

**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΡΓΑΣΤΗΡΙΟ ΗΛΕΚΤΡΟΝΙΚΗΣ

**Σχεδιασμός Ενσωματωμένων Συστημάτων**

9ο εξάμηνο

2η Εργαστηριακή Άσκηση:

**Ασκήσεις στη Βελτιστοποίηση Δυναμικών Δομών Δεδομένων (Dynamic Data Type Refinement – DDTR)**

ΑΜ: 03116112

Ονοματεπώνυμο: Γεζεκελιάν Βικέν

**Άσκηση 1: Βελτιστοποίηση δυναμικών δομών δεδομένων του αλγορίθμου DRR.**

(a)

Για το συγκεκριμένο ερώτημα, φτιάξαμε απλώς τον κώδικα που βρίσκεται εντός του combos.py, ο οποίος αναλόγως με τα arguments που παίρνει φτιάχνει τον αντίστοιχο συνδυασμό υλοποιήσεων δομών δεδομένων.

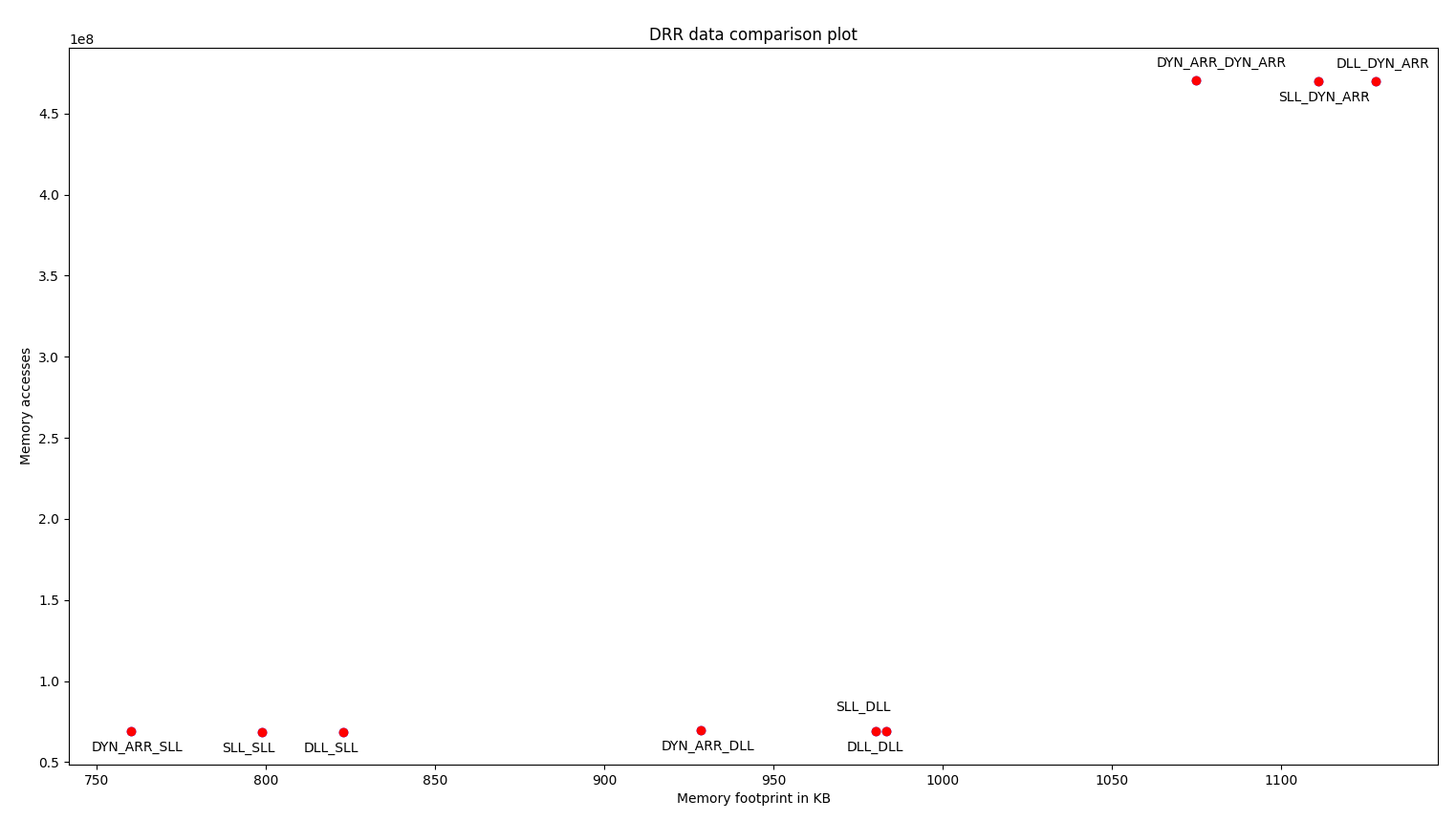
Έπειτα, χρησιμοποιήσαμε το bash script C\_1\_a.sh. Το script αυτό, αρχικά, για κάθε διαφορετικό συνδυασμό αλλάζει κατάλληλα τον κώδικα και τον κάνει compile δημιουργώντας συνολικά 9, κατάλληλα ονοματισμένα, εκτελέσιμα. Έπειτα, τρέχει κάθε ένα από αυτά τα εκτελέσιμα, και δημιουργεί με τις εντολές που δίνονται στην εκφώνηση και εντός του φάκελου Outputs τα κατάλληλα αρχεία:

1 αρχείο all\_memory\_accesses.txt, εντός του οποίου προσθέτει πρώτα το όνομα του iteration που τρέχουμε και έπειτα τον συνολικό αριθμό προσβάσεων

9 αρχεία X\_Y\_memory\_footprint.txt, όπου το κάθε ένα περιέχει το memory footprint του αντίστοιχου executable

Τα αποτελέσματα των παραπάνω μετρήσεων, αναπαρίστανται στη συνέχεια σε μορφή πίνακα και plot(το οποίο έχει δημιουργηθεί από το 1\_a\_plotter.py)

|  |  |  |  |
| --- | --- | --- | --- |
| CL Type | PK Type | Memory Accesses | Memory footprint |
| SLL | SLL |  | 798,8 KB |
| SLL | DLL |  | 980,3 KB |
| SLL | DYN\_ARR |  | 1.111 KB |
| DLL | SLL |  | 823,0 KB |
| DLL | DLL |  | 983,3 KB |
| DLL | DYN\_ARR |  | 1.128 KB |
| DYN\_ARR | SLL |  | 760,2 KB |
| DYN\_ARR | DLL |  | 928,5 KB |
| DYN\_ARR | DYN\_ARR |  | 1.075 KB |



(b)

Όπως βλέπουμε από τον πίνακα, η βέλτιστη υλοποίηση για τη μείωση του αριθμού των προσβάσεων μνήμης, είναι SLL(Single Linked List) τόσο για τους κόμβους, όσο και τα πακέτα

(c)

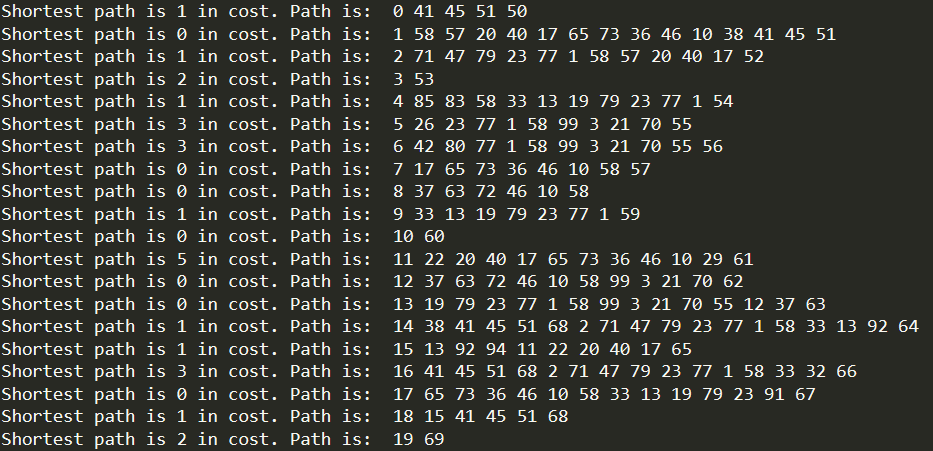
Όπως βλέπουμε από τον πίνακα, η βέλτιστη υλοποίηση για τη μείωση των απαιτήσεων μνήμης, είναι DYN\_ARR(Dynamic Array) για τους κόμβους και SLL(Single Linked List) για τα πακέτα

Επίσης, από το διάγραμμα που παραθέσαμε παραπάνω, βλέπουμε πως ο τελευταίος συνδυασμός, DYN\_ARR και SLL είναι και ο βέλτιστος κατά pareto.

**Άσκηση 2: Βελτιστοποίηση δυναμικών δομών δεδομένων του αλγορίθμου Dijkstra**

(α)

Ο κώδικας που μας δίνεται, βρίσκεται εντός του αρχείου dijkstra\_original.c. Οι μόνες αλλαγές που έχουν γίνει σε αυτόν, αφορούν μερικά warnings του compiler(include της <stdlib.h> και ένα return 0 στην int dijkstra). Τον κάναμε λοιπόν compile με την εντολή   
gcc48 -Wall -Wextra -g -Wno-unused-but-set-variable -o dijkstra\_original dijkstra\_original.c -pthread,   
και το output του κώδικα αφού τρέξαμε   
./dijkstra\_original input.dat > Outputs/dijkstra\_original.txt, βρίσκεται στο αρχείο dijkstra\_original.txt και είναι το εξής:



(b)

Ο μετασχηματισμένος κώδικας, βρίσκεται στο αρχείο dijkstra.c, ενώ ακόμη έχουμε δημιουργήσει και το check.py, το οποίο ελέγχει εάν τα outputs του μετασχηματισμένου κώδικα για κάθε διαφορετική δομή δεδομένων είναι σωστα.

(c)

Για την εκτέλεση αυτού του ερωτήματος(όπως και του προηγούμενου) έχουμε δημιουργήσει το bash script runner.sh. Το script αυτό, αρχικά κάνει compile τον κώδικα που μας δίνεται και δημιουργεί ένα executable dijkstra\_original. Έπειτα, δίνοντας διαφορετικά options στον κώδικα που έχουμε φτιάξει κατά το compile, δημιουργεί 3 διαφορετικά executables:

dijkstra\_sll, δίνοντας -DSLL  
dijkstra\_dll, δίνοντας -DDLL  
dijkstra\_dyn\_arr δίνοντας -DDYN\_ARR

Το καθένα από τα παραπάνω χρησιμοποιεί την δομή δεδομένων που προδίδει το όνομά του. Στη συνέχεια, το runner.sh τρέχει τις 4 αυτές παραλλαγές του κώδικα, αποθηκεύοντας τα outputs τους και συγκρίνοντας τα μέσω του check.py ώστε να σιγουρευτεί πως τα αποτελέσματα είναι όλα σωστά. Τέλος, δημιουργεί τα αρχεία για τα memory accesses και memory footprint της κάθε παραλλαγής(συμπεριλμβανομένου του αυθεντικού κώδικα) και τα αποθηκεύει στα κατάλληλα αρχεία, ακριβώς όπως και στην άσκηση 1.

Τα αποτελέσματα των memory accesses και memory footprints φαίνονται στον παρακάτω πίνακα:

|  |  |  |
| --- | --- | --- |
| Node Struct | Memory Accesses | Memory footprint |
| Original | 104.915.259 | 17,68 KB |
| SLL |  | 707,7 KB |
| DLL |  | 941,7 KB |
| DYN\_ARR |  | 360,6 KB |

(d)

Όπως βλέπουμε από τον πίνακα, η βέλτιστη υλοποίηση για τη μείωση του αριθμού των προσβάσεων μνήμης, είναι η SLL(Single Linked List)

(e)

Όπως βλέπουμε από τον πίνακα, η βέλτιστη υλοποίηση για τη μείωση των απαιτήσεων μνήμης, είναι(εξαιρώντας τον αρχικό κώδικα που δίνεται) η DYN\_ARR(Dynamic Array)