ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΤΕΛΙΚΗ ΑΝΑΦΟΡΑ 2023-2024

ΠΙΤΣΑΣ ΛΟΥΚΑΣ - 1115201900156
ΣΤΕΡΓΙΟΣ ΤΖΙΡΑΚΗΣ - 1115201900188

Εισαγωγή

Σε αυτή την αναφορά, παρουσιάζουμε το τελικό παραδοτέο στα πλαίσια του μαθήματος ***Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα*** για το ακαδημαϊκό έτος 2023-2024. Το πρόβλημα προς επίλυση είναι η κατασκευή KNN γράφων για την εύρεση k-κοντινότερων γειτόνων πάνω σε πολυδιάστατα σύνολα δεδομένων. Η εφαρμογή που αναπτύξαμε σε C++ ακολουθεί την λογική του αλγορίθμου ***NN-Descent*** όπως ήταν ζητούμενο. Υλοποιούνται διάφορες βελτιστοποιήσεις που περιγράφονται στην υλοποίηση του PyNNDescent καθώς και από τους Wei Dong, Charikar Moses, Kai Li. 2011.

Στην αναφορά μας, εξηγούμε τις σχεδιαστικές επιλογές μας για την ανάπτυξη του εν λόγω αλγορίθμου καθώς και των λοιπών βελτιστοποιήσεων. Παρουσιάζουμε πειραματικά δεδομένα και τα αποτελέσματα της εφαρμογής πάνω σε αυτά.

Δομές δεδομένων

Για τη κατασκευή του γράφου, υλοποιούμε τη παρακάτω δομή για να αναπαραστήσουμε ένα κόμβο μέσα στο σύνολο δεδομένων:

```
//A class defining the KNN (k-nearest neighbors) problem.
class KNN {
    //The data points become nodes inside our graph.
    //We define neighbors and reverse neighbors seperately,
    //thus we have both a directed and an undirected graph at will.
    struct node {
        float norm{ 0 }; //Euclidean norm
        float *cord{ nullptr }; //Coordinate array

        omp_lock_t lock; //Lock for parallelism
        minAVL<float, unsigned int> Cedge; //Candidates found during solve
        N_AVL<float, unsigned int> edge; //Indices to neighbors, sorted by distance - only the k best are kept
        RN_AVL<unsigned int> Redge; //Indices to reverse neighbors
    };
```

Ο γράφος είναι απλά ένας πίνακας με n (graph size) από τις παραπάνω δομές. Οι κόμβοι μας εσωτερικά αποτελούνται σχεδόν εξ ολοκλήρου απο AVL δέντρα: οι γείτονες, οι πιθανοί γείτονες, καθώς και οι αντίστροφοι γείτονες, αναπαριστώνται με κάποια μορφή AVL δέντρων (των οποίων οι λεπτομέρειες υλοποίησης φαίνονται στο αρχείο AVL.h). Πρόκειται για templated δομές που δημιουργήσαμε απο την αρχή για υλοποίηση της εφαρμογής μας.

Η επιλογή AVL δέντρων βασίζεται στους εξής παράγοντες:

- **<u>Οργάνωση Δεδομένων:</u>** Τα AVL δέντρα που χρησιμοποιούνται για τους γείτονες ταξινομούνται αυτόματα. Με άλλα λόγια, οι γείτονες θα είναι πάντα ταξινομημένοι με βάση τη καλύτερη απόσταση, και δεν χρειάζεται παραπάνω ενέργεια για ταξινόμηση.

  Οι γείτονες και οι αντίστροφοι γείτονες είναι χωρισμένοι σε νέους και παλιούς για την διαδικασία δειγματοληψίας. Η διάκριση γίνεται με τον διαχωρισμό σε δύο δέντρα, ένα για τους καινούργιους και ένα για τους παλιούς γείτονες. Αν και αυτό σημαίνει πως χρειάζονται μεταφορές μεταξύ των δύο δέντρων, θεωρούμε πως είναι μια καλή επιλογή, καθώς εξαλείφεται εντελώς η ανάγκη αποθήκευσης επιπλέον πληροφορίας (bool flags), καθώς και η ανάγκη αναζήτησης των νέων γειτόνων κατά τη διαδικασία δειγματοληψίας.

  Για χαμηλότερα ποσοστά δειγματοληψίας, η αποδοτικότητα της προσέγγισής μας συνεχώς αυξάνεται, καθώς μειώνονται οι μεταφορές, ενώ αυξάνεται ο λόγος: (χώρος που απαιτείται για αποθήκευση bool flags) / (πλήθος δειγμάτων).

- **<u>Ταχύτητα:</u>** Η εισαγωγή και εξαγωγή ενός στοιχείου στο δέντρο έχει χειρότερη απόδοση O(logn), ενώ το δέντρο παραμένει ισορροπημένο. Αυτό σημαίνει πως οποιαδήποτε αναζήτηση επίσης έχει χειρότερη απόδοση O(logn). Η αναζήτηση είναι απαραίτητη για την διαγραφή ενός αντίστροφου γείτονα.

  Κατα το βήμα ενημέρωσης του KNN γράφου, ο χειρότερος γείτονας (edge) συγκρίνεται με τον καλύτερο πιθανό γείτονα (Cedge). Η εύρεση μέγιστου/ελάχιστου στα AVL δέντρα είναι εξαιρετικά γρήγορη, ενώ η όλη διαδικασία σύγκρισης σταματάει μόλις βρεθεί ο πρώτος πιθανός γείτονας που δεν προσφέρει βελτίωση, αφού και οι δύο δομές γειτόνων είναι ταξινομημένες.

- **<u>Εξοικονόμηση χώρου:</u>** Τα AVL δέντρα που χρησιμοποιούνται για τους γείτονες καθώς και για τους πιθανούς γείτονες (Cedges) επιτρέπουν την ύπαρξη μέχρι k μοναδικών αντικειμένων μέσα στη δομή. Δεν υπάρχουν ποτέ διπλότυπα, ενώ η προσπάθεια εισαγωγής νέου γείτονα ενώ τα δέντρα είναι γεμάτα θα επιτύχουν μόνο αν ο νέος γείτονας έρχεται με καλύτερη απόσταση απο τον χειρότερο γείτονα.

Συνολικά η κλάση μας καταλαμβάνει όσο χώρο καταλαμβάνει ο γράφος, σύν μερικές επιπλέον πληροφορίες μικρού κόστους:

```
//The graph itself
vector<struct node> graph;

//Additional parameters
float *worst{ nullptr }; //Used on true solve
char dataset[100] = { '\0' };
unsigned int dim{ 0 }, threads{ 4 };
float (KNN::*distf)(unsigned int, unsigned int){ nullptr }; //Distance function
```

Στάδια εκτέλεσης

Για να είναι σαφής η δομή της εφαρμογής, αλλα και της κατανομής χρονισμού του προγράμματός μας, εξηγούμε συνοπτικά τα στάδια που ακολουθεί μια τυπική εκτέλεση του **KNN**:

- **Στάδιο 1 (initialization):** Κατα την κλήση της συνάρτησης KNN::solve() (αφού δηλαδή έχει διαβαστεί ο γράφος απο το αρχείο dataset), ξεκινάει ο πρώτος χρονισμός του προγράμματός μας. Αυτό το στάδιο περιλαμβάνει την αρχικοποίηση των γειτόνων μέσω δέντρων τυχαίων προβολών (RPTs), και σε προηγούμενες εκδόσεις της εργασίας, μέσω k-τυχαίων γειτόνων.
- **Στάδιο 2 (solve):** Σε αυτό το στάδιο εμπίπτει η επαναληπτική διαδικασία βελτίωσης του KNN γράφου. Διαιρείται σε δύο φάσεις: την δημιουργία υποψηφίων γειτόνων για κάθε κόμβο (Region A), και έπειτα την ενημέρωση του γράφου με τους υποψηφίους (Region B). Στο πρόγραμμά μας η πρώτη φάση καταλαμβάνει το περισσότερο χρόνο, αφού σε αυτό γίνονται και οι υπολογισμοί αποστάσεων.
- **Στάδιο 3 (determine accuracy):** Με τη συνάρτηση accuracy(), εξετάζουμε την ακρίβεια του KNN γράφου με τον αυθεντικό (100% ευστοχία) γράφο. Εάν υπάρχει λυμένος ο γράφος για το αντίστοιχο k, η διαδικασία είναι γρήγορη καθώς οι πραγματικοί γείτονες εξάγονται από το αρχείο. Αν δεν υπάρχει τέτοιο αρχείο, οι γείτονες υπολογίζονται από την αρχή. Αυτή η διαδικασία είναι αρκετά χρονοβόρα.

Τα στάδια που ακολουθεί το εκτελέσιμο **GraphSolve** είναι παρόμοια.

```
File: 00010000-4.bin | k: 20 | Delta: 0.001 | Sampling rate: 80% | Fast sampling: 0 | RPTrees: 12 | RPT leaf size: 4 | Threads: 4
Initializing graph... initialization complete | Time elapsed: 1905ms
Solving graph... graph solved at 12 iterations | Time elapsed: 13376ms
Calculating accuracy... relative accuracy: 99.6984% | Absolute accuracy: 99.4387% | Time elapsed: 672ms
```

*Τυπική εκτέλεση του **KNN**, 10000 nodes*

```
linux18:/home/users/sdi1900156/Documents/Project2023>./GraphSolve 00005000-1.bin -k 50
Solving graph... graph solved | Time elapsed: 15426ms
Exporting graph... done | Time elapsed: 185ms
linux18:/home/users/sdi1900156/Documents/Project2023>
```

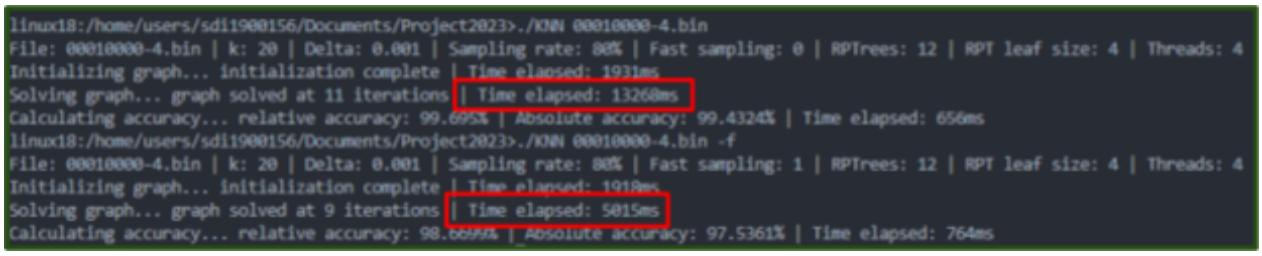
*Τυπική εκτέλεση του **GraphSolve**, 5000 nodes, k=50*

Τυχαία VS γρήγορη δειγματοληψία

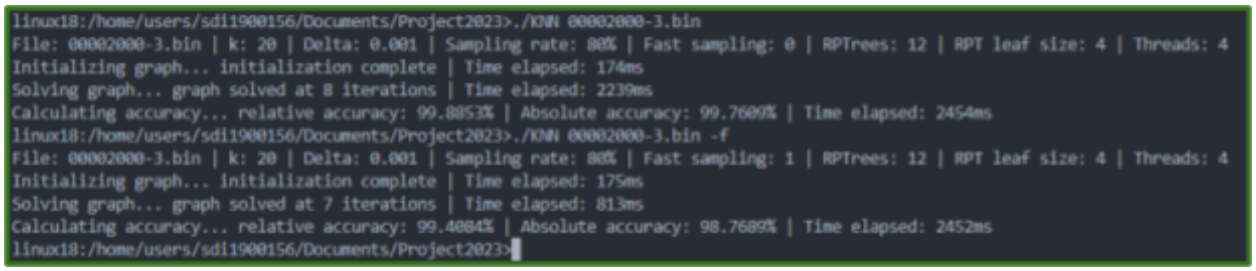
Η διαγραφή εσωτερικών κόμβων του AVL δέντρου (δηλαδή εκείνων που δεν είναι κοντά στα φύλλα) είναι αρκετά πιο αργή από την διαγραφή φύλλων του δέντρου. Υπο κανονικές συνθήκες, η δειγματοληψία κατα το στάδιο εύρεσης πιθανών γειτόνων είναι ημι-τυχαία (χωρίς χρήση RNG), η οποία μας προσφέρει ιδανική ευστοχία στον γράφο (>99% για αντίστοιχο k).

Ωστώσο, υλοποιούμε και έναν πιο γρήγορο τρόπο δειγματοληψίας, ο οποίος επιφέρει κατα μέσο όρο 66% γρηγορότερη λύση του γράφου με περίπου 1% απώλεια στην ευστοχία. Η γρήγορη δειγματοληψία γίνεται με εξαγωγή των min κόμβων.

Παραδείγματα κανονικής VS γρήγορης δειγματοληψίας:



VS fast sampling, 10000 nodes



VS fast sampling, 20000 nodes

VS fast sampling, 5000 nodes

Παραλληλία

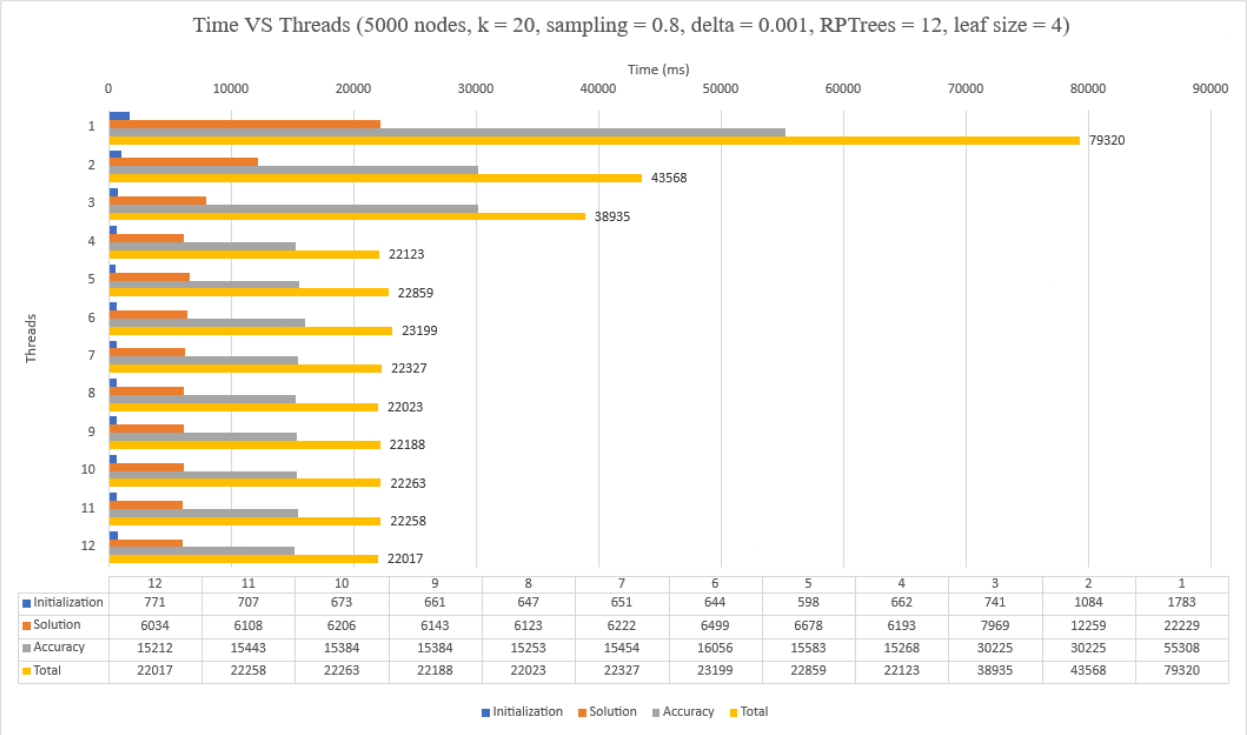
Λόγω της φύσης του προβλήματος (οργάνωση του γράφου σε πίνακα & ανάγκη για επαναληπτικές διαδικασίες), αρκετά στάδια του προβλήματος μπορούσαν να παραλληλοποιηθούν. Η πολυνημάτωση της εφαρμογής μας υλοποιείται μέσω της χρήσης OpenMP. Όλες οι παράλληλες περιοχές μας είναι πάνω σε for loops. Μέσω της χρήσης omp_locks, επιτυγχάνεται το mutual exclusion όταν αυτό χρειάζεται (κυρίως κατα την εισαγωγή γειτόνων).

Τα σημεία όπου παραλληλοποιήθηκαν είναι αρκετά. Ωστώσο, συνοπτικά μπορούμε να αναφέρουμε πως παραλληλοποιήσαμε τα παρακάτω μέρη του κώδικά μας. Όταν αναφερόμαστε σε jobs, αναφερόμαστε στο είδος της διαίρεσης που έγινε σε μια εργασία, π.χ. για τη δημιουργία x δέντρων τυχαίων προβολών, η δουλειά που θα έχει ένα νήμα κάθε φορά θα είναι η δημιουργία ενός δέντρου (job = 1 tree).

- Δημιουργία δέντρων τυχαίων προβολών (job = 1 tree)
- Αρχικοποίηση γειτόνων μέσω των φύλλων από τα RPTrees (job = 1 leaf)
- Στάδιο 2A: Δειγματοληψία και εύρεση υποψήφιων γειτόνων για κάθε γείτονα (job = 1 neighbor)
- Στάδιο 2B: Ενημέρωση των γειτόνων του KNN γράφου (job = 1 neighbor)
- Υπολογισμός ευστοχίας από αρχείο (παράλληλη ανάγνωση γραμμών από το αρχείο μέσω πολλαπλών file descriptors και χρήση fseek)
- Υπολογισμός νόρμας σημείου
- Υπολογισμός του πραγματικού KNN γράφου

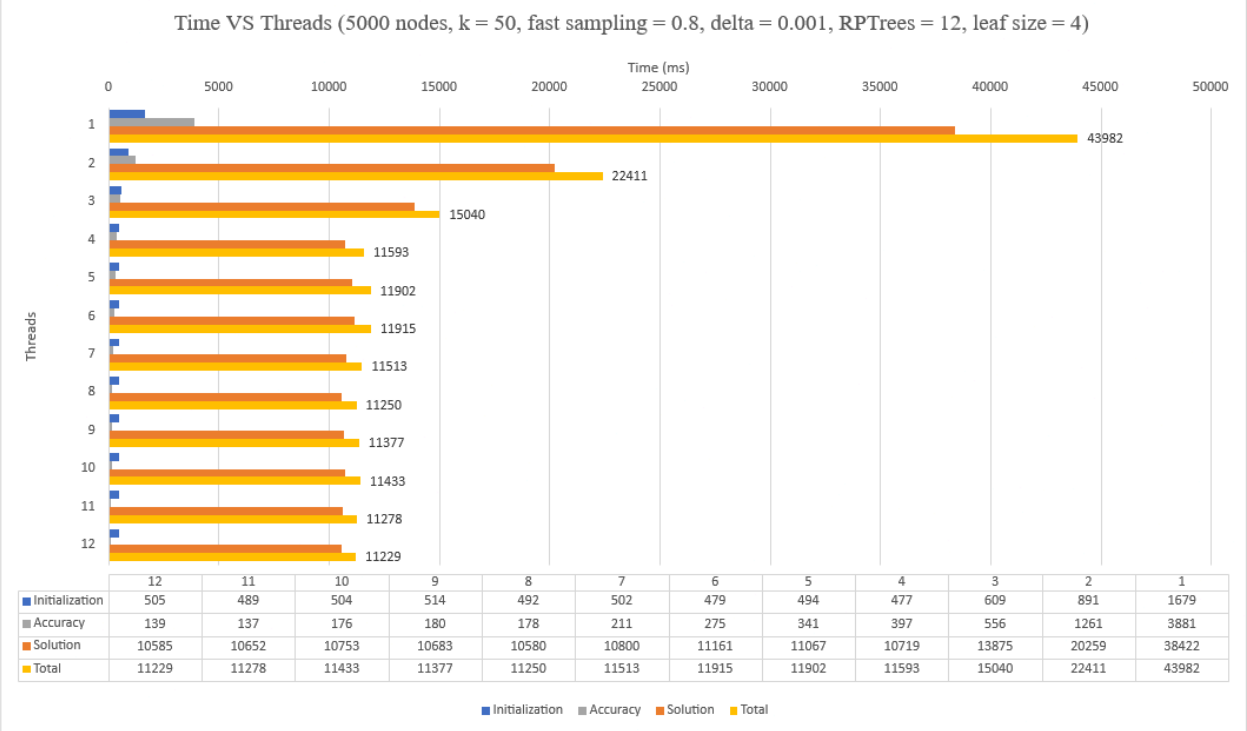
Στο σύστημα στο οποίο κάνουμε τις δοκιμές (οι προδιαγραφές του αναγράφονται παρακάτω), παρατηρούμε οτι με 4 threads φτάνουμε γρήγορα στη μέγιστη αποδοτικότητα. Σίγουρα, η βελτίωση που έρχεται με τη πολυνημάτωση είναι μεγάλη σε μια εφαρμογή όπως αυτή, ειδικά για τον υπολογισμό του πραγματικού γράφου.

Στο εκτελέσιμο **KNN** αλλά και **GraphSolve** μπορούν να δοθούν ως όρισμα ο αριθμός των νημάτων που θα χρησιμοποιούνται από την εκάστοτε εφαρμογή (μέσω της σημαίας -t).



	12	11	10	9	8	7	6	5	4	3	2	1
Initialization	771	707	673	661	647	651	644	598	662	741	1084	1783
Solution	6034	6108	6206	6143	6123	6222	6499	6678	6193	7969	12259	22229
Accuracy	15212	15443	15384	15384	15253	15454	16056	15583	15268	30225	30225	55308
Total	22017	22258	22263	22188	22023	22327	23199	22859	22123	38935	43568	79320

*Χρόνος VS νήματα για εκτέλεση του **KNN** (χωρίς λυμένο γράφο), 5000 nodes, k = 20*

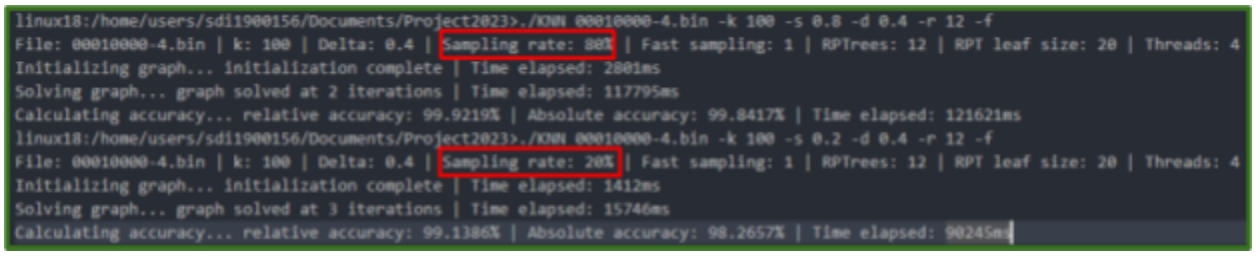


	12	11	10	9	8	7	6	5	4	3	2	1
Initialization	505	489	504	514	492	502	479	494	477	609	891	1679
Accuracy	139	137	176	180	178	211	275	341	397	556	1261	3881
Solution	10585	10652	10753	10683	10580	10800	11161	11067	10719	13875	20259	38422
Total	11229	11278	11433	11377	11250	11513	11915	11902	11593	15040	22411	43982

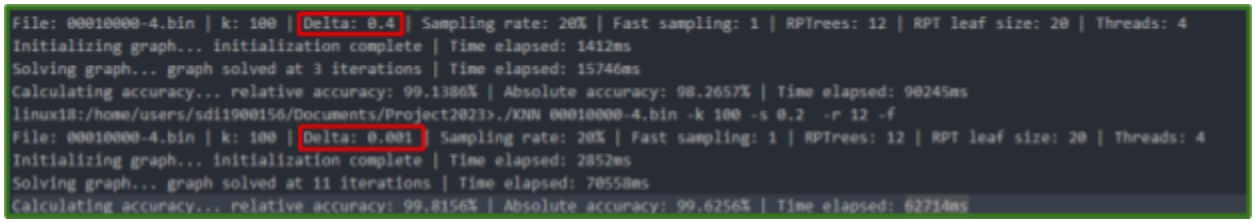
*Χρόνος VS νήματα για εκτέλεση του **KNN** (με λυμένο γράφο), 5000 nodes, k = 50, fast sampling*

Δειγματοληψία & Σταθερά δ

Η σταθερά δειγματοληψίας δεν είχε μεγάλο αντίκτυπο στα αποτελέσματα για μικρά k. Οι διακυμάνσεις είναι εως 2% στην ευστοχία, με αντίστοιχες τιμές στην χρονική απόδοση. Ωστώσο, για μεγαλύτερα k, η σταθερά δειγματοληψίας μαζί με την σταθερά δ παίζουν καθοριστικό ρόλο για τον τερματισμό του προγράμματος σε εύλογο χρονικό διάστημα. Ο συνδυασμός μικρό sampling rate/μεγάλο δ για μεγαλύτερα k μπορεί να φέρει πολύ καλύτερο χρόνο με πολύ καλή ευστοχία, ειδικά με την χρήση RPT δέντρων. Ο ιδανικός συνδυασμός εξαρτάται απο το k, αλλα και το μέγεθος του dataset.



VS (fast) sampling rate, 10000 nodes, k = 100, delta = 0.4



VS delta, 10000 nodes, k = 100, sampling rate = 20%

Προδιαγραφές συστήματος

Τα προγράμματα μεταγλωττίστηκαν και εκτελέστηκαν σε σύστημα με τα εξής specifications:

- **System**
  - Kernel: 5.4.0-167-generic x86_64
  - Bits: 64
  - Version: 9.4.0
  - Console: tty 6
  - Distro: Ubuntu 20.04.6 LTS (Focal Fossa)
- **CPU**
  - Model: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
  - Architecture: x86_64
  - Byte Order: Little Endian
  - Address sizes: 39 bits physical, 48 bits virtual
  - CPU(s): 4
  - Thread(s) per core: 1
  - Core(s) per socket: 4
  - CPU max MHz: 3600.0000
  - CPU min MHz: 800.0000
  - Virtualization: VT-x
  - L1d cache: 128 KiB
  - L1i cache: 128 KiB
  - L2 cache: 1 MiB
  - L3 cache: 6 MiB
- **Compiler:** g++ (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0