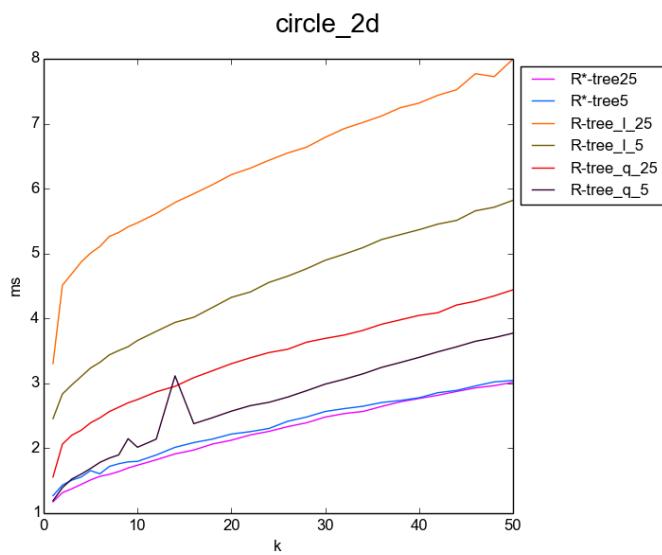
Slika A.6: Rezultati časovnih meritev KD in ball dreves na množici elipsa3d pri različnem števila iskanih elementov  $k$  in velikosti listov.



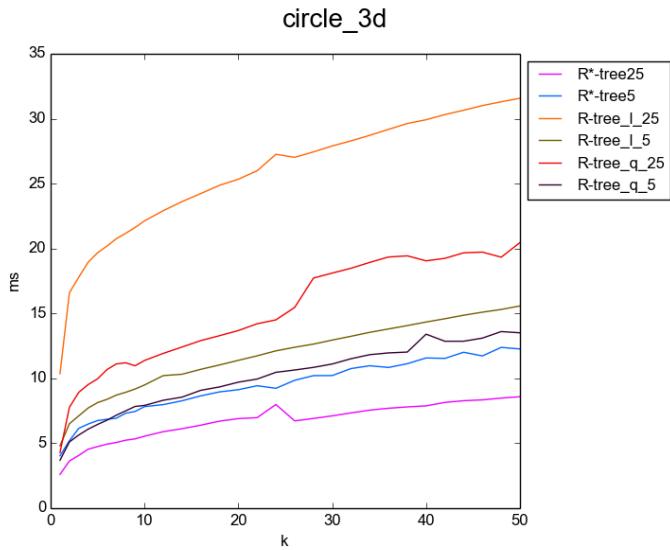
Slika A.7: Rezultati časovnih meritev KD in ball dreves na množici NPŠ pri različnem števila iskanih elementov  $k$  in velikosti listov.



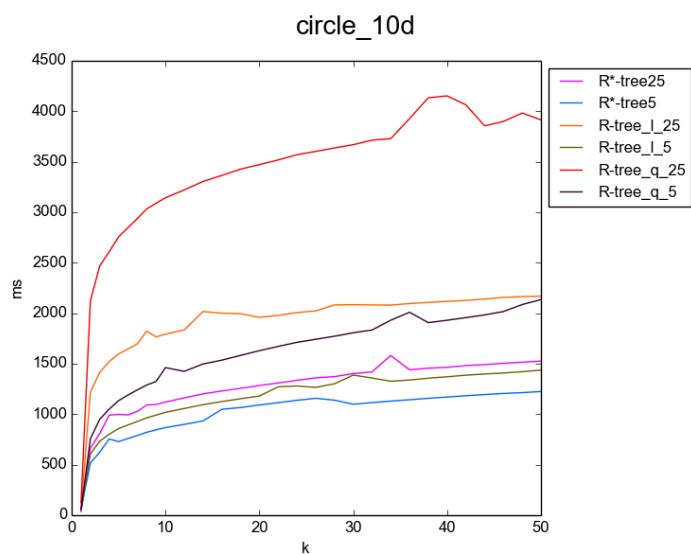
Slika A.8: Rezultati časovnih meritev KD in ball dreves na množici uniform1d pri različnem števila iskanih elementov  $k$  in velikosti listov.



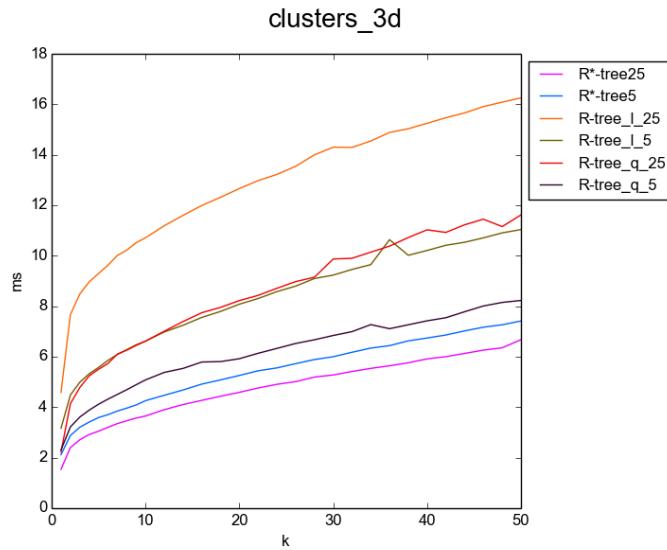
Slika A.9: Rezultati časovnih meritev R in R\* dreves na množici krog2d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



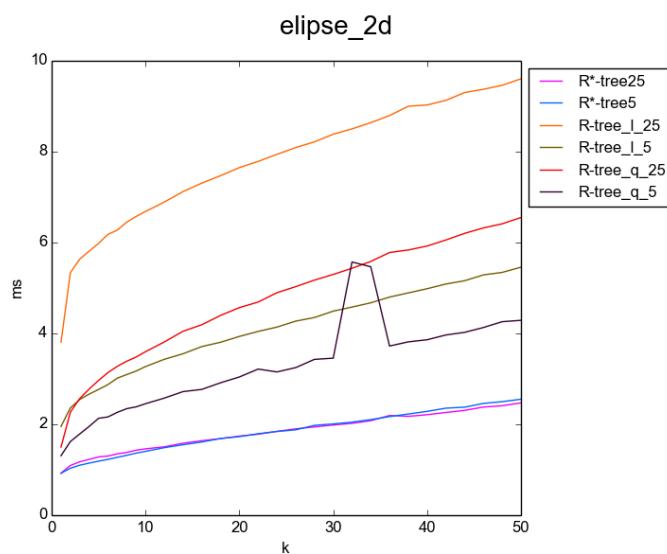
Slika A.10: Rezultati časovnih meritev R in R\* dreves na množici krog3d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



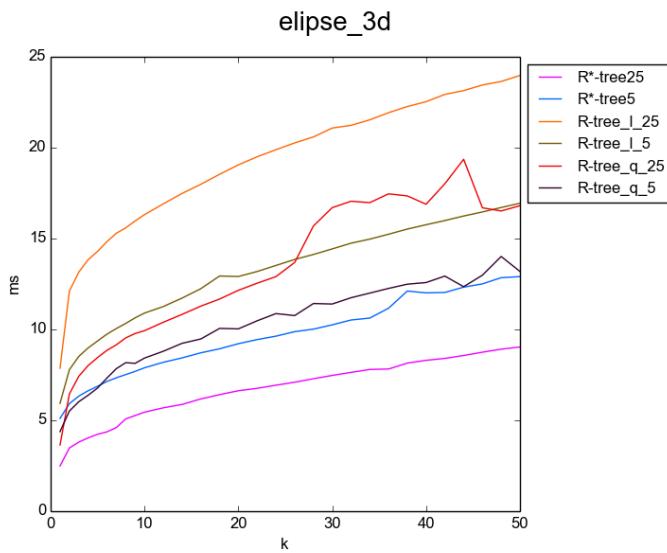
Slika A.11: Rezultati časovnih meritev R in R\* dreves na množici krog10d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



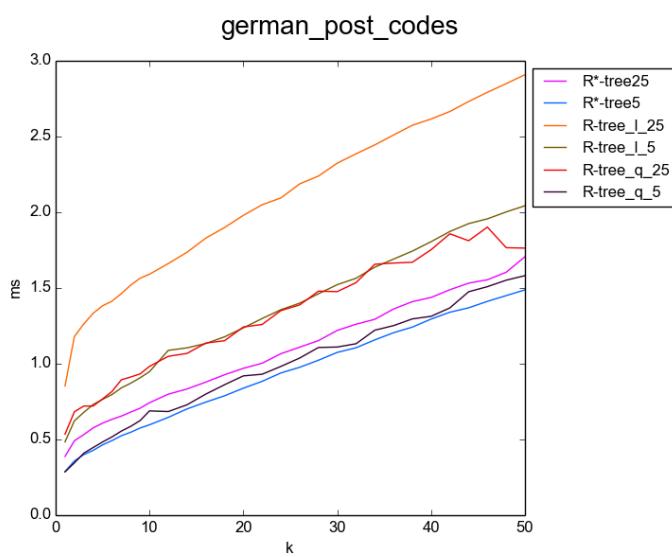
Slika A.12: Rezultati časovnih meritev R in R\* dreves na množici gruče3d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



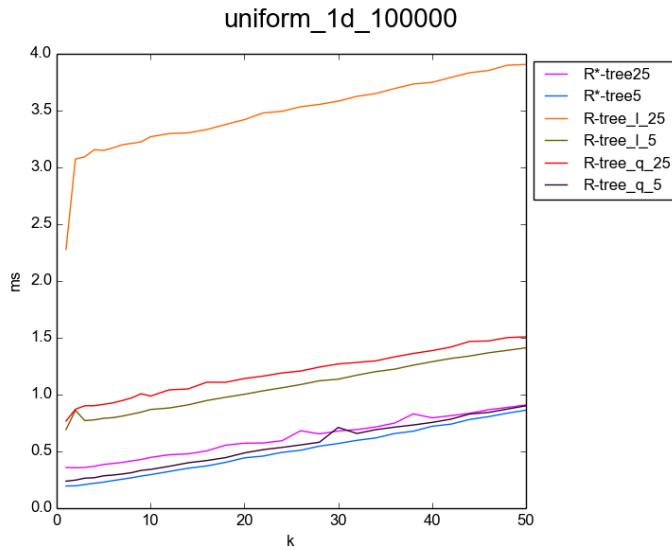
Slika A.13: Rezultati časovnih meritev R in R\* dreves na množici elipsa2d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



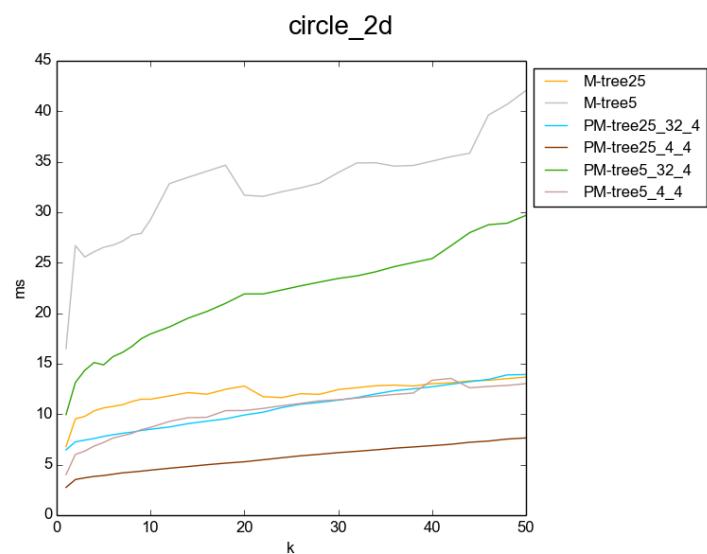
Slika A.14: Rezultati časovnih meritev R in R\* dreves na množici elipsa3d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



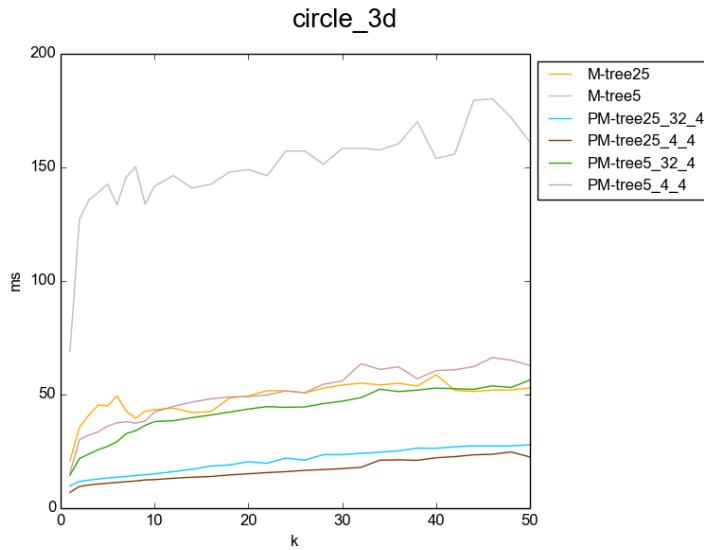
Slika A.15: Rezultati časovnih meritev R in R\* dreves na množici NPŠ v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



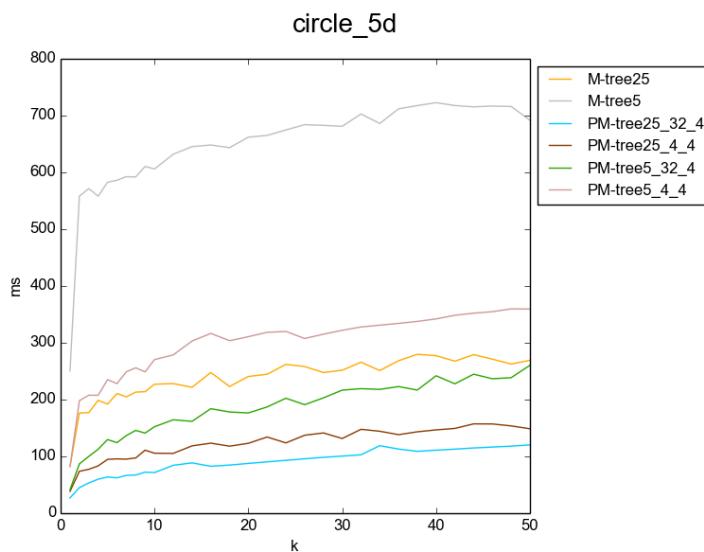
Slika A.16: Rezultati časovnih meritev R in  $R^*$  dreves na množici uniform1d v odvisnosti od števila iskanih sosedov, razdelitve in velikosti vozlišč.



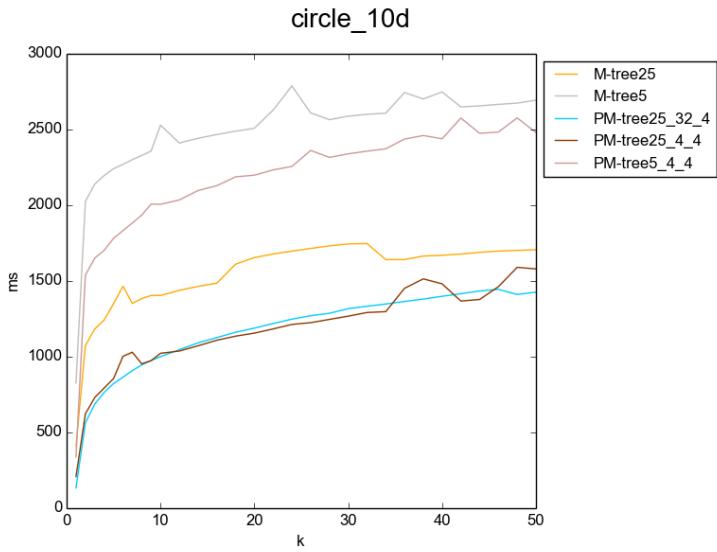
Slika A.17: Rezultati časovnih meritev PM dreves na množici krog2d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



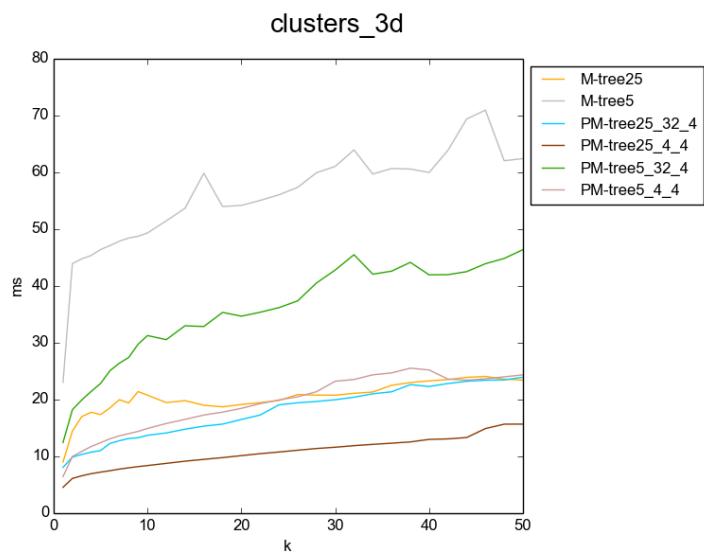
Slika A.18: Rezultati časovnih meritev PM dreves na množici krog3d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



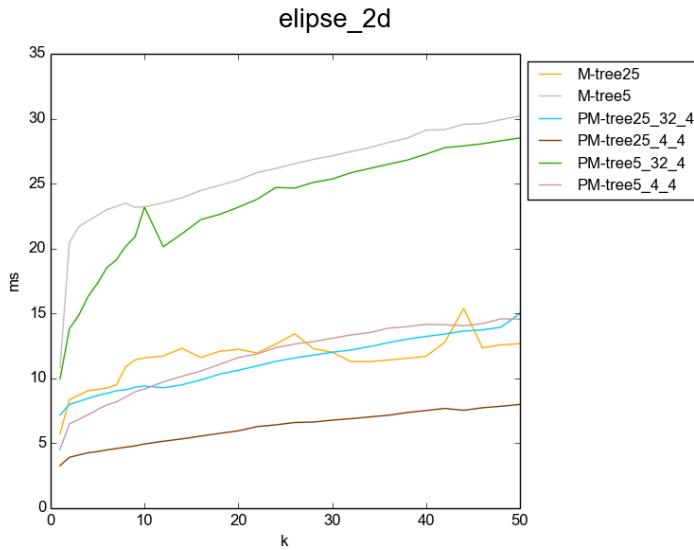
Slika A.19: Rezultati časovnih meritev PM dreves na množici krog5d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



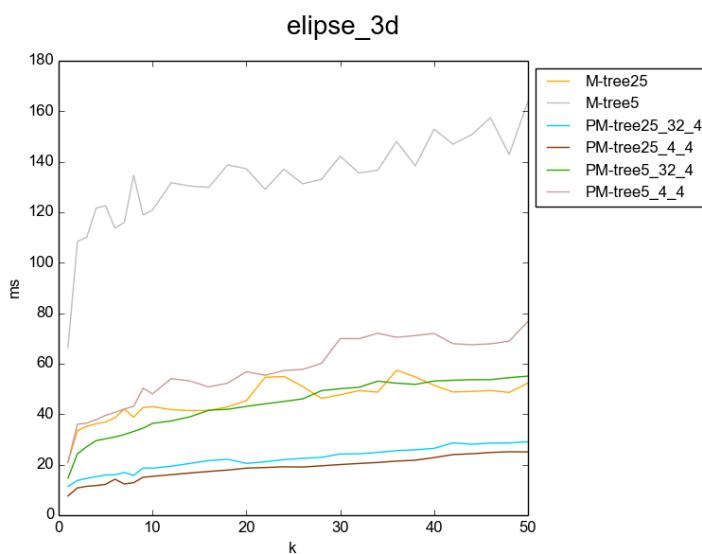
Slika A.20: Rezultati časovnih meritev PM dreves na množici krog10d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



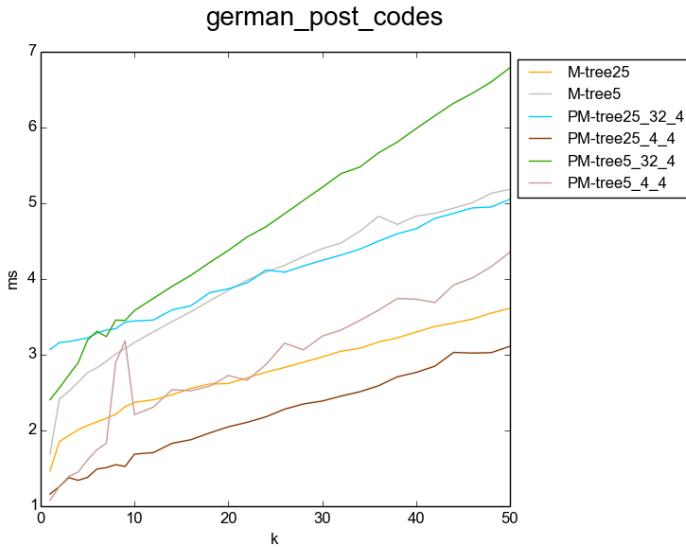
Slika A.21: Rezultati časovnih meritev PM dreves na množici gruče3d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



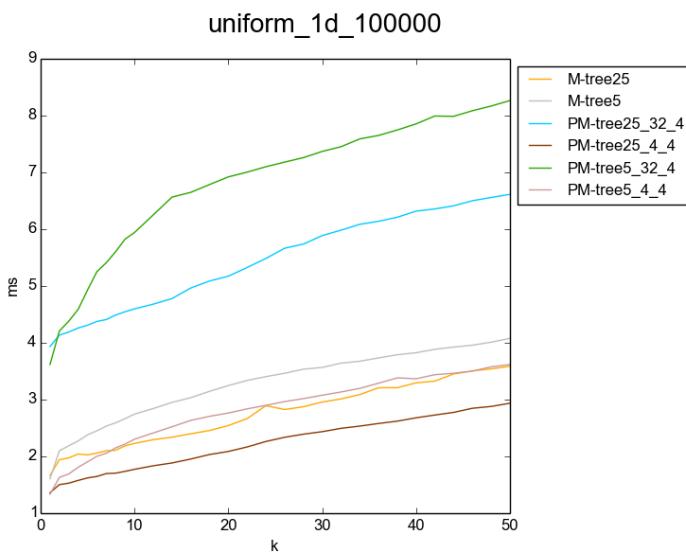
Slika A.22: Rezultati časovnih meritev PM dreves na množici elipsa2d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



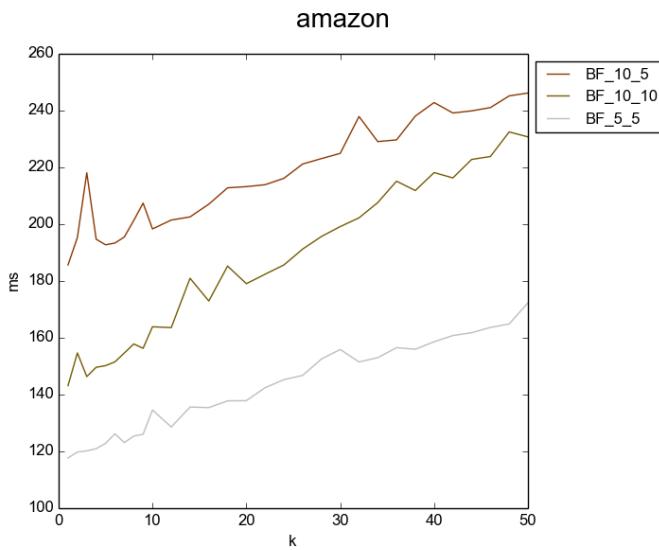
Slika A.23: Rezultati časovnih meritev PM dreves na množici elipsa3d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



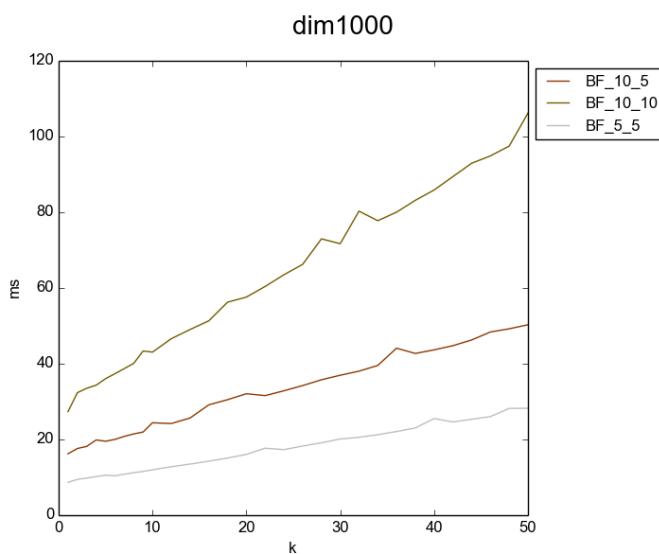
Slika A.24: Rezultati časovnih meritev PM dreves na množici NPŠ v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



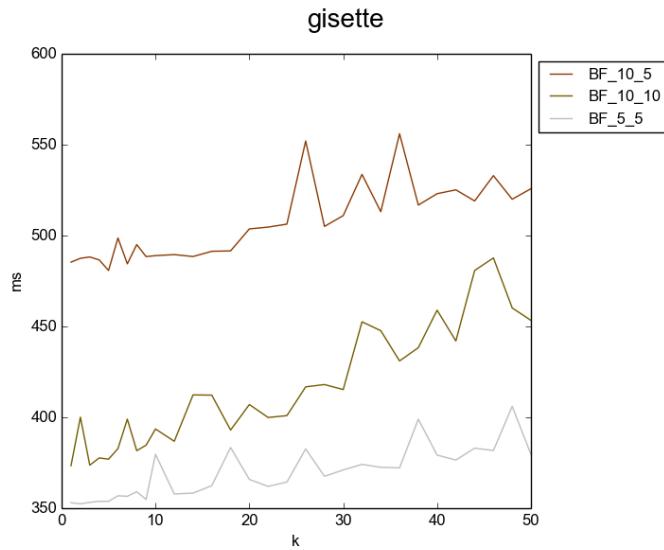
Slika A.25: Rezultati časovnih meritev PM dreves na množici uniform1d v odvisnosti od števila iskanih sosedov, pivotov in velikosti vozlišč.



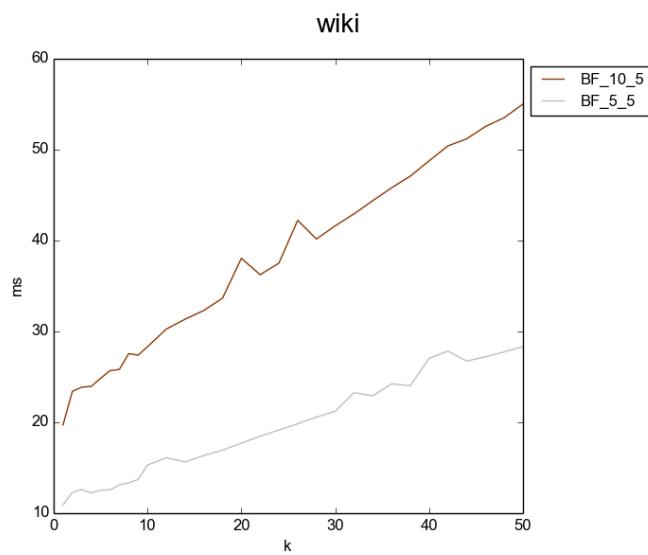
Slika A.26: Rezultati časovnih meritev gozda robov na množici amazon v odvisnosti od števila iskanih sosedov, dreves in upoštevanih dimenzij.



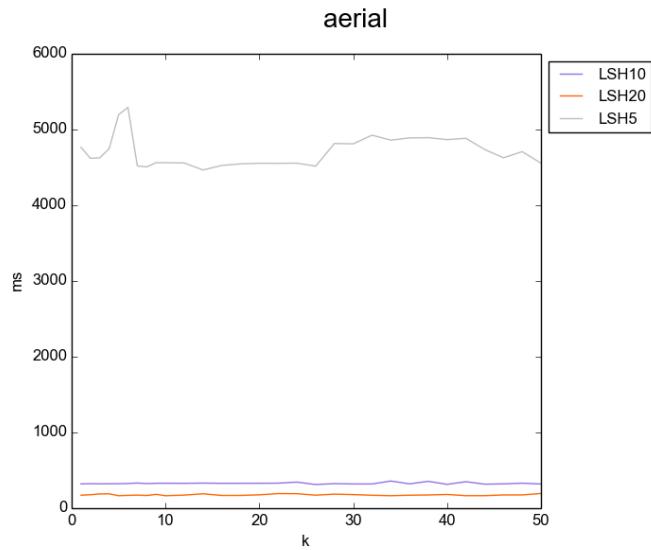
Slika A.27: Rezultati časovnih meritev gozda robov na množici dim1000 v odvisnosti od števila iskanih sosedov, dreves in upoštevanih dimenzij.



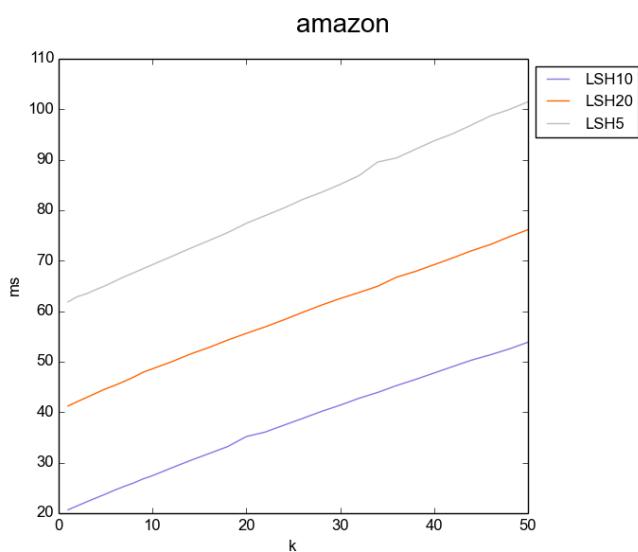
Slika A.28: Rezultati časovnih meritev gozda robov na množici gisette v odvisnosti od števila iskanih sosedov, dreves in upoštevanih dimenzij.



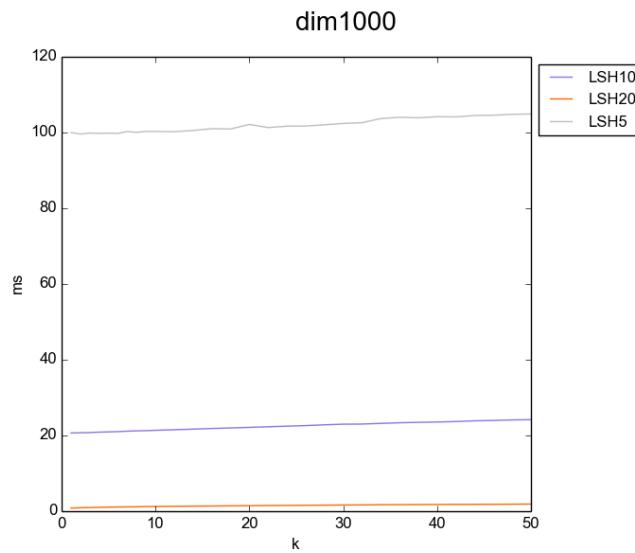
Slika A.29: Rezultati časovnih meritev gozda robov na množici wiki v odvisnosti od števila iskanih sosedov, dreves in upoštevanih dimenzij.



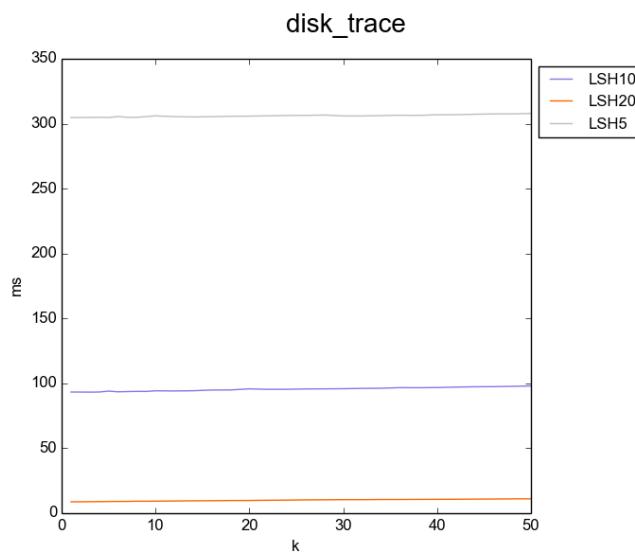
Slika A.30: Rezultati časovnih meritev LSH na množici aerial v odvisnosti od števila iskanih sosedov in binarnih projekcij.



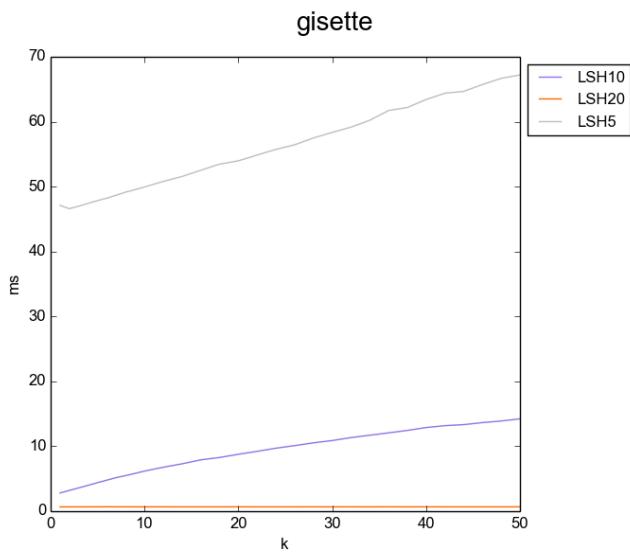
Slika A.31: Rezultati časovnih meritev LSH na množici amazon v odvisnosti od števila iskanih sosedov in binarnih projekcij.



Slika A.32: Rezultati časovnih meritev LSH na množici dim1000 v odvisnosti od števila iskanih sosedov in binarnih projekcij.



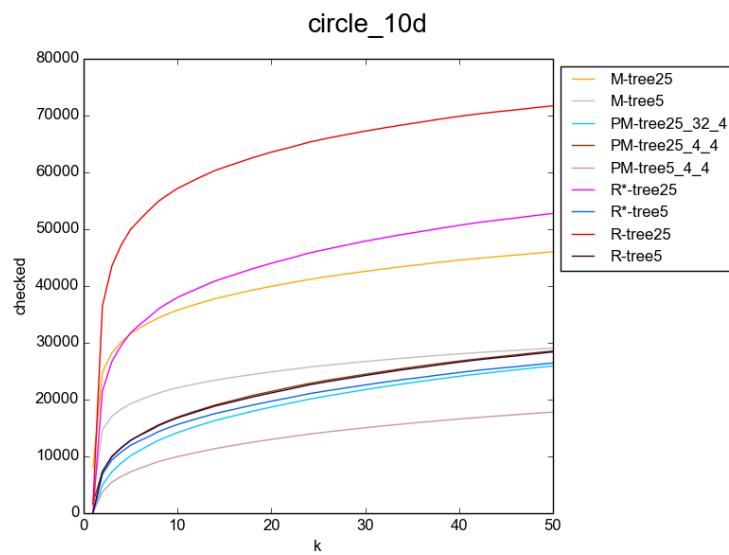
Slika A.33: Rezultati časovnih meritev LSH na množici disk trace v odvisnosti od števila iskanih sosedov in binarnih projekcij.



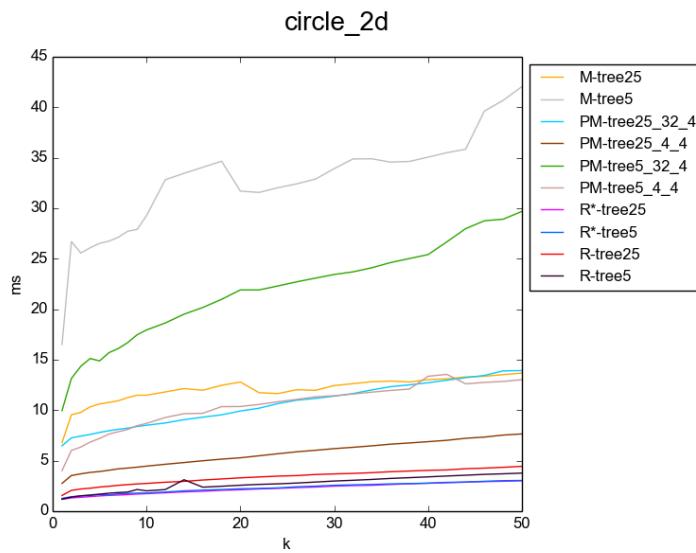
Slika A.34: Rezultati časovnih meritev LSH na množici gisette v odvisnosti od števila iskanih sosedov in binarnih projekcij.

## Dodatek B

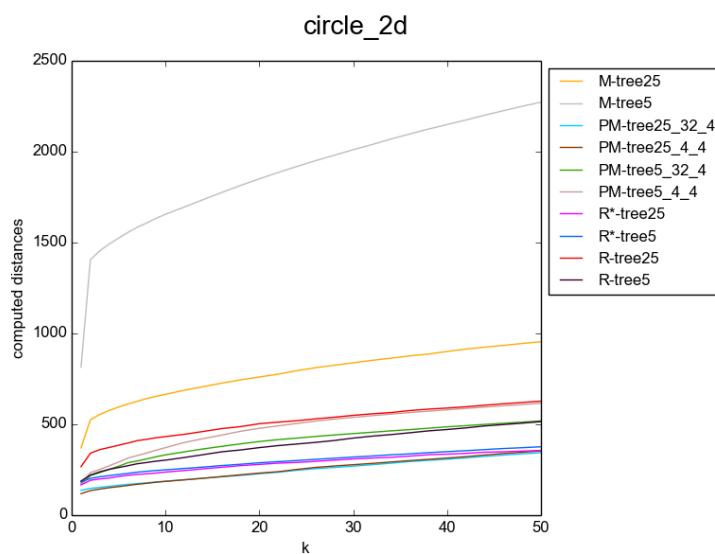
Dodatek vsebuje preostale rezultate testiranj eksaktnih in približnih metod. Rezultati izkoriščanja večjedrnih procesorjev vsebujejo konice na slikah. Razlog zanje je lahko premajhno število vzporedno iskanih točk ali premajhno število ponovitev testiranj pri tako hitrih izvedbah.



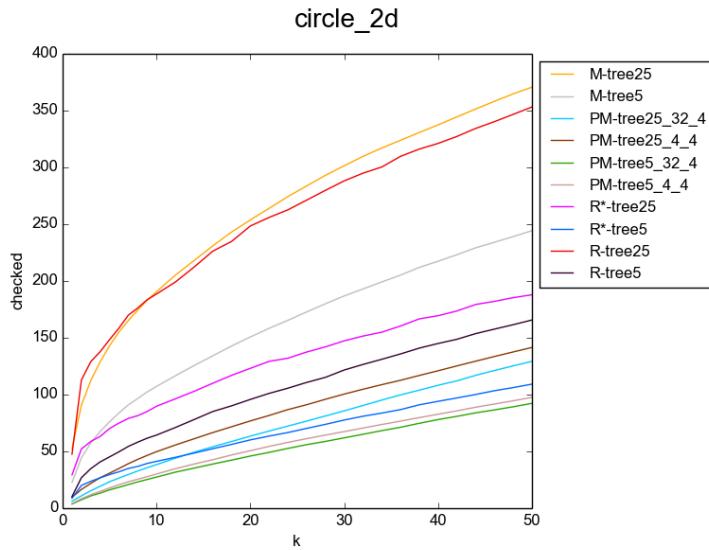
Slika B.1: Povprečno število pregledanih točk v listih na množici krog10d v odvisnosti od števila iskanih elementov k.



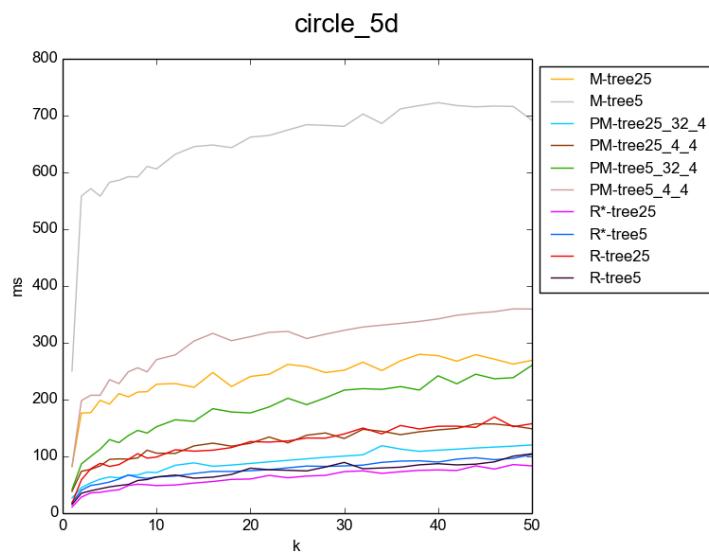
Slika B.2: Rezultati časovnih meritev python implementacij na množici krog2d v odvisnosti od števila iskanih elementov k.



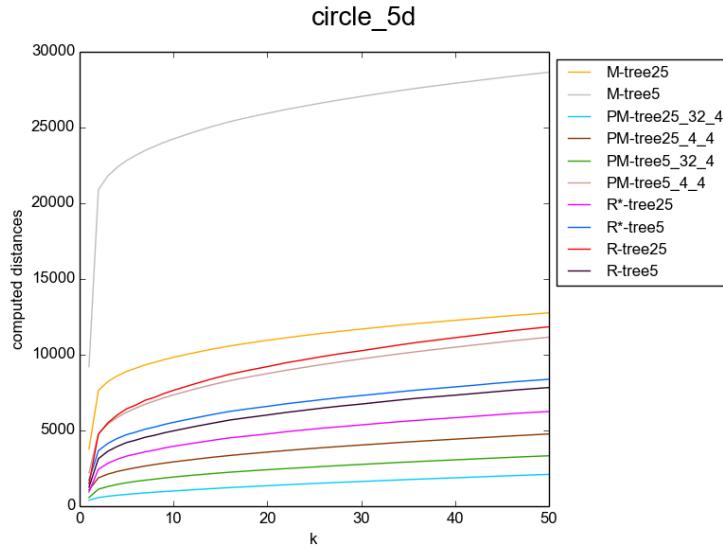
Slika B.3: Povprečno število računanj razdalj na množici krog2d v odvisnosti od števila iskanih elementov k.



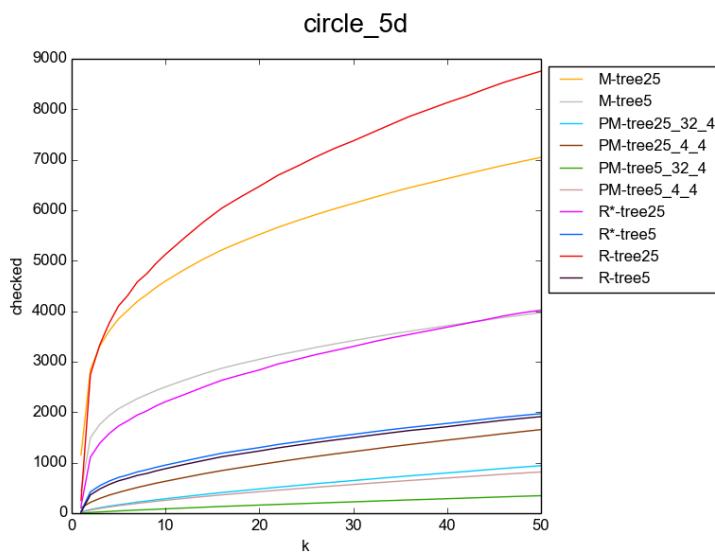
Slika B.4: Povprečno število pregledanih točk v listih na množici krog2d v odvisnosti od števila iskanih elementov k.



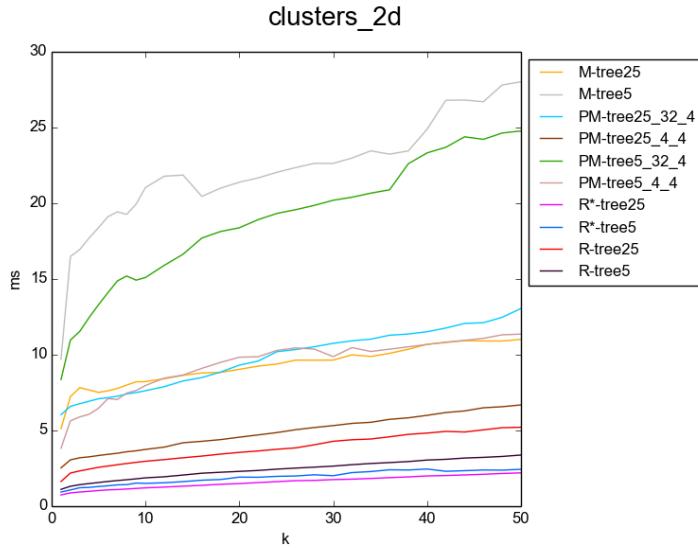
Slika B.5: Rezultati časovnih meritev python implementacij na množici krog5d v odvisnosti od števila iskanih elementov k.



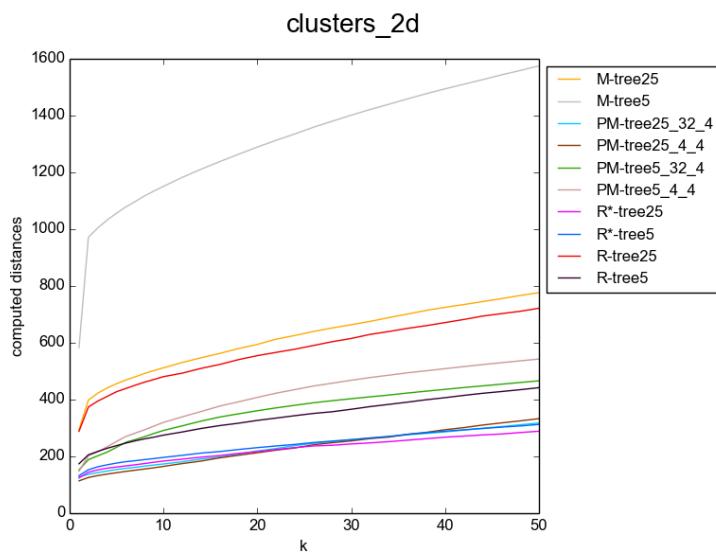
Slika B.6: Povprečno število računanj razdalj na množici krog5d v odvisnosti od števila iskanih elementov k.



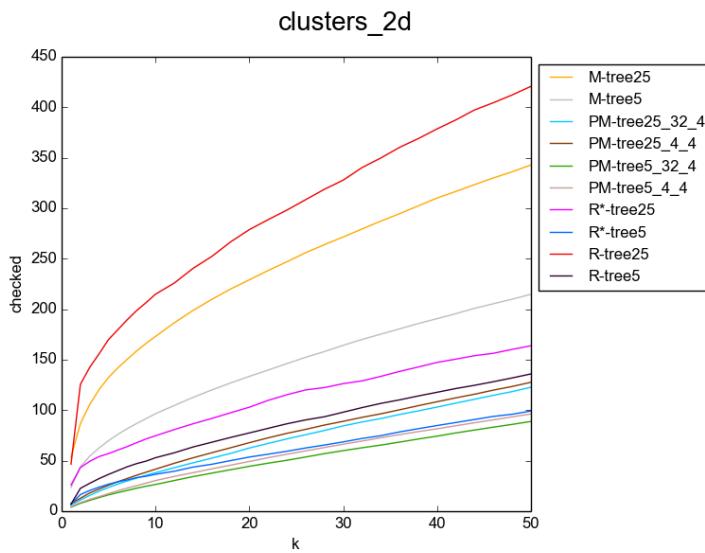
Slika B.7: Povprečno število pregledanih točk v listih na množici krog5d v odvisnosti od števila iskanih elementov k.



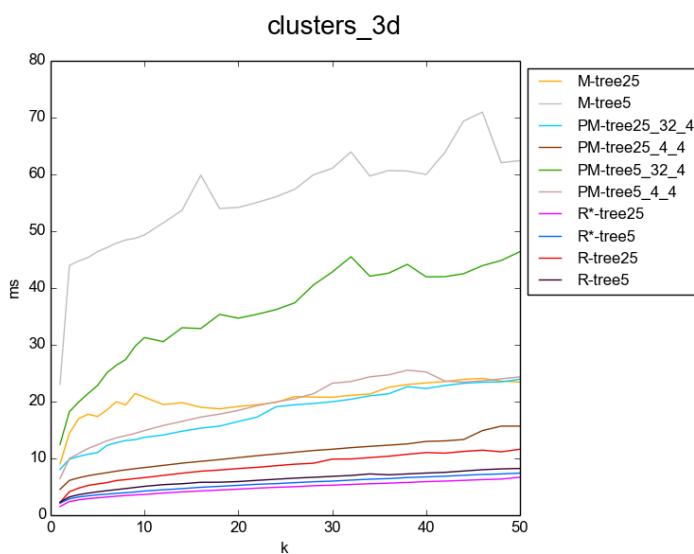
Slika B.8: Rezultati časovnih meritev python implementacij na množici gruče2d v odvisnosti od števila iskanih elementov k.



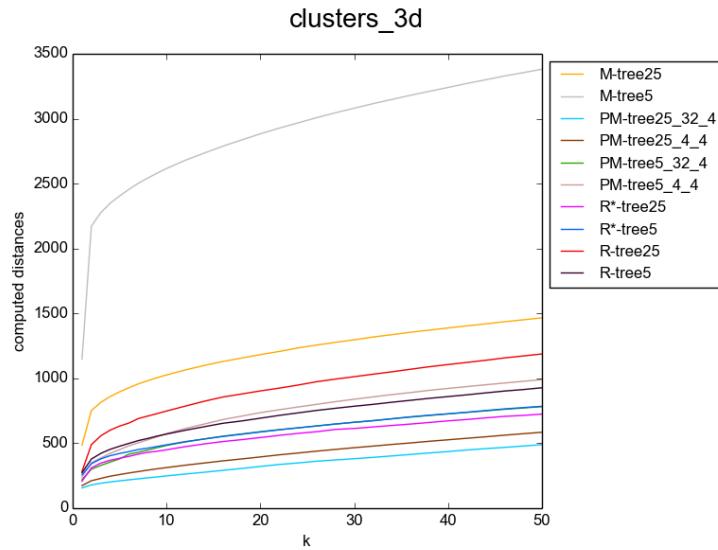
Slika B.9: Povprečno število računanj razdalj na množici gruče2d v odvisnosti od števila iskanih elementov k.



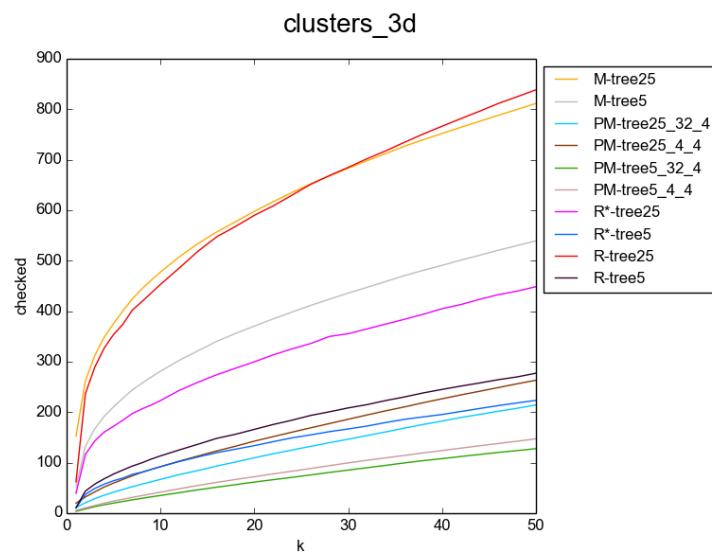
Slika B.10: Povprečno število pregledanih točk v listih na množici gruče2d v odvisnosti od števila iskanih elementov k.



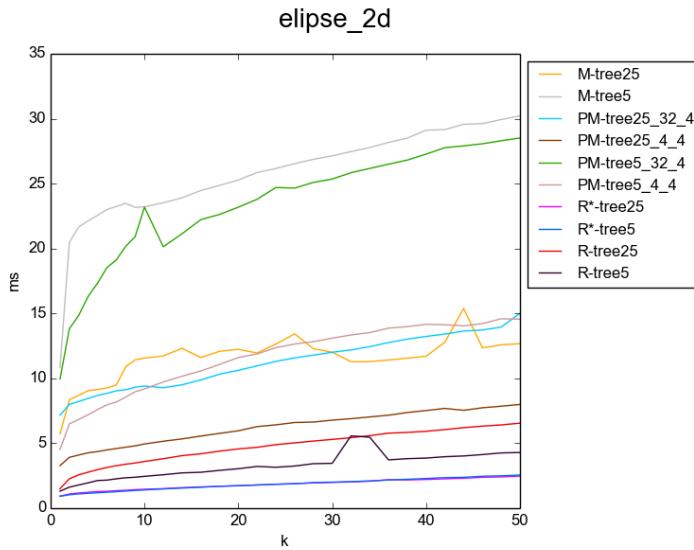
Slika B.11: Rezultati časovnih meritev python implementacij na množici gruče3d v odvisnosti od števila iskanih elementov k.



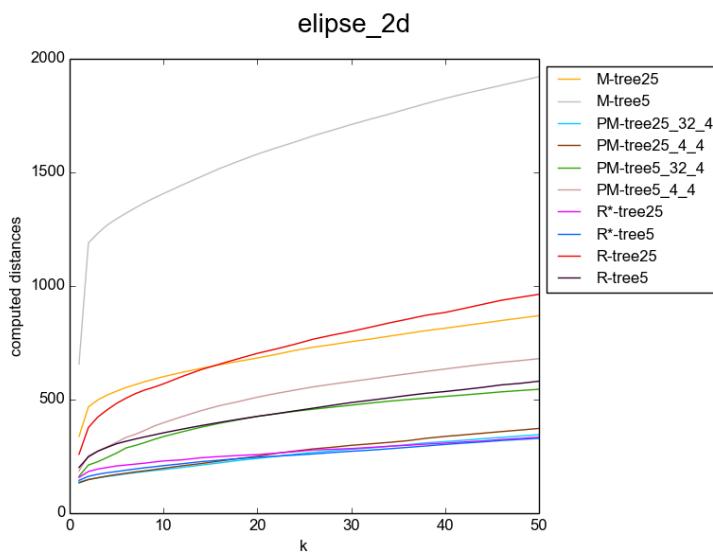
Slika B.12: Povprečno število računanj razdalj na množici gruče3d v odvisnosti od števila iskanih elementov k.



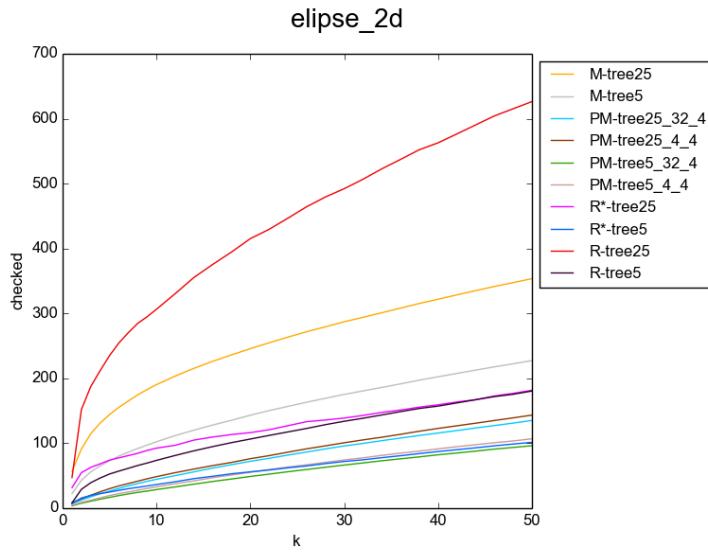
Slika B.13: Povprečno število pregledanih točk v listih na množici gruče3d v odvisnosti od števila iskanih elementov k.



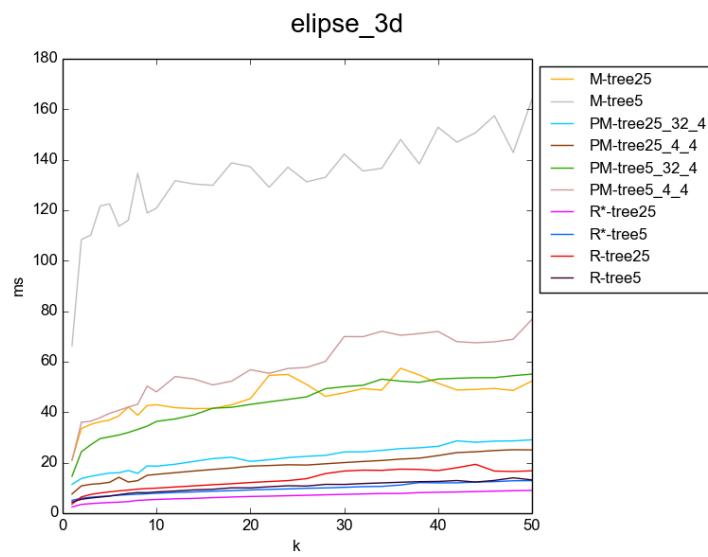
Slika B.14: Rezultati časovnih meritev python implementacij na množici elipsa2d v odvisnosti od števila iskanih elementov k.



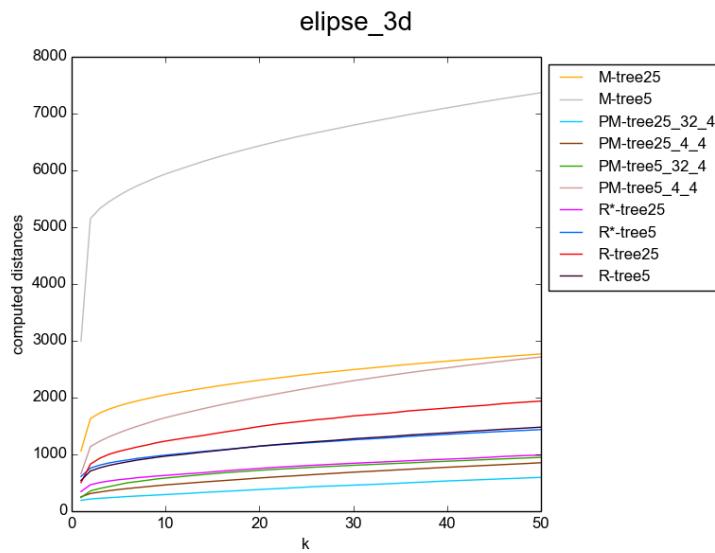
Slika B.15: Povprečno število računanj razdalj na množici elipsa2d v odvisnosti od števila iskanih elementov k.



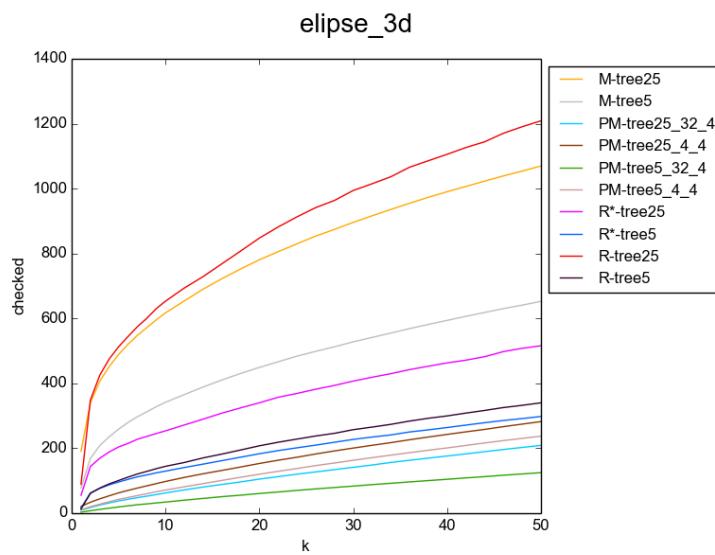
Slika B.16: Povprečno število pregledanih točk v listih na množici elipsa2d v odvisnosti od števila iskanih elementov  $k$ .



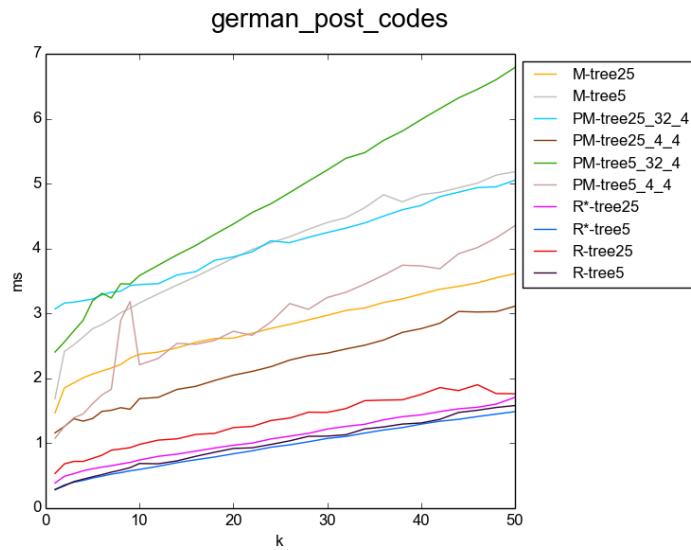
Slika B.17: Rezultati časovnih meritev python implementacij na množici elipsa3d v odvisnosti od števila iskanih elementov  $k$ .



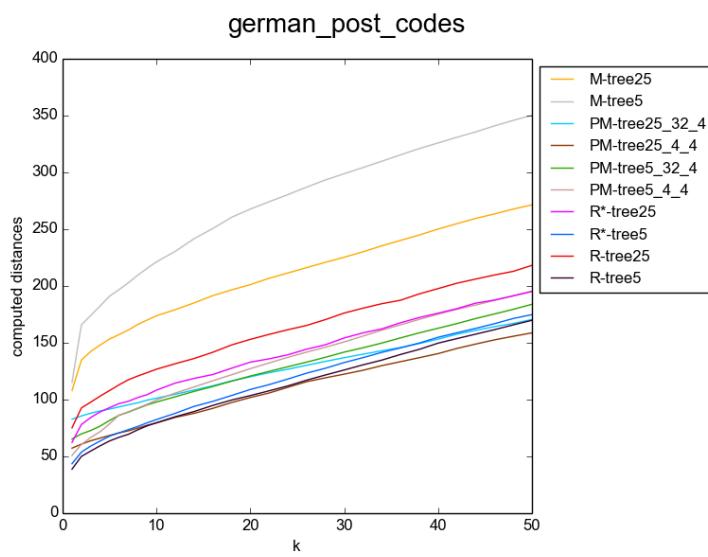
Slika B.18: Povprečno število računanj razdalj na množici elipsa3d v odvisnosti od števila iskanih elementov k.



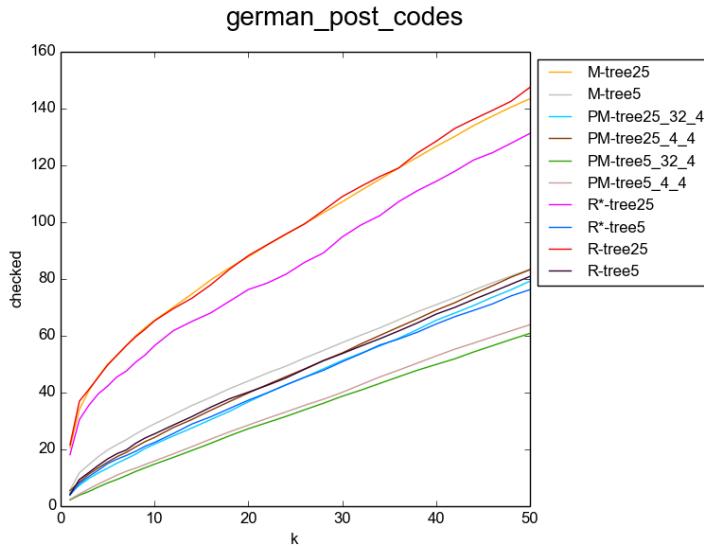
Slika B.19: Povprečno število pregledanih točk v listih na množici elipsa3d v odvisnosti od števila iskanih elementov k.



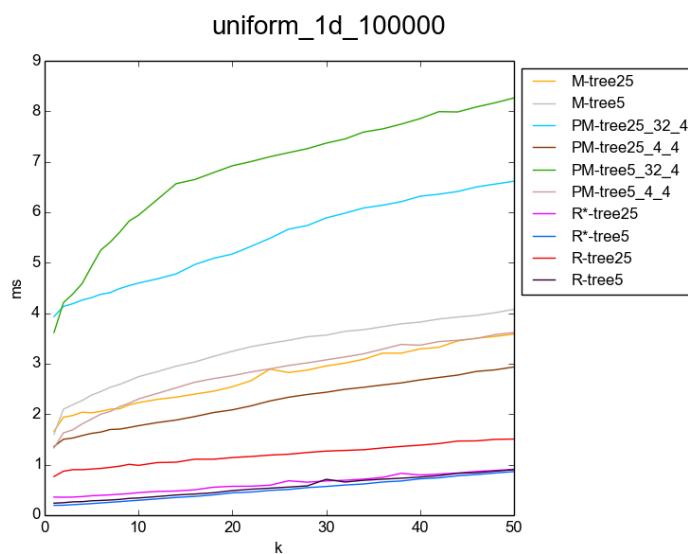
Slika B.20: Rezultati časovnih meritev python implementacij na množici NPŠ v odvisnosti od števila iskanih elementov k.



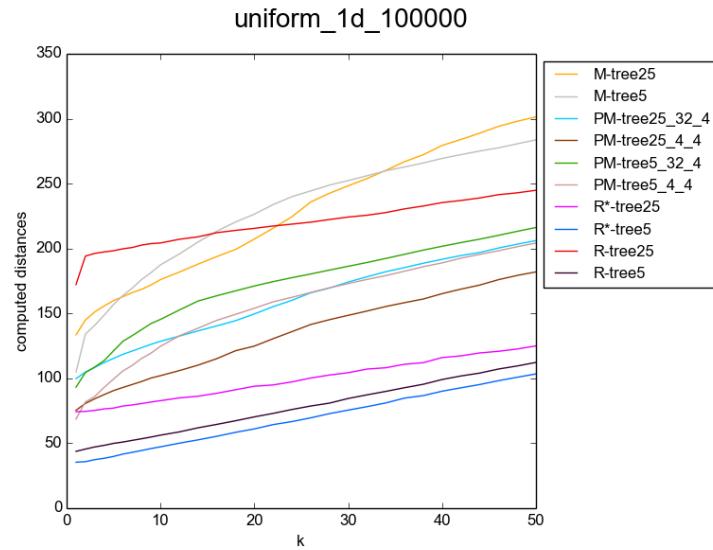
Slika B.21: Povprečno število računanj razdalj na množici NPŠ v odvisnosti od števila iskanih elementov k.



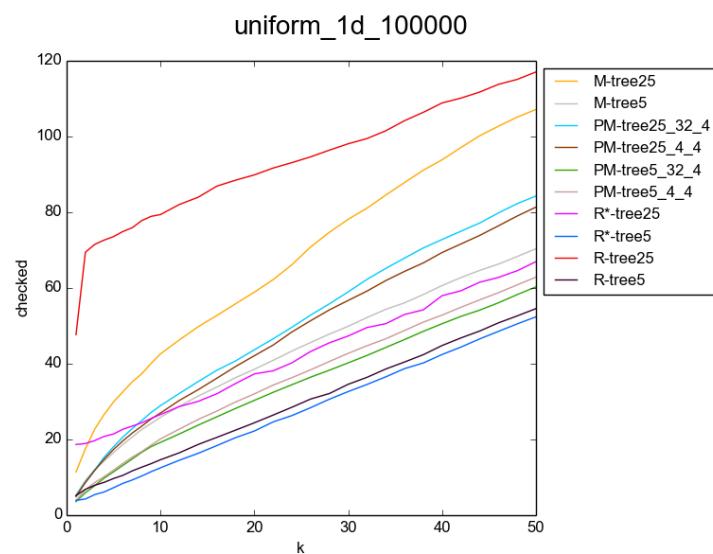
Slika B.22: Povprečno število pregledanih točk v listih na množici NPŠ v odvisnosti od števila iskanih elementov k.



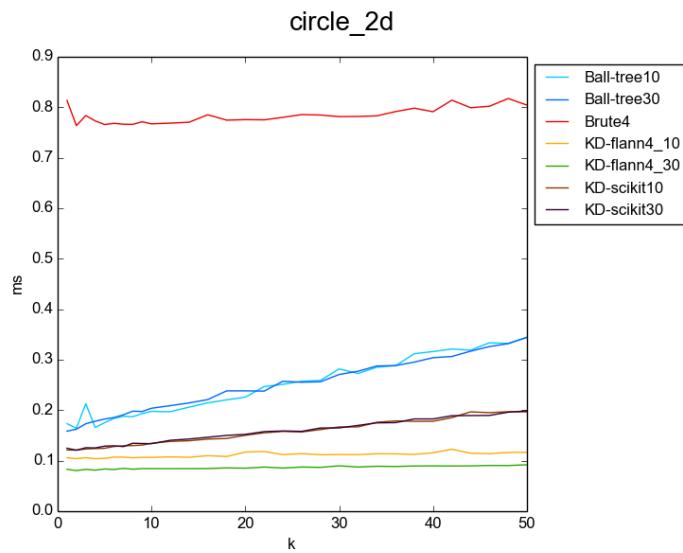
Slika B.23: Rezultati časovnih meritev python implementacij na množici uniform1d v odvisnosti od števila iskanih elementov k.



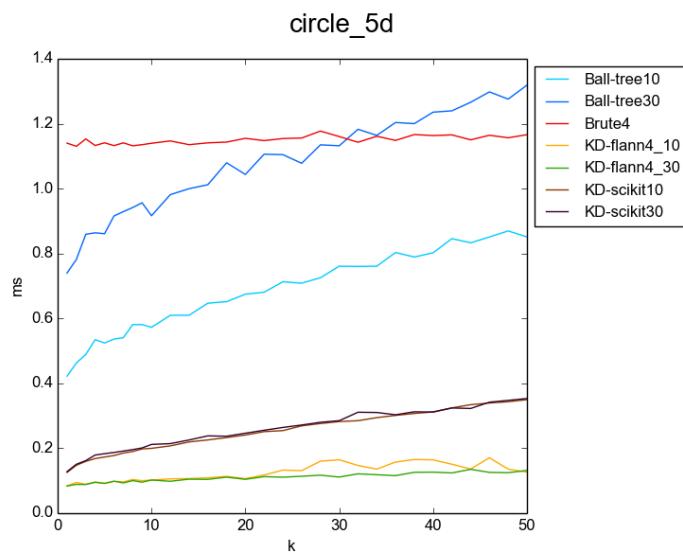
Slika B.24: Povprečno število računanj razdalj na množici uniform1d v odvisnosti od števila iskanih elementov k.



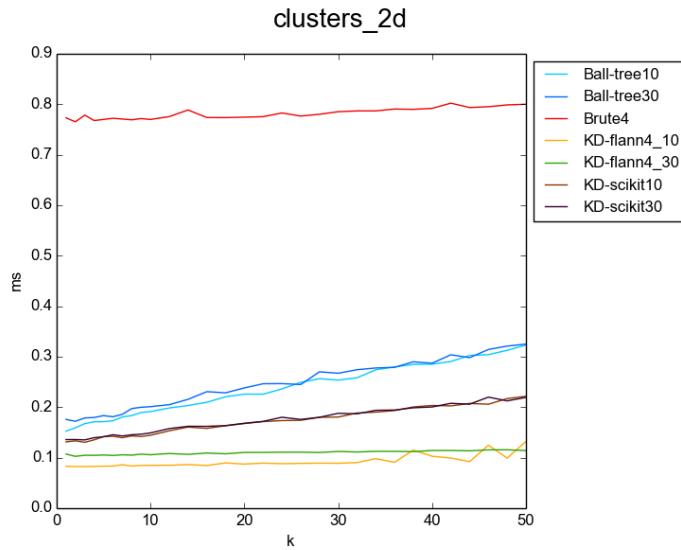
Slika B.25: Povprečno število pregledanih točk v listih na množici uniform1d v odvisnosti od števila iskanih elementov k.



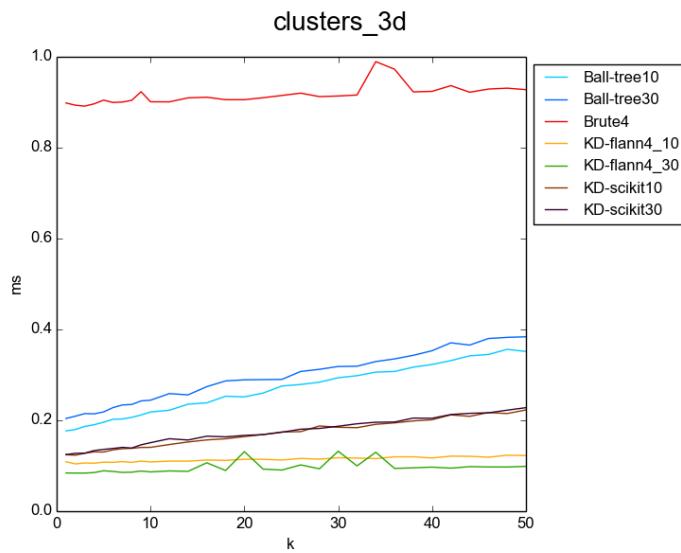
Slika B.26: Rezultati časovnih meritev C++ implementacij na množici krog2d v odvisnosti od števila iskanih elementov  $k$ .



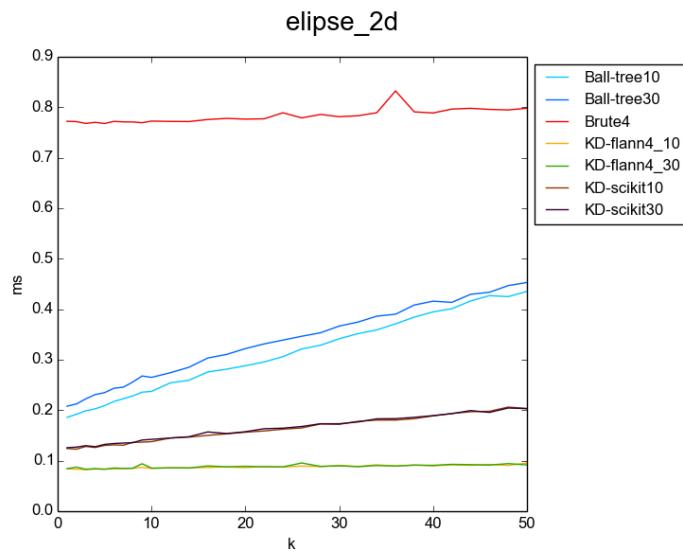
Slika B.27: Rezultati časovnih meritev C++ implementacij na množici krog5d v odvisnosti od števila iskanih elementov  $k$ .



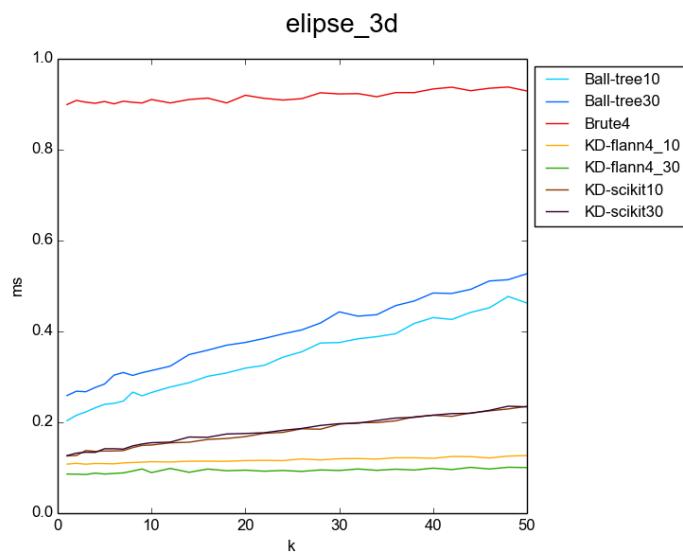
Slika B.28: Rezultati časovnih meritev C++ implementacij na množici gruče2d v odvisnosti od števila iskanih elementov k.



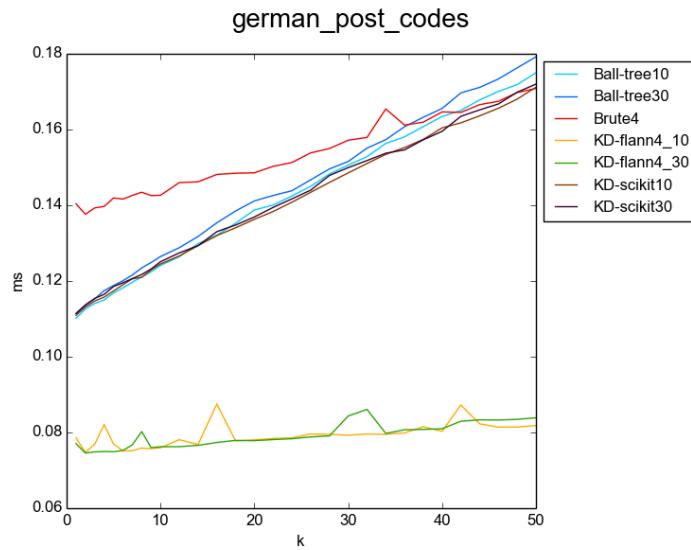
Slika B.29: Rezultati časovnih meritev C++ implementacij na množici gruče3d v odvisnosti od števila iskanih elementov k.



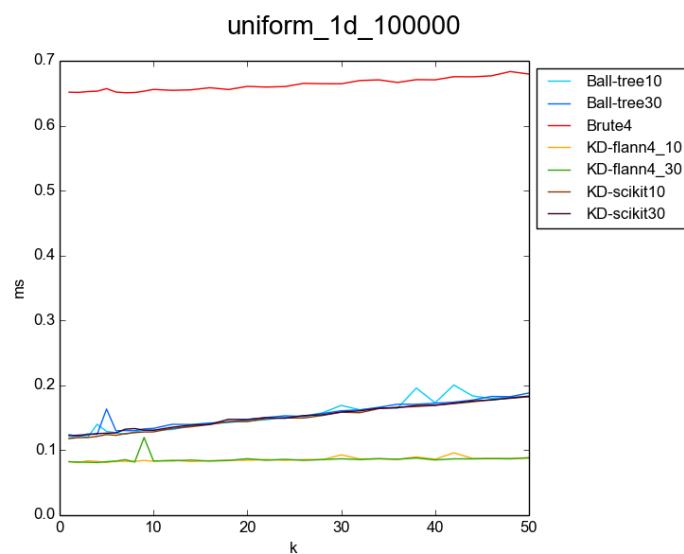
Slika B.30: Rezultati časovnih meritev C++ implementacij na množici elipsa2d v odvisnosti od števila iskanih elementov k.



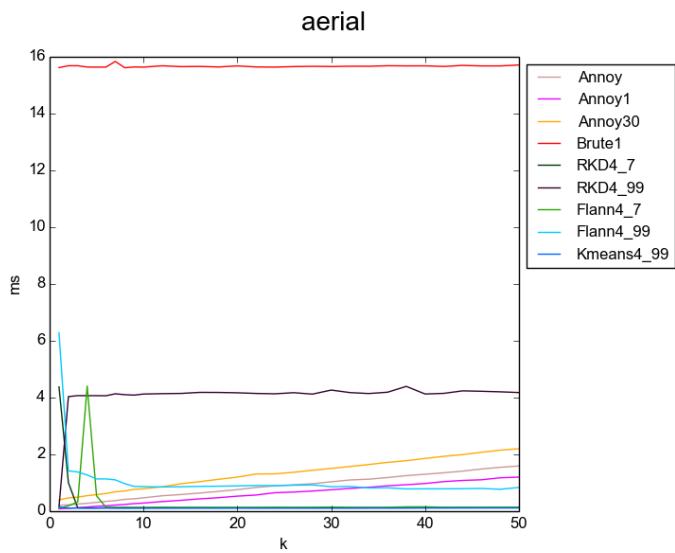
Slika B.31: Rezultati časovnih meritev C++ implementacij na množici elipsa3d v odvisnosti od števila iskanih elementov k.



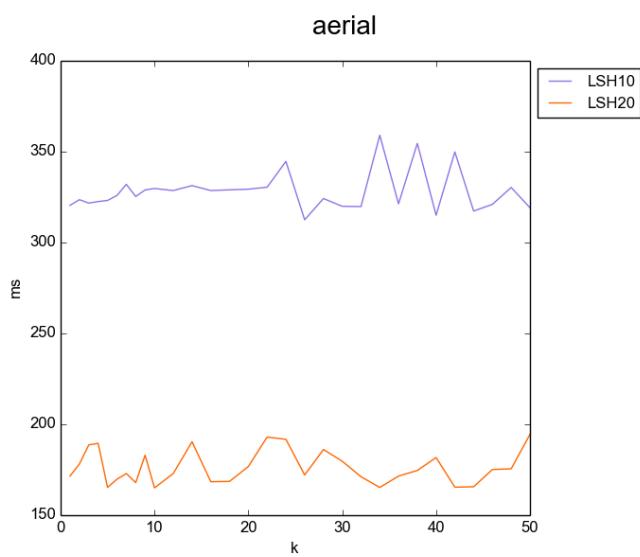
Slika B.32: Rezultati časovnih meritev C++ implementacij na množici NPŠ v odvisnosti od števila iskanih elementov  $k$ .



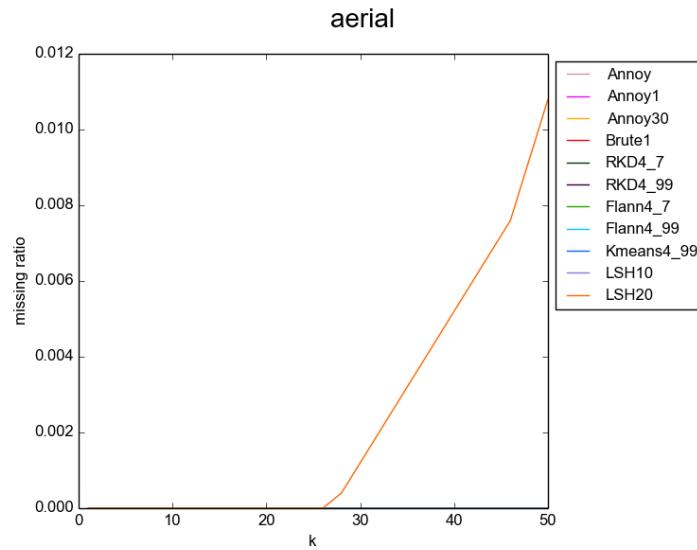
Slika B.33: Rezultati časovnih meritev C++ implementacij na množici uniform1d v odvisnosti od števila iskanih elementov  $k$ .



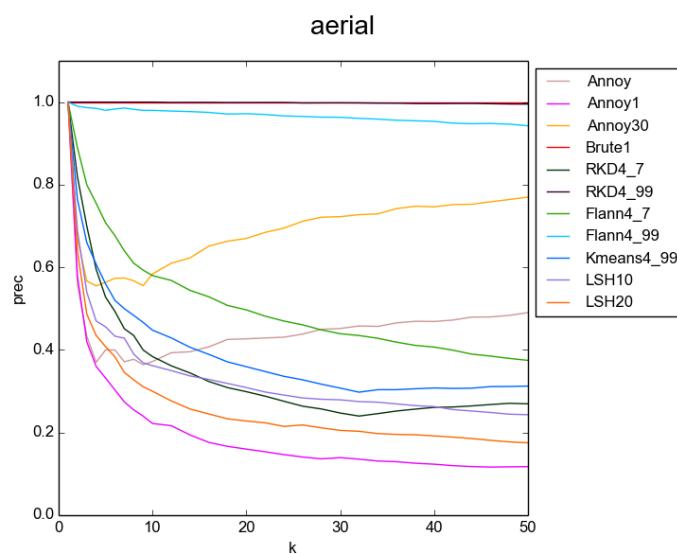
Slika B.34: Rezultati časovnih meritev C++ implementacij na množici aerial v odvisnosti od števila iskanih elementov  $k$ .



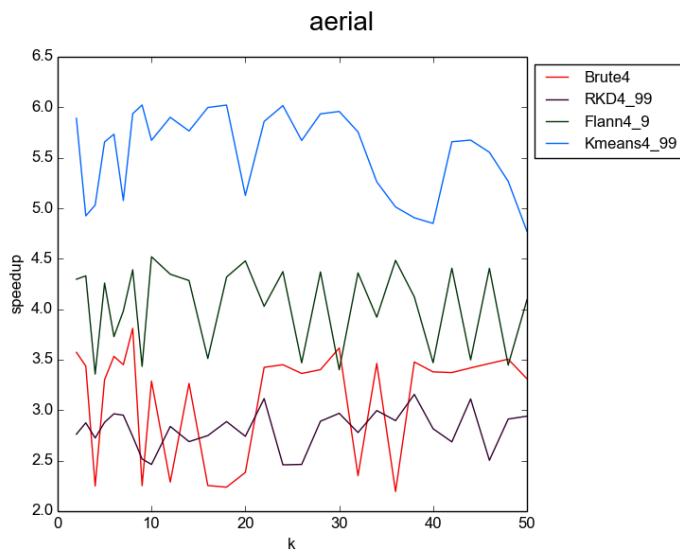
Slika B.35: Rezultati časovnih meritev implementacij v programskejem jeziku python na množici aerial v odvisnosti od števila iskanih elementov  $k$ .



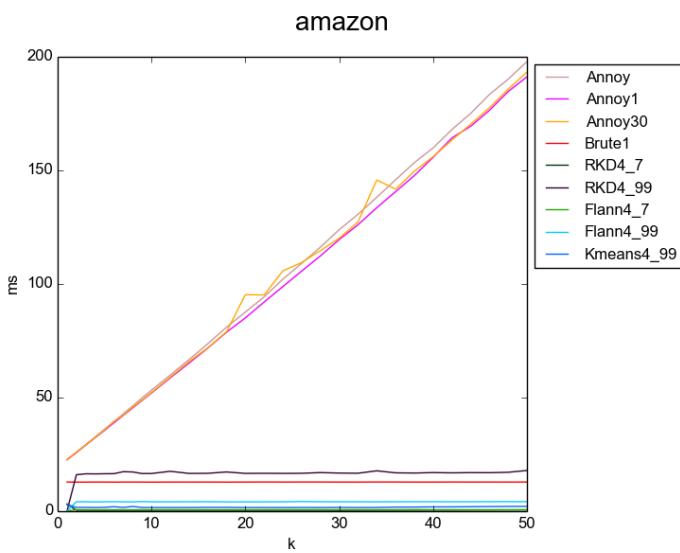
Slika B.36: Graf deleža manjkajočih vrnjenih sosedov metod na množici aerial v odvisnosti od števila iskanih elementov k.



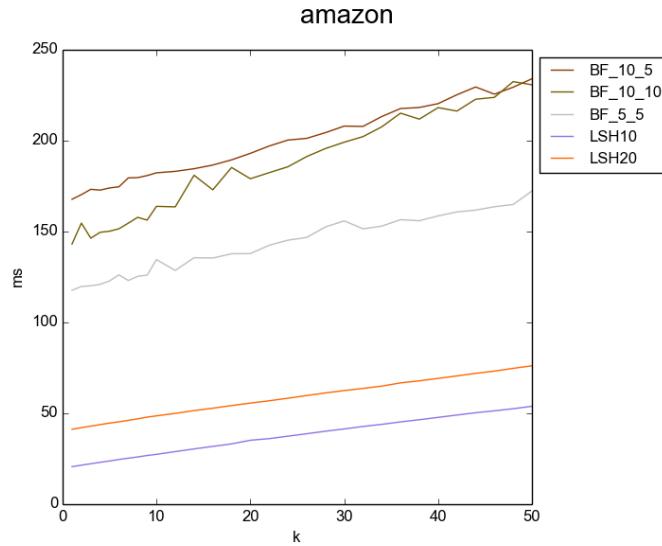
Slika B.37: Rezultati točnosti metod na množici aerial v odvisnosti od števila iskanih elementov k.



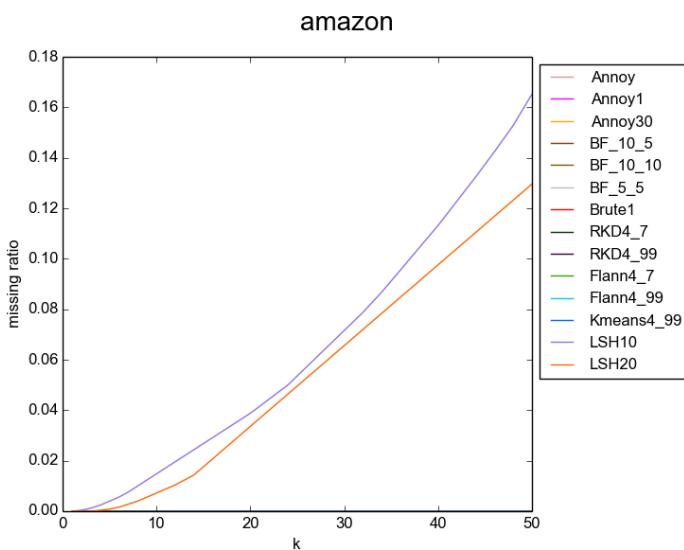
Slika B.38: Graf pohitritve metod pri uporabi več jader na množici aerial v odvisnosti od števila iskanih elementov k.



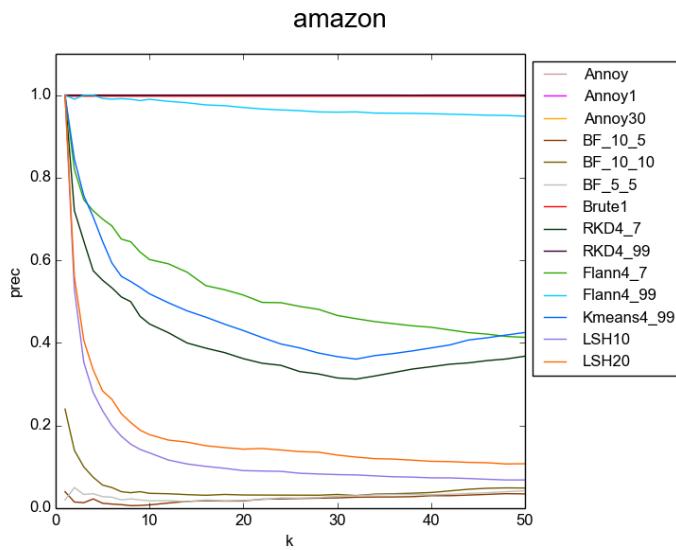
Slika B.39: Rezultati časovnih meritev C++ implementacij na množici amazon v odvisnosti od števila iskanih elementov k.



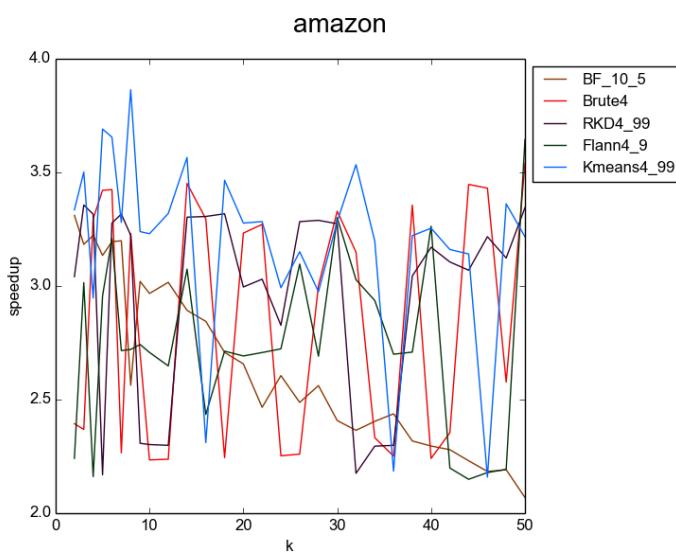
Slika B.40: Rezultati časovnih meritev implementacij v programskej jeziku python na množici amazon v odvisnosti od števila iskanih elementov k.



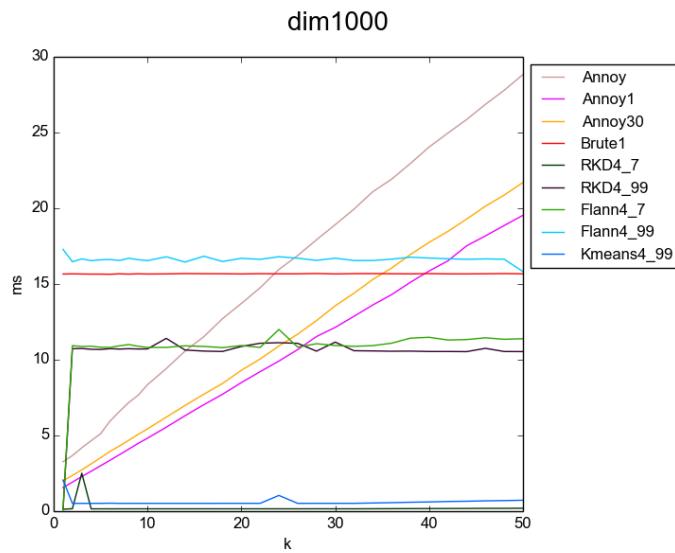
Slika B.41: Graf deleža manjka jočih vrnjenih sosedov metod na množici amazon v odvisnosti od števila iskanih elementov k.



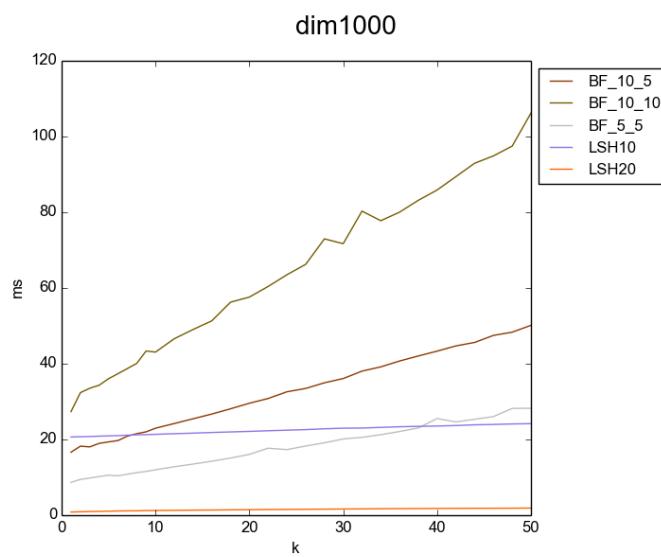
Slika B.42: Rezultati točnosti metod na množici amazon v odvisnosti od števila iskanih elementov k.



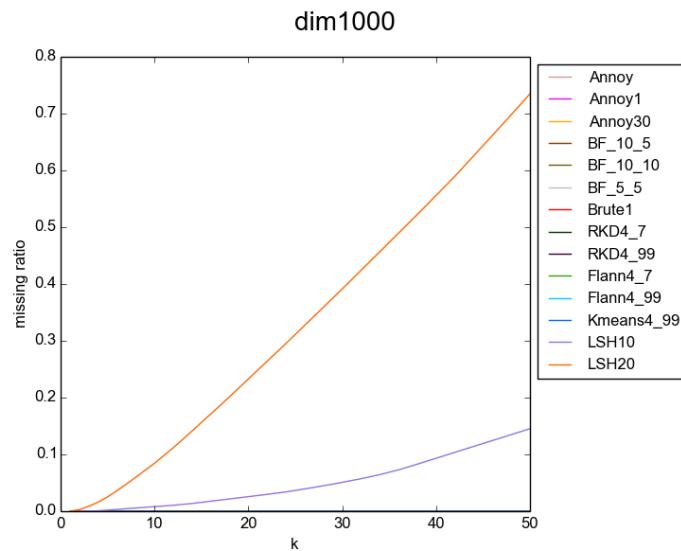
Slika B.43: Graf pohitritve metod pri uporabi več jeder na množici amazon v odvisnosti od števila iskanih elementov k.



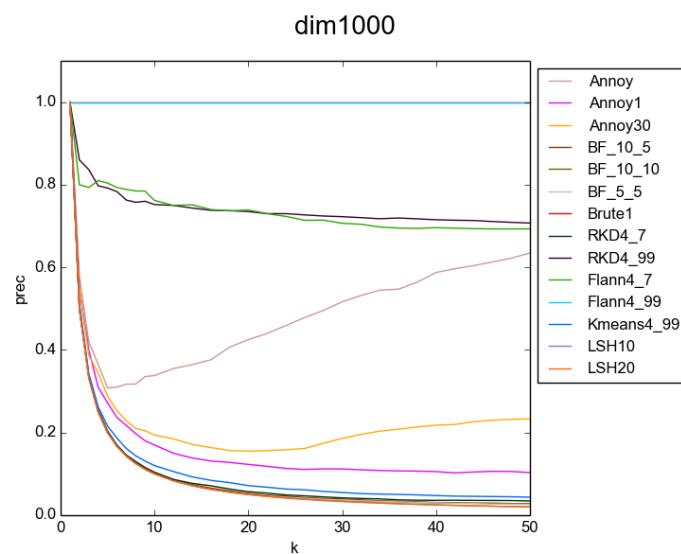
Slika B.44: Rezultati časovnih meritev C++ implementacij na množici dim1000 v odvisnosti od števila iskanih elementov k.



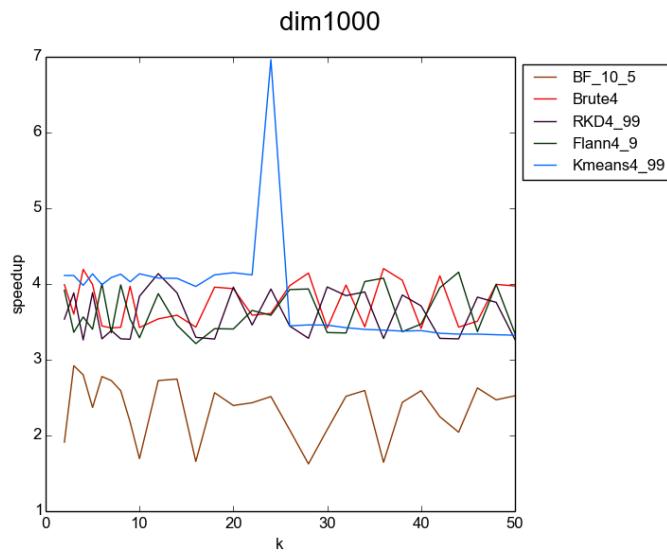
Slika B.45: Rezultati časovnih meritev implementacij v programskejem jeziku python na množici dim1000 v odvisnosti od števila iskanih elementov k.



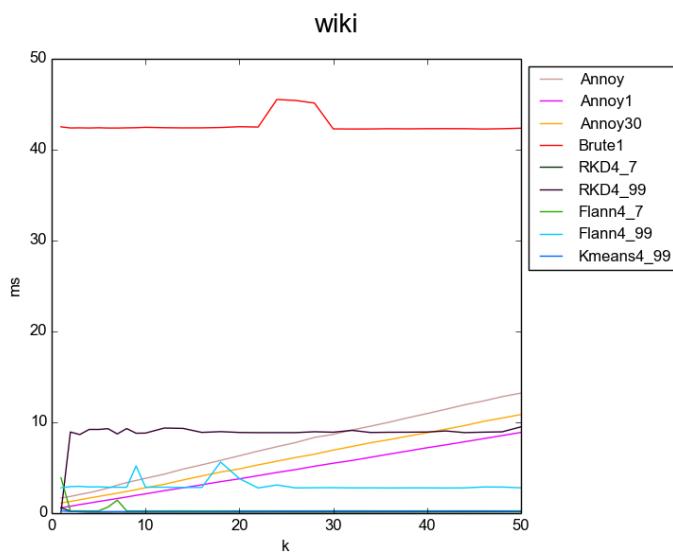
Slika B.46: Graf deleža manjkajočih vrnjenih sosedov metod na množici dim1000 v odvisnosti od števila iskanih elementov k.



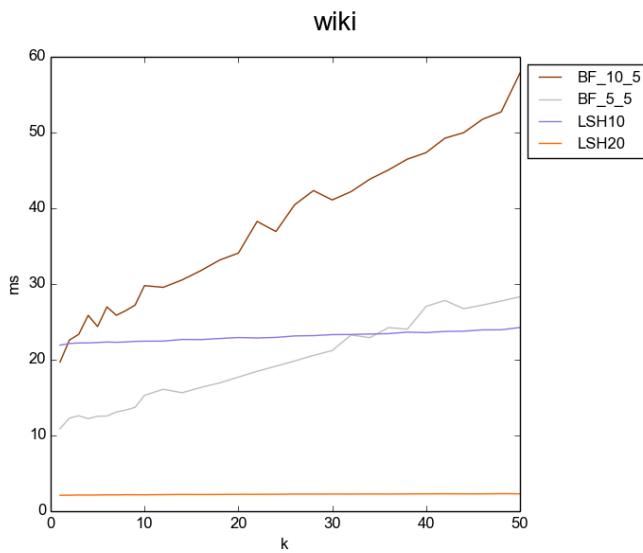
Slika B.47: Rezultati točnosti metod na množici dim1000 v odvisnosti od števila iskanih elementov k.



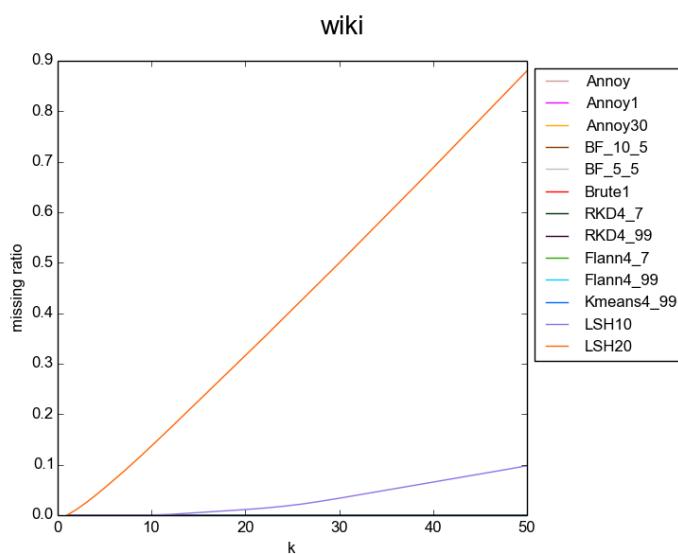
Slika B.48: Graf pohitritve metod pri uporabi več jader na množici dim1000 v odvisnosti od števila iskanih elementov k.



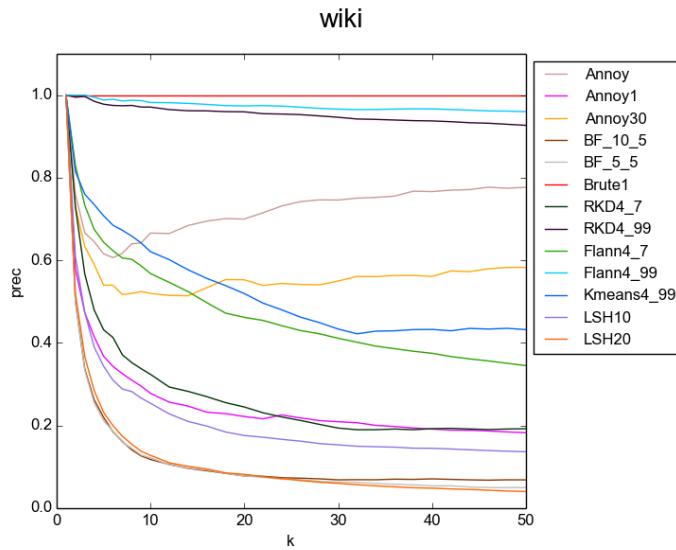
Slika B.49: Rezultati časovnih meritev C++ implementacij na množici wiki v odvisnosti od števila iskanih elementov k.



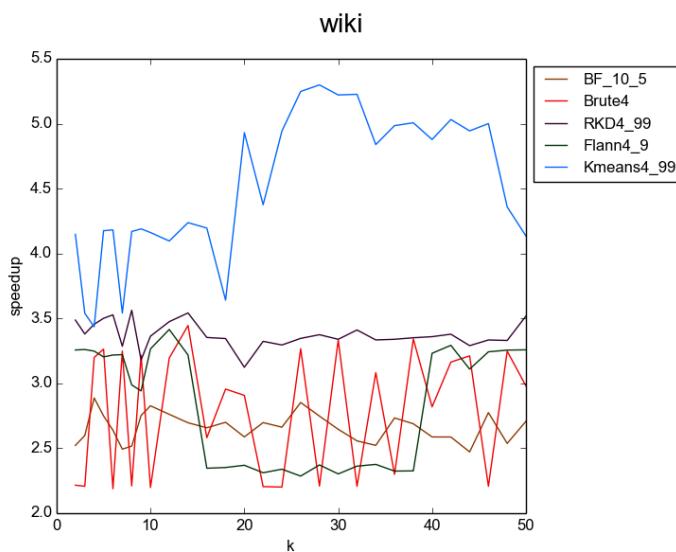
Slika B.50: Rezultati časovnih meritev implementacij v programskem jeziku python na množici wiki v odvisnosti od števila iskanih elementov k.



Slika B.51: Graf deleža manjkajočih vrnjenih sosedov metod na množici wiki v odvisnosti od števila iskanih elementov k.



Slika B.52: Rezultati točnosti metod na množici wiki v odvisnosti od števila iskanih elementov  $k$ .



Slika B.53: Graf pohitritve metod pri uporabi več jeder na množici wiki v odvisnosti od števila iskanih elementov  $k$ .



# Dodatek C

Dodatek vsebuje primer uporabe naše knjižnice za iskanje najbližjih sosedov na množici NPŠ z R-drevesom in tabelo z navedenimi parametri metod posameznih struktur.

---

**Algoritem 10** Primer uporabe R-drevesa

---

```
1: from nnsearch.datasets import load_dataset
2: from nnsearch.exact import RTree
3: dataset = load_dataset("german_post_codes")
4: #create and build r-tree
5: rtree = RTree()
6: rtree.build(data=dataset)
7: #plot r-tree
8: rtree.plot()
9: #query single point
10: query = dataset.data[0]
11: neighbours, distances = rtree.query(query, k=5)
12: print "neighbours:", neighbours
13: print "distances:", distances
```

---

Tabela C.1: Prikaz parametrov metod posameznih struktur.

Struktura	Metoda	Parametri metode
<b>FlannAuto</b>	build query save load	data, precision queries, k filename filename, data
<b>K-means</b>	build query save load	data, branching, iterations, centers_init, cb_index, precision queries, k filename filename, data
<b>RKD-tree</b>	build query save load	data, trees, precision queries, k filename filename, data
<b>LSHFlann</b>	build query save load	data, nr_tables, key_size, multi_probe_level queries, k filename filename, data
<b>LSHNearPy</b>	build insert query candidate_count save load	data, lshashes, distance, vector_filters, storage vector, data queries, k, return_data vector filename filename
<b>Annoy</b>	build insert query	data, distance, trees vector queries, k
Tabela se nadaljuje na naslednji strani		

Tabela C.1 – nadaljevanje tabele iz prejšnje strani

Struktura	Metoda	Parametri metode
<b>BoundaryF</b>	save	filename
	load	filename, dimensions, distance
	build	data, trees, max_node_size, parallel
	insert	vector
	query	queries, k
<b>Ball-tree</b>	save	filename
	load	filename, data, distance
<b>KDS-tree</b>	build	data, leaf_size, distance
	query	queries, k
<b>KDF-tree</b>	build	data, leaf_size, distance
	query	queries, k
	save	filename
	load	filename, data
<b>PM-tree</b>	build	data, max_node_size, p, nhr, npd, distance, mink_p, promote_fn, partition_fn, nr_pivot_groups
	insert	entry
	query	queries, k
	save	filename
	load	filename
<b>R-tree</b>	build	data, min_node_size, max_node_size, split_method
	insert	entry
	query	queries, k, mink_p
	delete	entry
	plot	filename, marker_size, height

Tabela se nadaljuje na naslednji strani

**Tabela C.1 – nadaljevanje tabele iz prejšnje strani**

<b>Struktura</b>	<b>Metoda</b>	<b>Parametri metode</b>
<b>R*-tree</b>	save	filename
	load	filename
	build	data, min_node_size, max_node_size
	insert	entry
	query	queries, k, mink_p
	delete	entry
	plot	filename, marker_size, height
<b>Brute</b>	save	filename
	load	filename
	build	data
	query	queries, k