Τμήμα Μηχανικών Η/Υ και Πληροφορικής
Πολυτεχνική Σχολή

Πολυδιάστατες Δομές Δεδομένων και Υπολογιστική Γεωμετρία

Ακαδημαϊκό έτος 2020-2021

Εργασία 5

K-D trees, Quad Trees και 2D Range Trees: Υλοποίηση 2D kNN Queries με τις παραπάνω τρεις δομές και πειραματική σύγκριση.

Γεωργόπουλος - Νίνος Νικόλαος ΑΜ: 1054385 Έτος: 5ο (gninos@ceid.upatras.gr)

Νάκκας Νικόλαος ΑΜ: 1054359 Έτος: 5ο (nakkas@ceid.upatras.gr)

Παπάκος Μιλτιάδης ΑΜ:1054360 Έτος: 5ο (papakos@ceid.upatras.gr)

Συροκώστας Κωνσταντίνος ΑΜ: 1054339 Έτος: 5ο (syrokostas@ceid.upatras.gr)

Εισαγωγή

Το Dataset το πήραμε από την ιστοσελίδα https://www.kaggle.com/c/facebook-v-predicting-check-ins/overview σύμφωνα με την οποία το Facebook και η Kaggle ξεκίνησαν έναν διαγωνισμό μηχανικής μάθησης για το 2016.

Ο στόχος αυτού του διαγωνισμού είναι να προβλέψει σε ποιο μέρος θα ήθελε να κάνει check in ένα άτομο. Για τους σκοπούς αυτού του διαγωνισμού, το Facebook δημιούργησε έναν τεχνητό κόσμο που αποτελείται από περισσότερα από 100.000 μέρη που βρίσκονται σε μια πλατεία 10 χλμ. Χ 10 χλμ. Για ένα δεδομένο σύνολο συντεταγμένων, ο στόχος σας είναι να επιστρέψει μια λίστα με τις πιο πιθανές θέσεις.

Τα δεδομένα κατασκευάστηκαν για να μοιάζουν με σήματα τοποθεσίας που προέρχονται από κινητές συσκευές, δίνοντάς μια γεύση για το τι χρειαζόμαστε ώστε να εργαστούμε με πραγματικά δεδομένα που περιπλέκονται από ανακριβείς και θορυβώδεις τιμές.

Περιγραφή αρχείων

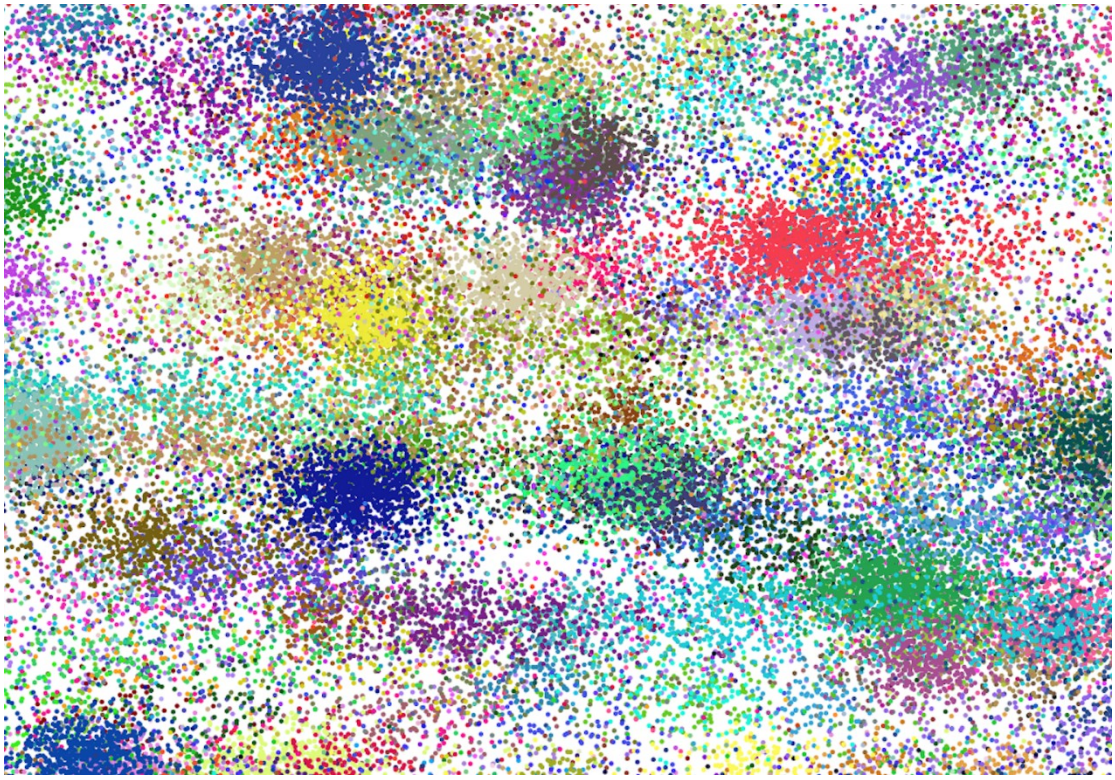
- train.csv, test.csv
  - row_id: id του check-in event
  - x y: συντεταγμένες
  - accuracy: ακρίβεια της τοποθεσίας
  - time: χρόνικη σήμανση
  - place_id: id της επιχείρησης

Στο παρακάτω αρχείο έχουμε αφαιρέσει όλες τις περιττές στήλες και έχουμε κρατήσει μόνο τις συντεταγμένες.

- train_x_y.csv
  - x y: συντεταγμένες

Και τα τρία παραπάνω αρχεία περιέχουν 29.118.021 σημεία (δηλαδή 29.118.021 rows).

Υπάρχει και ένα αρχείο το train_x_y_10K.csv το οποίο αποτελείται από 10.000 σημεία και πάνω σε αυτό έγιναν οι αρχικές δοκιμές.



Στην παραπάνω εικόνα απεικονίζονται τα σημεία πάνω στο τετράγωνο διαστάσεων 10km.

Υλοποίηση 2D kNN Queries με K-D tree

Το K-D tree υλοποιήθηκε με 2 διαφορετικά προγράμματα.

Το 1ο: Mds_Main_solution.py
Αποτελείται από έτοιμες συναρτήσεις της sklearn και συγκεκριμένα την KDTree όπου δημιουργεί η βιβλιοθήκη αυτή το K-D tree και στη συνέχεια με την ίδια βιβλιοθήκη κάνουμε το query. Παρατηρήσαμε ότι οι χρόνοι των query είναι κακοί (της τάξης των 1 εώς 6 second). Επομένως δεν θα αναφερθούμε σε αυτά και η σύγκρισή με τα δικά μας δεν έχει κανένα νόημα.

Ουσιαστικά το πρόγραμμα αυτό βρίσκει τους K πιο κοντινούς γείτονες του csv και όχι συγκεκριμένου σημείου.

Και παίρνουμε 2 μεταβλητές:

- την nearest_dist όπου μας δίνεται η απόσταση των κ (=2 στη συγκεκριμένη περίπτβση) κοντινών γειτόνων όπως φαίνεται παρακάτω.

	0	1
0	0.00000	0.05114
1	0.00000	0.04331
2	0.00000	0.09096
3	0.00000	0.05091
4	0.00000	0.04964
5	0.00000	0.04713
6	0.00000	0.05611
7	0.00000	0.05570
8	0.00000	0.06127
9	0.00000	0.02934

- Και την nearest_ind όπου μας δείχνει τους 2 πιο κοντινούς γειτόνους και τα id τους (τους δείκτες τους)

	0	1
0	0	9529
1	1	3358
2	2	8490
3	3	3319
4	4	8352
5	5	6908
6	6	7767
7	7	5573
8	8	2022
9	9	2372

<u>Το 2ο πρόγραμμα</u> (αρχείο kdtree.py)

Αποτελεί τη βασική λύση του προβλήματος και είναι η απάντηση στο πρόβλημα. Έχει υλοποιηθεί η δομή του kd tree με την συνάρτηση make_kdtree(). Επίσης έχει υλοποιηθεί και το query με τη συνάρτηση get_knn(). Δεν χρησιμοποιείται καμία έτοιμη συνάρτηση για την υλοποίηση της δομής και του query. Η πιο σημαντική συνάρτηση που χρησιμοποιείται είναι η cProfile όπου τρέχει benchmarks για να παίρνουμε τους χρόνους που κάνει για να εκτελέσει κάθε συνάρτηση και συνολικά το πρόγραμμα.

Στο πρόγραμμα αυτό δίνεις ένα συγκεκριμένο σημείο στη μεταβλητή point_k (π.χ. (5.1, 5.1)) και δίνει μια τιμή στο K (π.χ. =10) που αποτελεί τους K πιο κοντινούς γειτόνους του συγκεκριμένου σημείου.

Παρακάτω είναι ένα παράδειγμα εκτέλεσης για με το αρχείο train_x_y_10K.csv το οποίο αποτελείται από 10.000 σημεία.

```
         59209236 function calls (59197126 primitive calls) in 15.320 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   15.319   15.319 <string>:1(<module>)
    303/1    0.001    0.000    0.001    0.001 kdtree.py:22(get_knn)
       11    0.000    0.000    0.000    0.000 kdtree.py:39(<genexpr>)
      214    0.000    0.000    0.000    0.000 kdtree.py:53(dist_sq)
      642    0.000    0.000    0.000    0.000 kdtree.py:54(<genexpr>)
      214    0.000    0.000    0.000    0.000 kdtree.py:57(dist_sq_dim)
  11809/1    0.027    0.000   15.318   15.318 kdtree.py:6(make_kd_tree)
        1    0.000    0.000   15.319   15.319 kdtree.py:75(bench)
 59040000    3.579    0.000    3.579    0.000 kdtree.py:8(<lambda>)
   119535    0.008    0.000    0.008    0.000 kdtree.py:9(<lambda>)
      303    0.000    0.000    0.000    0.000 pydev_import_hook.py:16(do_import)
       32    0.000    0.000    0.000    0.000 {built-in method _heapq.heappushpop}
       10    0.000    0.000    0.000    0.000 {built-in method _heapq.heappush}
      303    0.000    0.000    0.000    0.000 {built-in method builtins.__import__}
        1    0.000    0.000   15.320   15.320 {built-in method builtins.exec}
    23832    0.001    0.000    0.001    0.000 {built-in method builtins.len}
     5905    0.022    0.000    0.031    0.000 {built-in method builtins.sorted}
      214    0.000    0.000    0.000    0.000 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
```

Τέλος δίνεται η απόσταση μεταξύ των K=10 κοντινών σημείων από ένα συγκεκριμένο σημείο το point=(5.1, 5.1) και δεξιά δίνεται το σημείο και η απόσταση που έχει από αυτό.

```
(0.006259299999999924, [5.0289, 5.0653])
(0.026510650000000125, [5.2587, 5.0636])
(0.034960610000000336, [5.281000000000001, 5.0531])
(0.0578864799999999, [4.8598, 5.1138])
(0.05991625, [5.3397, 5.0504])
(0.06019780000000028, [5.3418, 5.1416])
(0.07461493000000002, [5.0682, 5.3713])
(0.1098487300000002, [4.9068, 5.3693])
(0.1111801300000003, [5.4147, 5.2102])
(0.11675189000000014, [5.441, 5.1217])
```

KD

K N	5	10	50	100	500
100	Close to 0	Close to 0	Close to 0	0.000998 sec	Not possible
1.000	0.001000 sec	0.000998 sec	0.001995 sec	0.002993 sec	0.004987 sec
10.000	0.002994 sec	0.003023 sec	0.004020 sec	0.005984 sec	0.009974 sec
100.000	0.009973 sec	0.010970 sec	0.010939 sec	0.012967 sec	0.019914 sec
300.000	0.016955 sec	0.022969 sec	0.019945 sec	0.021904 sec	0.033416 sec

Παρατηρώ ότι:

Για μικρότερο N (αριθμό δεδομένων) σταθερό και αυξανόμενο K έχουμε γρηγορότερη αύξηση του χρόνου του query σε σύγκριση με ένα μεγαλύτερο N και αυξανόμενο K. Και στις δύο περιπτώσεις έχουμε λογαριθμική αύξηση όμως.

Για σταθερό μικρό K Και αυξανόμενο N, έχουμε μικρότερη αύξηση του χρόνου που χρειάζεται το query σε σχέση με ένα μεγαλύτερο σταθερό K. Και στις δύο περιπτώσεις έχουμε λογαριθμική αύξηση όμως.

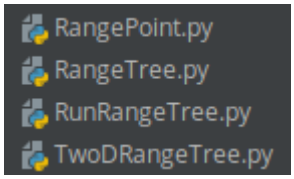
Αυτά επιβεβαιώνονται από τον εξής πίνακα που βρίσκεται στο Wikipedia entry του K-D tree :

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$

[1]

Υλοποίηση 2D kNN Queries με Range Tree

Το range tree είναι δομή που χρησιμοποιείται για querying δεδομένων, συνήθως σε μεγαλύτερες της μιας διάστασης. Σε αυτή την περίπτωση θα ασχοληθούμε με δυσδιάστατα range trees.



Στο αρχείο παρατηρούνται 6 κλάσεις. Η RunRangeTree περιέχει τον αλγόριθμο run που χρησιμοποιείται για να κληθεί η getknn στη main.

Η RangeTree περιέχει τις κλάσεις RangeLeaf και RangeNode αποτελούν δομικά στοιχεία των δέντρων που υλοποιούμαι καθώς και την κλάση RangeTree που αποτελεί τη δομή μας σε μια διάσταση.

Η TwoDRangeTree είναι η υλοποίηση της δομής σε 2 διαστάσεις. Μέσα σε αυτή βρίσκεται και ο αλγόριθμος για την εύρεση των k κοντινότερων γειτόνων.

Ο λόγος που έχουμε υλοποίηση σε μία αλλά και δύο διαστάσεις είναι ότι για range queries του X έχουμε όντως δισδιάστατο range δένδρο αλλά αναζητώντας την μεταβλητή Y του κάθε X, έχουμε να κάνουμε με πολλά μονοδιάστατα query.

Το challenge μας με αυτή την υλοποίηση είναι ότι όχι μόνο δεν υπήρχε αλγόριθμος για queries με τη χρήση RangeTrees για σύγκριση των χρόνων μας, αλλά ούτε καν η δομή δεδομένων ώστε να αποτελέσει τη βάση για τα queries.

Παρόλα αυτά καταλήξαμε σε μια ικανοποιητική υλοποίηση που έδωσε σωστά αποτελέσματα και τους εξής χρόνους :

```
(4.2708 5.143)
(5.9567 4.7968)
(4.5027 4.36)
(4.5698 4.1206)
(4.8707 6.2239)
```

Που είναι και οι σωστοί γείτονες.

Range Algorithm

K N	5	10	50	100	500
100	0.000141 sec	0.000178 sec	0.000407 sec	0.000614 sec	**Not possible**
1.000	0.000996sec	0.001285 sec	0.008347 sec	0.010647sec	0.012929sec
10.000	0.012966sec	0.011971sec	0.007979sec	0.012019sec	0.978072sec
100.000	0.061836sec	0.059870sec	0.110181sec	0.111702sec	0.258473sec
300.000	0.17318 sec	0.170661 sec	0.16828 sec	0.17191 sec	0.308912 sec

Παρατηρώ ότι έχω απότομη λογαριθμική αύξηση όσο αυξάνεται το N με σταθερό K και γραμμική για τα μεγαλύτερα K όσο αυξάνεται το K με σταθερό N. Αυτό γίνεται λόγω του τύπου :

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n \log^{d-1} n)$	$O(n \log^{d-1} n)$
Search	$O(\log^d n + k)$	$O(\log^d n + k)$

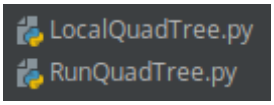
[2]

Όπου βλέπω ότι για μεγαλύτερα K επικρατεί ο αριθμός τον γειτόνων και συνεπώς η γραμμική πολυπλοκότητα του Search. Παρατηρούμε λοιπόν ότι ο χρόνος επηρεάζεται μεν από την αύξηση του αριθμού των γειτόνων, όσο και από τον αριθμό τον δεδομένων. Δίνοντας βάση στον αριθμό δεδομένων, όπως θα παρατηρηθεί και στη συνέχεια, η αύξηση αυτή είναι τόσο απότομη, που θα μας οδηγήσει αναγκαστικά είτε σε επιπλέον βελτιστοποίησή του, είτε σε επιλογή εναλλακτικού αλγόριθμου για ένα πιο ρεαλιστικό σενάριο όπου πολύ περισσότερα δεδομένα θα είναι αναγκαίο να είναι μέρος των υπολογισμών μας.

Υλοποίηση 2D kNN Queries με Quad tree

Το Quad Tree είναι μία δομή η στην οποία κάθε κόμβος έχει ακριβώς 4 παιδιά. Η δομή χρησιμοποιείται για την αποθήκευση δισδιάστατων δεδομένων και χωρίζω αναδρομικά το επίπεδο σε 4 κομμάτια (upper-left, upper-right, lower-left, lower-right) τα οποία αποτελούν και τα παιδιά του κάθε κόμβου. Τα δεδομένα που θέλουμε να αποθηκεύσουμε στη δομή βρίσκονται στα φύλλα του δέντρου.

Για να δουλέψουμε με τη συγκεκριμένη δομή χρησιμοποιήσαμε τη βιβλιοθήκη quads για την Python η οποία μας προσέφερε μία έτοιμη υλοποίηση για τη δομή. Επιπλέον η βιβλιοθήκη περιέχει έναν αλγόριθμο για την εύρεση των k κοντινότερων γειτόνων. Ωστόσο εμείς υλοποιήσαμε τον δικό μας αλγόριθμο και συγκρίναμε τα αποτελέσματα μας καθώς και τον χρόνο εκτέλεσης με τον αλγόριθμο της βιβλιοθήκης.



Για την υλοποίηση του αλγορίθμου δημιουργήσαμε τα 2 παραπάνω αρχεία.

Το αρχείο LocalQuadTree περιέχει την κλάση LocalQuadTree η οποία κληρονομεί την κλάση QuadTree της βιβλιοθήκης quads. Μέσα σε αυτή υλοποιήσαμε τον αλγόριθμος για εύρεση των k κοντινότερων γειτόνων. Επιπλέον υλοποιήσαμε έναν αλγόριθμο για αναζήτηση στο δέντρο, τον οποίο χρειαζόμαστε για την υλοποίηση του αλγορίθμου των kNN.

Το αρχείο RunQuadTree περιέχει μία συνάρτηση που τρέχει το αλγόριθμο για εύρεση k κοντινότερων γειτόνων (τη δική μας υλοποίηση αλλά και αυτή της βιβλιοθήκης) και τυπώνει τους κοντινότερους γείτονες και τον χρόνο εκτέλεσης.

Από την εκτέλεση του αλγορίθμου προκύπτουν οι ακόλουθοι χρόνοι εκτέλεσης.

Για τον αλγόριθμο που υλοποιήσαμε:

K N	5	10	50	100	500
100	0.000224 sec	0.000411 sec	0.001481 sec	0.001693 sec	**Not possible**
1.000	0.000188 sec	0.000373 sec	0.002037 sec	0.013004 sec	0.106455 sec
10.000	0.000235 sec	0.000487 sec	0.006906 sec	0.006623 sec	0.060348 sec
100.000	0.000242 sec	0.000329 sec	0.001533 sec	0.018279 sec	0.305588 sec
300.000	0.000482 sec	0.001044 sec	0.001342 sec	0.008498 sec	0.168473 sec

Για τον αλγόριθμο της βιβλιοθήκης:

K N	5	10	50	100	500
100	0.000202 sec	0.000365 sec	0.000525 sec	0.000552 sec	**Not possible**
1.000	0.00019 sec	0.000371 sec	0.001621 sec	0.003805 sec	0.004665 sec
10.000	0.000397 sec	0.000437 sec	0.002356 sec	0.004022 sec	0.015056 sec
100.000	0.000236 sec	0.000319 sec	0.001037 sec	0.005296 sec	0.017204 sec
300.000	0.00105 sec	0.001116 sec	0.000886 sec	0.002212 sec	0.0172 sec

Παρατηρώ ότι και οι 2 αλγόριθμοι είναι λογαριθμικοί ως προς N, οπότε ακόμα και αν τα δεδομένα αυξηθούν πολύ ο χρόνος που χρειάζεται ο αλγόριθμος επηρεάζεται πολύ λίγο.

Ωστόσο παρατηρώ και ότι ο αλγόριθμος είναι γραμμικός ως προς k που σημαίνει ότι μεγάλη αύξηση του αριθμού των γειτόνων θα οδηγήσει σε μεγάλη αύξηση του χρόνο του αλγορίθμου.

Επιπλέον γνωρίζουμε ότι για το Quad Tree μπορούμε να διασφαλίσουμε την χρονική πολυπλοκότητα μόνο για δεδομένα που ακολουθούν ομοιόμορφη κατανομή (όπως αυτά που έχουμε εδώ). Όμως αν τα δεδομένα ακολουθούσαν κάποιον power law οι παραπάνω παρατηρήσεις δεν θα ίσχυαν.

Έτσι καταλήγουμε στο ότι τα Quad Trees θα ήταν μία καλή δομή για περιπτώσεις που έχουμε πολλά δεδομένα, αλλά θέλουμε να βρούμε λίγους κοντινότερους γείτονες αρκεί τα δεδομένα να είναι ακολουθούν μία ομοιόμορφη κατανομή.

Τέλος αξίζει να σημειωθεί πως υπάρχουν περιπτώσεις (λίγες βέβαια) όπου ο αλγόριθμος της βιβλιοθήκης δεν δουλεύει σωστά, ενώ ο δικός μας, μας επιστρέφει τα σωστά σημεία. Μία τέτοια περίπτωση είναι η παρακάτω με 100.000 στοιχεία και k=4.

```
Time for library based algorithm is:
0.000248sec
4(Quad tree) nearest neighbours are:
[<Point: (5.1141, 5.0915)>, <Point: (5.0784, 5.0884)>, <Point: (5.0732, 5.0861)>]
-----------------
Time for our own V2 based algorithm is:
0.000394sec
4our (Quad tree) nearest neighbours are:
[<Point: (5.1141, 5.0915)>, <Point: (5.0784, 5.0884)>, <Point: (5.0732, 5.0861)>, <Point: (5.1015, 5.0659)>]
```

Πειραματική σύγκριση αλγορίθμων

Quad Library

K N	5	10	50	100	500
100	0.000202 sec	0.001000 sec	0.000995 sec	0.000998 sec	**Not possible**
1.000	0.000997 sec	0.000371 sec	0.000999 sec	0.003987 sec	0.004987 sec
10.000	0.000397 sec	0.000998 sec	0.002992 sec	0.002991 sec	0.013964 sec
100.000	0.000236 sec	0.000319 sec	0.000997 sec	0.003990 sec	0.018916 sec
300.000	0.000998 sec	0.000997 sec	0.000998 sec	0.003990 sec	0.014961 sec

Quad Algorithm

K N	5	10	50	100	500
100	0.000224 sec	0.000411 sec	0.001481 sec	0.002987 sec	**Not possible**
1.000	0.000188 sec	0.000373 sec	0.001993 sec	0.012000 sec	0.169514 sec
10.000	0.000235 sec	0.000487 sec	0.006980 sec	0.007979 sec	0.090728 sec
100.000	0.001001 sec	0.000986 sec	0.001994 sec	0.025441 sec	0.448187 sec
300.000	0.000482 sec	0.001044 sec	0.001342 sec	0.008498 sec	0.168473 sec

Range Algorithm

K N	5	10	50	100	500
100	0.000141 sec	0.000178 sec	0.000407 sec	0.000614 sec	**Not possible**
1.000	0.000996 sec	0.001285 sec	0.008347 sec	0.010647 sec	0.012929 sec
10.000	0.012966 sec	0.011971 sec	0.007979 sec	0.012019 sec	0.978072 sec
100.000	0.061836 sec	0.059870 sec	0.110181 sec	0.111702 sec	0.258473 sec
300.000	0.17318 sec	0.170661 sec	0.16828 sec	0.17191 sec	0.308912 sec

KD

K N	5	10	50	100	500
100	Close to 0	Close to 0	Close to 0	0.000998 sec	**Not possible**
1.000	0.001000 sec	0.000998 sec	0.001995 sec	0.002993 sec	0.004987 sec
10.000	0.002994 sec	0.003023 sec	0.004020 sec	0.005984 sec	0.009974 sec
100.000	0.009973 sec	0.010970 sec	0.010939 sec	0.012967 sec	0.019914 sec
300.000	0.016955 sec	0.022969 sec	0.019945 sec	0.021904 sec	0.033416 sec

Στην συνέχεια μετά την ατομική ανάλυση των χρόνων και γενικών παρατηρήσεων του κάθε αλγόριθμου, θα προχωρήσουμε σε συγκρίσεις μεταξύ τους για να εξάγουεμ κάποια συμπεράσματα.

Αρχικά, οι χρόνοι που δίνονται από το Library based quad-tree KNN search algorithm, παρατηρώ ότι μπορούν να μας δώσουν μερικές χρήσιμες πληροφορίες.

Συγκρίνοντάς τον με το δικό μας implementation του quad tree algorithm, καταλαβαίνουμε ότι έχει γίνει αρκετά καλή δουλειά, αφού σε αρκετές περιπτώσεις τον ξεπερνάμε σε ταχύτητα, ακόμα και για πολύ μεγάλα N. Από την άλλη, εντοπίζουμε στην δική μας υλοποίηση ένα κομμάτι που έχει περιθώρια επιπλέον βελτιστοποίησης. Αυτό είναι η περίπτωση περισσότερων γειτόνων, αφού για μεγαλύτερα K ο αλγόριθμός μας φαίνεται να έχει μεγαλύτερη καθυστέρηση σε σχέση με τον πρώτο.
Στην συνέχεια, για να εξάγουμε το επόμενο συμπέρασμα θα συγκρίνουμε τους χρόνους του KD algorithm μας με τον Range. Συγκεκριμένα διαβάζουμε στο Wikipedia entry του Range algorithm :
“The range tree is an alternative to the *k*-d tree. Compared to *k*-d trees, range trees offer faster query times of (in Big O notation) but worse storage of , where *n* is the number of points stored in the tree, *d* is the dimension of each point and *k* is the number of points reported by a given query.” [2]

Αυτό αντιφάσκει στους χρόνους που βρήκαμε μεταξύ του K-NN μας και του Range μας για όλα τα N εκτός από πολύ μικρό αριθμό δεδομένων της τάξης των 100-1000 και όλα τα K. Επομένως, ο αλγόριθμός μας για KNN search using Range Tree, έχει χώρο επιπλέον βελτιστοποίησης.
Επιπλέον άλλο ένα ενδεχόμενο κοιτώντας τους δύο πίνακες :

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n \log^{d-1} n)$	$O(n \log^{d-1} n)$
Search	$O(\log^{d} n + k)$	$O(\log^{d} n + k)$

[1], [2]

είναι ότι για τα συγκεκριμένα δεδομένα που επιλέξαμε, έχουμε average case για τον kd, επομένως αποφεύγεται η γραμμική πολυπλοκότητα και καταφέρνει να ξεπεράσει τον range.

Τέλος συγκρίνοντας τον K-D αλγόριθμό μας τον δικό μας quad tree αλλά και της βιβλιοθήκης βλέπουμε ότι είναι αρκετά αποδοτικός και γρήγορος, αφού είναι ελαφρώς πιο αργός από της βιβλιοθήκης και ελαφρώς πιο γρήγορος από τον δικό μας, κυρίως για μεγαλύτερα N. Ακόμα μια παρατήρηση, είναι ότι στον quad έχουμε πολύ γρηγορότερη αύξηση χρόνου αναλογικά με το πλήθος τον γειτόνων σε σχέση με τον kd. Άρα για μεγαλύτερα K, θα προτιμήσουμε τον kd αλγόριθμο.

Πηγές

[1] https://en.wikipedia.org/wiki/K-d_tree
[2] https://en.wikipedia.org/wiki/Range_tree