Σχεδιασμός Ενσωματωμένων Συστημάτων

2η Εργαστηριακή Άσκηση

Ασκήσεις στη Βελτιστοποίηση Δυναμικών Δομών Δεδομένων

(Dynamic Data Type Refinement – DDTR)

Νικολέτα-Μαρκέλα Ηλιακοπούλου
03116111

Φοίβος-Ευστράτιος Καλεμκερής
03116010

1 Εισαγωγή - Σκοπός της Άσκησης

Ο σκοπός της άσκησης είναι να βελτιστοποιηθούν οι δυναμικές δομές δεδομένων δύο δικτυακών εφαρμογών: του Deficit Round Robin (DRR) και του αλγορίθμου Dijkstra, με χρήση της μεθοδολογίας «Βελτιστοποίησης Δυναμικών Δομών Δεδομένων» - Dynamic Data Type Refinement (DDTR). Θα βελτιστοποιήσουμε τις δυναμικές δομές δεδομένων των δύο αυτών αλγορίθμων ως προς

- τις προσβάσεις στη μνήμη **(memory accesses)** που απαιτούνται για να προσπελαστουν τα δεδομένα τους
- τη μέγιστη ποσότητα μνήμης που καταλαμβάνουν **(memory footprint)**

Τα εργαλεία που χρησιμοποιήθηκαν στην άσκηση είναι

- **Linux** (Ubuntu 18.04)
- **gcc** (Ubuntu 7.5.0-3ubuntu1 18.04) **7.5.0**
- valgrind-3.16.1 (**Massif, Lackey** εργαλεία της ValgrindSuite)

2 Deficit Round Robin

2.1 Περιγραφή Διαδικασίας

Για τον αλγόριθμο αυτό έχουμε έτοιμη την υλοποίηση του στα αρχεία drr.h, drr.c και προσυνδεδεμένη την βιβλιοθήκη DDTR με τον source code. Μεταγλωττίζουμε το αρχείο 9 φορές, επιλέγοντας κάθε φορά διαφορετικό συνδυασμό από τις δομές δεδομένων που μας δίνονται (**Single Linked List, Double Linked List, Dynamic Array**). Αυτό γίνεται με το να κάνουμε comment και uncomment τα αντίστοιχα definitions των δομών δεδομένων στο αρχείο "drr.h". Αυτό θα γίνει με την εντολή

```
$ gcc drr.c -o drri -pthread -lcdsl -L./../synch_implementations \
      -I./../synch_implementations
```

($i = 0 \rightarrow 9$)
Έχοντας πλέον τα 9 εκτελέσιμα αρχεία του κώδικα τρέχουμε για το καθένα από αυτά τις παρακάτω εντολές, ώστε να βρούμε τον αριθμό των προσβάσεων στη μνήμη κατά την εκτέλεση του κώδικα.

```
$ valgrind --log-file="mem_accesses_logi.txt" --tool=lackey --trace-mem=yes ./drri

$ cat mem_accesses_logi.txt | grep 'I\| L' | wc -l
```

Έπειτα, για να βρούμε τη μέγιστη χρήση μνήμης εκτελούμε τις παρακάτω εντολές

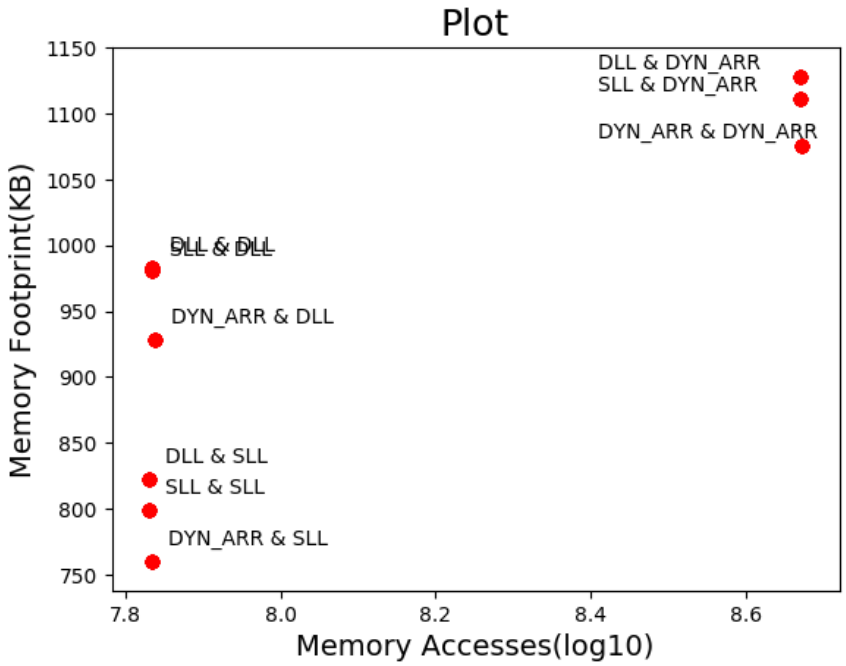
```
$ valgrind --tool=massif ./drri

$  ms_print massif.out.XXXXX > mem_footprint_logi.txt
```

Όπου $XXXXX$ ο αριθμός που αντιστοιχεί στο αρχείο που παράχθηκε.

2.2 Αποτελέσματα

Client List	Packet List	Memory Accesses	Memory Footprint
SLL	SLL	67732342	798.8 KB
SLL	DLL	68376553	980.3 KB
SLL	DYN_ARR	469437498	1.111 MB
DLL	SLL	67744776	823.0 KB
DLL	DLL	68388587	983.3 KB
DLL	DYN_ARR	469453681	1.128 MB
DYN_ARR	SLL	68257559	760.2 KB
DYN_ARR	DLL	68923684	928.5 KB
DYN_ARR	DYN_ARR	470156422	1.075 MB

Πίνακας 1: Memory Accesses και Memory Footprint για τους συνδυασμούς δυναμικών δομών δεδομένων στον DRR



Σχήμα 1: Διάγραμμα memory accesses-memory footprint για τους διάφορους συνδυασμούς δυναμικών δομών δεδομένων στον DRR

2.2.1 Memory Accesses

Παρατηρούμε ότι ανεξαρτήτως της δομής που χρησιμοποιούμε στο client list, η δομή packet list παίζει καθοριστικό ρόλο στις προσβάσεις στη μνήμη. Συγκεκριμένα, η χρήση του dynamic array στο packet list οδηγεί σε πολύ περισσότερες προσβάσεις στη μνήμη απ' ότι η χρήση των άλλων δύο δομών. Οι υπόλοιποι συνδυασμοί απαιτούν ίδιας τάξης αριθμό προσβάσεων στη μνήμη, με τον SLL - SLL να είναι ο καλύτερος όλων.

2.2.2 Memory Footprint

Από τη μεριά του packet list παρατηρούμε πως το memory footprint αυξάνεται με τη σειρά SLL -> DLL -> DYN_ARR. Αυτό είναι λογικό, καθώς είμαστε στην περίπτωση όπου οι κόμβοι εισάγονται και διαγράφονται από τη δομή δεδομένων δυναμικά. Ο dynamic array ουσιαστικά δεσμεύει περισσότερη μνήμη από ότι χρειάζεται, λόγω του resize, κάτι το οποίο μπορεί να χρειαστεί να γίνει συχνά στην περίπτωση του drr.

Από τη μεριά του client list το memory footprint μειώνεται με τη σειρά DLL -> SLL -> DYN_ARR. Αυτό μπορεί να συμβεί επειδή η λίστα περιέχει αντίστοιχο αριθμό κόμβων με αυτόν του array. Άρα σε αυτή την περίπτωση έχουμε χοντρικά:

- footprint=(sizeof(int)+2*sizeof(int*)), για DLL
- footprint=(sizeof(int)+sizeof(int*)), για SLL
- footprint=(sizeof(int))*N , για DYN_ARR

όπου N ο αριθμός των στοιχείων.
Σύμφωνα με τα παραπάνω, αναμένουμε πως ο χειρότερος συνδυασμός όσον αφορά το memory footprint θα είναι ο DLL - DYN_ARR, ενώ ο καλύτερος συνδυασμός ο DYN_ARR - SLL, τα οποία επιβεβαιώνονται, όπως φαίνεται τόσο στο διάγραμμα όσο και στον πίνακα των αποτελεσμάτων που παραθέσαμε.

Στον ακόλουθο πίνακα συνοψίζονται τα συμπεράσματα για τα memory accesses και το memory footprint:

Μετρική	**Βέλτιστος συνδυασμός**	**Τιμή**
Memory Accesses	SLL - SLL	67732342
Memory Footprint	DYN_ARR - SLL	760.2 KB

Πίνακας 2: Memory Accesses και Memory Footprint για τους βέλτιστους συνδυασμούς δυναμικών δομών δεδομένων στον DRR

3 Dijkstra

3.1 Περιγραφή Διαδικασίας

Αρχικά, έχουμε τον έτοιμο κώδικα για τον αλγόριθμο του Dijkstra χωρις όμως να έχει εισαχθεί η βιβλιοθήκη DDTR. Τρέχουμε τον κώδικα και καταγράφουμε τα αποτελέσματα που παράγει. Παρακάτω φαίνονται τα αποτελέσματα.

```
Shortest path is 1 in cost. Path is:  0 41 45 51 50
Shortest path is 0 in cost. Path is:  1 58 57 20 40 17 65 73 36 46 10 38 41 45 51
Shortest path is 1 in cost. Path is:  2 71 47 79 23 77 1 58 57 20 40 17 52
Shortest path is 2 in cost. Path is:  3 53
Shortest path is 1 in cost. Path is:  4 85 83 58 33 13 19 79 23 77 1 54
Shortest path is 3 in cost. Path is:  5 26 23 77 1 58 99 3 21 70 55
Shortest path is 3 in cost. Path is:  6 42 80 77 1 58 99 3 21 70 55 56
Shortest path is 0 in cost. Path is:  7 17 65 73 36 46 10 58 57
Shortest path is 0 in cost. Path is:  8 37 63 72 46 10 58
Shortest path is 1 in cost. Path is:  9 33 13 19 79 23 77 1 59
Shortest path is 0 in cost. Path is:  10 60
Shortest path is 5 in cost. Path is:  11 22 20 40 17 65 73 36 46 10 29 61
Shortest path is 0 in cost. Path is:  12 37 63 72 46 10 58 99 3 21 70 62
Shortest path is 0 in cost. Path is:  13 19 79 23 77 1 58 99 3 21 70 55 12 37 63
Shortest path is 1 in cost. Path is:  14 38 41 45 51 68 2 71 47 79 23 77 1 58 33 13 92 64
Shortest path is 1 in cost. Path is:  15 13 92 94 11 22 20 40 17 65
Shortest path is 3 in cost. Path is:  16 41 45 51 68 2 71 47 79 23 77 1 58 33 32 66
Shortest path is 0 in cost. Path is:  17 65 73 36 46 10 58 33 13 19 79 23 91 67
Shortest path is 1 in cost. Path is:  18 15 41 45 51 68
Shortest path is 2 in cost. Path is:  19 69
```

Έπειτα, εισάγουμε την βιβλιοθήκη DDTR και ως δομές δεδομένων επιλέγουμε τις αντίστοιχες με τον αλγόριθμο DRR με τη μόνη διαφορά ότι δεν χρησιμοποιούμε ζευγάρια αυτών αλλά μια δομή κάθε φορά. Όπως και προηγουμένως θα κάνουμε comment και uncomment τα definitions για κάθε τύπο δεδομένων, ενώ θα χρησιμοποιήσουμε κάθε φορά το script *run.sh* (*./run.sh ddtr*) για την εκτέλεση του κώδικα και τη λήψη των αποτελεσμάτων. Για την εισαγωγή της βιβλιοθήκης DDTR, ακολουθήσαμε τα βήματα που περιγράφονται στην εκφώνηση. Ειδικότερα:

1. Κάναμε include τα 3 header files (cdsl_queue.h, cdsl_deque.h, cdsl_dyn_array.h), στα οποία βρίσκονται οι δηλώσεις των συναρτήσεων (function declarations) που μπορούν να χρησιμοποιηθούν στον κώδικα της εφαρμογής για να αντικαταστήσουν τη δομή δεδομένων της εφαρμογής με αυτές της βιβλιοθήκης DDTR:

```
// Step a: include necessary headers
#if defined(SLL)
#include "../synch_implementations/cdsl_queue.h"
#endif
#if defined(DLL)
#include "../synch_implementations/cdsl_deque.h"
#endif
#if defined(DYN_ARR)
#include "../synch_implementations/cdsl_dyn_array.h"
#endif
```

2. Αντικαταστήσαμε τις δηλώσεις (definitions – declarations) των δομών δεδομένων της εφαρμογής με αυτά της βιβλιοθήκης DDTR. Όπως αναφέρουμε σε σχόλιο, ο iterator θα χρειαστεί στη συνάρτηση dequeue παρακάτω:

```
// Step b: adjust definitions/ declarations
// The iterators will be needed for the dequeue function.
#if defined(SLL)
cdsl_sll *qHead;
iterator_cdsl_sll it;
#endif
#if defined(DLL)
cdsl_dll *qHead;
iterator_cdsl_dll it;
#endif
#if defined(DYN_ARR)
cdsl_dyn_array *qHead;
iterator_cdsl_dyn_array it;
#endif
```

3. Δημιουργήσαμε και αρχικοποιήσαμε (initialization) τις δομές δεδομένων εντός της συνάρτησης main(), καλώντας τις αντίστοιχες cdsl_XXX_init() συναρτήσεις (functions) από τη βιβλιοθήκη (XXX είναι η συγκεκριμένη υλοποίηση).

```
// Step c: initialise
#if defined(SLL)
    qHead = cdsl_sll_init();
#endif
#if defined(DLL)
    qHead = cdsl_dll_init();
#endif
#if defined(DYN_ARR)
    qHead = cdsl_dyn_array_init();
#endif
```

4. Αντικαταστήσαμε τις συναρτήσεις των λειτουργιών των δομών δεδομένων (data structure operation functions) της εφαρμογής με αυτές της βιβλιοθήκης. Ειδικότερα, χρειάστηκε να τροποποιήσουμε τις συναρτήσεις **enqueue** και **dequeue**. Για τη χρήση των παρεχόμενων από τη βιβλιοθήκη συναρτήσεων συμβουλευτήκαμε τον κώδικα *drr.c*, στον οποίο χρησιμοποιούνται. Προέκυψαν, έτσι, οι ακόλουθες συναρτήσεις:

```
void enqueue (int iNode, int iDist, int iPrev)
{
    QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));

    if (!qNew)
    {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
    }
    qNew->iNode = iNode;
    qNew->iDist = iDist;
    qNew->iPrev = iPrev;

    qHead->enqueue(0, qHead, (void *)qNew);
    g_qCount++;
}

void dequeue (int *piNode, int *piDist, int *piPrev)
{
    it = qHead->iter_begin(qHead);
    QITEM *item = (QITEM*)(qHead->iter_deref(qHead, it));

    *piNode = item->iNode;
    *piDist = item->iDist;
    *piPrev = item->iPrev;
    qHead->dequeue(0, (qHead));
    g_qCount--;
}
```

Μετα την εφαρμογή των Δομών Δεδομένων κανοντας χρήση του εργαλειου Lackey της Valgrind βρίσκουμε τα memory accesses για τον αλγόριθμο Dijkstra με την αντίστοιχη δομή δεδομενων, και μέσω του εργαλείου Massif της Valgrind βρίσκουμε το memory footprint. Να σημειωθεί ότι τα αποτελέσματα του αλγορίθμου μετά την εισαγωγή της βιβλιοθήκης DDTR είναι τα ίδια με αυτά που παραθέσαμε παραπάνω, οπότε είμαστε σίγουροι για την ορθότητα της υλοποίησής μας.

3.2 Αποτελέσματα

Παραθέτουμε στον ακόλουθο πίνακα τα αποτελέσματα των μετρήσεων όσον αφορά στα memory accesses και memory footprint για κάθε διαφορετική υλοποίηση του αλγορίθμου.

Data Structure	Memory Accesses	Memory Footprint
SLL	102317980	356.5 KB
DLL	102486489	471.3 KB
DYN_ARR	149800562	360.7 KB

Πίνακας 3: Memory Accesses και Memory Footprint για τους συνδυασμούς δυναμικών δομών δεδομένων στον Dijkstra

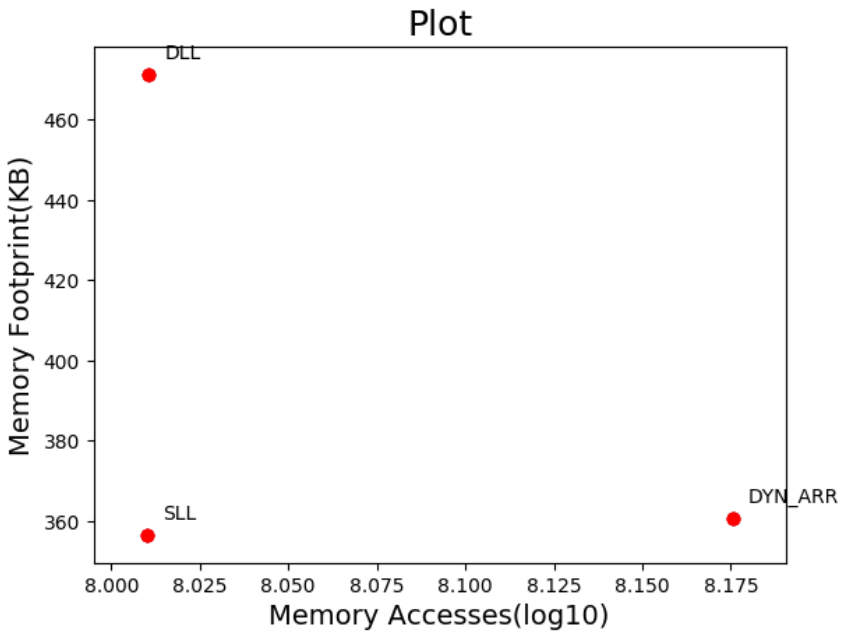
3.2.1 Memory Accesses

Παρατηρούμε ότι όσον αφορά στον αριθμό προσβάσεων στη μνήμη, αυτός αυξάνεται με τη σειρά SLL -> DLL -> DYN_ARR. Συνεπώς αντιλαμβανόμαστε ότι με τη χρήση της Single Linked List έχουμε τον μικρότερο αριθμό memory accesses.

3.2.2 Memory Footprint

Από την άλλη, αναφορικά με τις απαιτήσεις των προγραμμάτων σε μνήμη, αυτές αυξάνονται με τη σειρά SLL -> DYN_ARR -> DLL, με αποτέλεσμα και πάλι η Single Linked List να αναδεικνύεται στη βέλτιστη επιλογή.

Οι παραπάνω παρατηρήσεις αποτυπώνονται και στο ακόλουθο διάγραμμα.



Σχήμα 2: Διάγραμμα memory accesses-memory footprint για τις διάφορες δυναμικές δομές δεδομένων στον Dijkstra

Από τα παραπάνω, καθίσταται σαφές πως η καλύτερη δομή δεδομένων από άποψη τόσο αριθμού προσβάσεων στη μνήμη όσο και αποτυπώματος μνήμης είναι η SLL.

Σημείωση: Τα αντίστοιχα αρχεία κώδικα που τροποποιήθηκαν/ δημιουργήθηκαν για το σκοπό της άσκησης παρατίθενται στο συμπιεσμένο αρχείο, μαζί με τα Makefiles και τα script για την αυτοματοποίηση της διαδικασίας για τα Massif και Lackey και την κατασκευή των διαγραμμάτων.