ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

«2η Εργαστηριακή Άσκηση»

Εργαστηριακή Αναφορά στο μάθημα

Σχεδιασμός Ενσωματωμένων Συστημάτων

των φοιτητών

Αβραμίδης Σταύρος Α.Μ. 17811

Λάζου Μαρία-Αργυρώ Α.Μ. 20192

Διδάσκοντες:

Σωτήριος Ξύδης, Δημήτριος Σούντρης, Σωτήριος Κοκόσης

Νοέμβριος 2024

Άσκηση 1η

Εισαγωγή

Ο σκοπός της άσκησης είναι να βελτιστοποιηθούν οι δυναμικές δομές δεδομένων, του Deficit Round Robin (DRR), με χρήση της μεθοδολογίας «Βελτιστοποίησης Δυναμικών Δομών Δεδομένων» - Dynamic Data Type Refinement (DDTR).

Μεθοδολογία

Οι παρακάτω μετρήσεις, έγιναν με την χρήση του εργαλείου Valgrind, σε σύστημα με επεξεργαστή Intel Core i9-7900X, μνήμη RAM 32GB, λειτουργικό σύστημα CachyOS με kernel 6.11.8-1-cachyos-bore-x86_64_v4.

Χρησιμοποιήθηκε το παρακάτω σκρίπτ για την αυτοματοποίηση των μετρήσεων:

run_drr.sh

```
#!/bin/bash

CL_CONFIGS=(SLL_CL DLL_CL DYN_ARR_CL)
PK_CONFIGS=(SLL_PK DLL_PK DYN_ARR_PK)

BASE_CMD="gcc ./src/DRR/drr.c -pthread -lcdsl -no-pie -L./src/synch_implementations -I./src/synch_implementations"

mkdir -p results/drr

for cl in ${CL_CONFIGS[@]}; do
    for pk in ${PK_CONFIGS[@]}; do
        # Compile
        echo "DDR with $cl and $pk"
        $BASE_CMD -D$cl -D$pk -o drr_${cl}_${pk}.exe

        # Run tasks in the background
        (
            # Run on valgrind lackey
             valgrind --log-file="./results/drr/mem_accesses_log_${cl}_${pk}.txt" --tool=lackey --trace-mem=yes ./drr_${cl}_${pk}.exe

               grep -c 'I\| L' "./results/drr/mem_accesses_log_${cl}_${pk}.txt" >"./results/drr/mem_accesses_count_${cl}_${pk}.txt"
            rm -f "./results/drr/mem_accesses_log_${cl}_${pk}.txt"

            # Run valgrind massif
            valgrind --tool=massif ./drr_${cl}_${pk}.exe &
            MASSIF_PID=$!
            wait $MASSIF_PID
            ms_print massif.out.${MASSIF_PID} >"./results/drr/massif_log_${cl}_${pk}.txt"
            rm -f massif.out.${MASSIF_PID}
        ) &
    done
done

# Wait for all background tasks to complete
wait

rm ./drr_*.exe
```

Αποτελέσματα

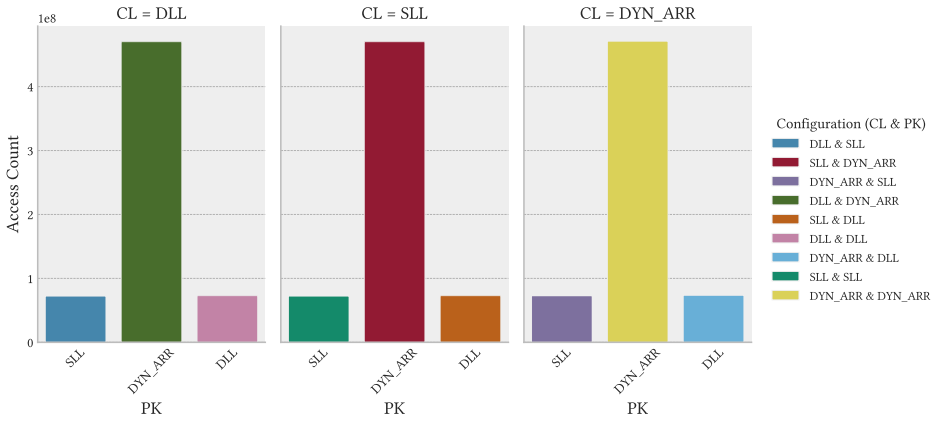
Στον παρακάτω πίνακα παρουσιάζονται τα αποτελέσματα των μετρήσεων για τον συνολικό αριθμό προσπελάσεων κάθε συνδυασμού παραμέτρων και το peak memory usage κάθε συνδυασμού παραμέτρων.

Ορίζονται τα εξής:

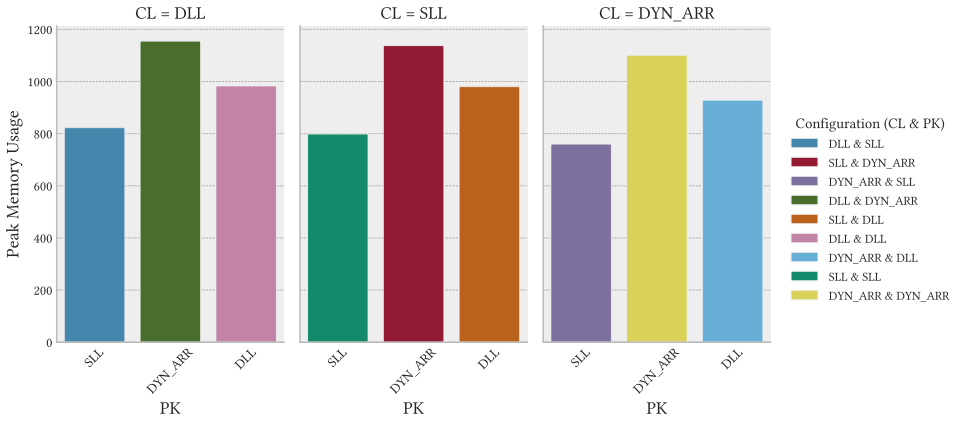
- SLL: Single Linked List
- DLL: Double Linked List
- DYN_ARRAY: Dynamic Array
- CL: Client List
- PK: Packet List

CL	PK	Access Count	Peak Memory Usage (kB)
DLL	SLL	72387715	823.0
SLL	DYN_ARR	470311196	1137.664
DYN_ARR	SLL	72868746	760.2
DLL	DYN_ARR	470325689	1155.072
SLL	DLL	73046057	980.3
DLL	DLL	73058113	983.3
DYN_ARR	DLL	73554157	928.5
SLL	SLL	72375051	798.8
DYN_ARR	DYN_ARR	471036994	1100.8

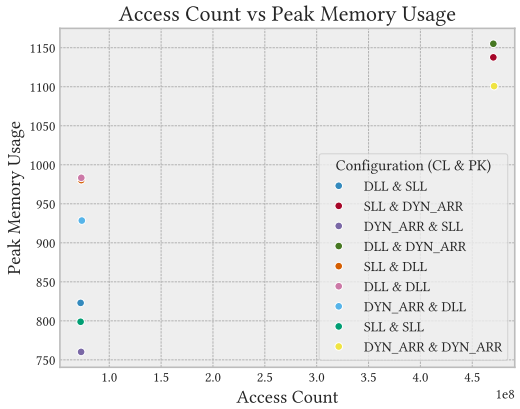
Πίνακας 1: Συγκεντρωτικά Αποτελέσματα DRR



Σχήμα 1: Συνολικός αριθμός προσπελάσεων κάθε συνδυασμού παραμέτρων



Σχήμα 2: Peak memory Usage κάθε συνδυασμού παραμέτρων



Σχήμα 3: Συνολικός αριθμός προσπελάσεων vs Peak memory Usage κάθε συνδυασμού παραμέτρων

Από τις καταγραφές, εντοπίστηκε ο συνδυασμός (CL = SLL, PL = SLL) να είναι αυτός με τον μικρότερο αριθμό προσπελάσεων στην μνήμη (72375051), ενώ ο συνδυασμός (CL = DYN_ARRAY, PL = SLL) εκείνος με το χαμηλότερο memory footprint (760.2 kB). Οι επιλογές αυτές ερμηνεύονται ως εξής:
Παρατηρούμε, αρχικά, ότι η επιλογή DYN_ARRAY για τα πακέτα (PK) οδηγεί σε αύξηση των accessess. Στις δομές αυτές, γίνονται συχνά insertions / deletions, άρα απαιτούνται resizing operations που περιλαμβάνουν reads & writes σε νέα memory locations όταν γεμίσει το block που βρίσκονταν αρχικά, ώστε να ικανοποιείται υ συνθήκη του dynamic array για δέσμευση **διαδοχικών** θέσεων μνήμης. Αντίθετα, η χρήση DYN_ARRAY στν λίστα των κόμβων του δικτύου (CL), που έχει γνωστό μέγεθος από την αρχή και σταθερό καθ'ολη την διάρκεια εκτέλεσης του αλγορίθμου δεν απαιτεί περιττά memory accesses λόγω resizing. Ακόμη, η συγκεκριμένη εφαρμογή ακολουθεί προβλέψιμα iterator based access patterns (επισκέπτεται σειριακά τους κόμβους τόσο στην δομή CL όσο και στην PK ξεκινώντας από το head και ακολουθώντας πάντα τον next) επομένως στην περίπτωση των λιστών, μπορεί να επωφεληθεί από τον Roving pointer που δείχνει κάθε φορά στην θέση του προηγούμενου access. Έτσι η πολυπλοκότητα εύρεσης πέφτει από O(n) σε O(1), όπου n ο αριθμός των προσβάσεων μνήμης (όπως δηλαδή και στην DYN_ARRAY). Τέλος, επειδή δεν κινούμαστε ποτές backwards ο prev pointer δεν μειώνει το κόστος κάποιας πρόσβασης.
Όσον αφορά στο memory footprint, οι λίστες απιτούν παραπάνω μνήμη per node για την αποθήκευση των pointers (έναν (next) για την single) και (δύο (next, prev) για την double) σε αντίθεση με το dynamic array που δεσμεύει συνεχόμενες θέσεις μνήμης και δεν χρειάζεται pointer chasing.

Άσκηση 2η

Εισαγωγή

Στην 2η άσκηση ζητήθηκε η εισαγωγή των δομών του 1ου μέρους και κατόπιν αξιολόγηση τους, σε μία προϋπάρχουσα υλοποίηση του αλγορίθμου των Shortest Path Trees (SPT) με την χρήση του αλγορίθμου Dijkstra.

Υλοποίηση

Ο δοθέν κώδικας τροποποιήθηκε ως εξής:

dijkstra_cdsl.c

```c
#include <stdio.h>
#include <stdlib.h>

#if defined(SLL)
    #include "../synch_implementations/cdsl_queue.h"
#elif defined(DLL)
    #include "../synch_implementations/cdsl_deque.h"
#elif (DYN_ARR)
    #include "../synch_implementations/cdsl_dyn_array.h"
#else
    #error "No synch implementation defined"
#endif

#define NUM_NODES 100
#define NONE      9999

struct _NODE
{
    int iDist;
    int iPrev;
};
typedef struct _NODE NODE;

struct _QITEM
{
    int            iNode;
    int            iDist;
    int            iPrev;
    struct _QITEM *qNext;
};
typedef struct _QITEM QITEM;

#if defined(SLL)
cdsl_sll                *qHead;
typedef iterator_cdsl_sll iterator;
#elif defined(DLL)
cdsl_dll                *qHead;
typedef iterator_cdsl_dll iterator;
#elif defined(DYN_ARR)
cdsl_dyn_array                *qHead;
typedef iterator_cdsl_dyn_array iterator;
#endif

static inline void init_head(void)
{
#if defined(SLL)
    qHead = cdsl_sll_init();
#elif defined(DLL)
    qHead = cdsl_dll_init();
#elif defined(DYN_ARR)
    qHead = cdsl_dyn_array_init();
#endif
}

int AdjMatrix[NUM_NODES][NUM_NODES];

int  g_qCount = 0;
NODE rgnNodes[NUM_NODES];
int  ch;
int  iPrev, iNode;
```

```
int  i, iCost, iDist;

void print_path(NODE *rgnNodes, int chNode)
{
    if (rgnNodes[chNode].iPrev != NONE) {
        print_path(rgnNodes, rgnNodes[chNode].iPrev);
    }
    printf(" %d", chNode);
    fflush(stdout);
}

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)
{
    iterator it = qHead->iter_begin(qHead);

    QITEM *qKill = (QITEM *)qHead->iter_deref(qHead, it);

    if (qHead != NULL) {

        *piNode = qKill->iNode;
        *piDist = qKill->iDist;
        *piPrev = qKill->iPrev;

        qHead->remove(0, qHead, qKill);
        g_qCount--;
    }
}

int qcount(void)
{
    return (g_qCount);
}

int dijkstra(int chStart, int chEnd)
{

    for (ch = 0; ch < NUM_NODES; ch++) {
        rgnNodes[ch].iDist = NONE;
        rgnNodes[ch].iPrev = NONE;
    }

    if (chStart == chEnd) {
        printf("Shortest path is 0 in cost. Just stay where you are.\n");
    } else {
        rgnNodes[chStart].iDist = 0;
        rgnNodes[chStart].iPrev = NONE;

        enqueue(chStart, 0, NONE);

        while (qcount() > 0) {
            dequeue(&iNode, &iDist, &iPrev);
            for (i = 0; i < NUM_NODES; i++) {
                if ((iCost = AdjMatrix[iNode][i]) != NONE) {
                    if ((NONE == rgnNodes[i].iDist) || (rgnNodes[i].iDist > (iCost + iDist))) {
                        rgnNodes[i].iDist = iDist + iCost;
                        rgnNodes[i].iPrev = iNode;
                        enqueue(i, iDist + iCost, iNode);
                    }
                }
            }
        }

        printf("Shortest path is %d in cost. ", rgnNodes[chEnd].iDist);
        printf("Path is: ");
```

```
        print_path(rgnNodes, chEnd);
        printf("\n");
    }
}

int main(int argc, char *argv[])
{
    int   i, j, k;
    FILE *fp;

    if (argc < 2) {
        fprintf(stderr, "Usage: dijkstra <filename>\n");
        fprintf(stderr, "Only supports matrix size is #define'd.\n");

        exit(1);
    }

    /* init qHead */
    init_head();

    /* open the adjacency matrix file */
    fp = fopen(argv[1], "r");

    /* make a fully connected matrix */
    for (i = 0; i < NUM_NODES; i++) {
        for (j = 0; j < NUM_NODES; j++) {
            /* make it more sparce */
            fscanf(fp, "%d", &k);
            AdjMatrix[i][j] = k;
        }
    }

    /* finds 10 shortest paths between nodes */
    for (i = 0, j = NUM_NODES / 2; i < 20; i++, j++) {
        j = j % NUM_NODES;
        dijkstra(i, j);
    }
}
```

Η μετρήσεις παράχθηκαν με την χρήση του παρακάτω σκρίπτ:

run_dijkstra.sh

```
#!/bin/bash

CONFIGS=(SLL DLL DYN_ARR)

BASE_CMD="gcc ./src/dijkstra/dijkstra_cdsl.c -pthread -lcdsl -no-pie -L./src/synch_implementations -I./src/synch_implementations"

mkdir -p results/dijkstra

for config in ${CONFIGS[@]}; do
    (
        # Compile
        echo "Dijkstra with $config"
        $BASE_CMD -D$config -o dijkstra_${config}.exe

        # Run on valgrind lackey
         valgrind --log-file="./results/dijkstra/mem_accesses_log_${config}.txt" --tool=lackey --trace-mem=yes ./dijkstra_${config}.exe ./src/dijkstra/input.dat
        grep -c 'I\| L' "./results/dijkstra/mem_accesses_log_${config}.txt" >"./results/dijkstra/mem_accesses_count_${config}.txt"
        rm -f "./results/dijkstra/mem_accesses_log_${config}.txt"

        # Run valgrind massif
        valgrind --tool=massif ./dijkstra_${config}.exe ./src/dijkstra/input.dat &
        MASSIF_PID=$!
        wait $MASSIF_PID
        ms_print massif.out.${MASSIF_PID} >"./results/dijkstra/massif_log_${config}.txt"
        rm -f massif.out.${MASSIF_PID}
    ) &
done

# Also run the plain dijkstra
(

```

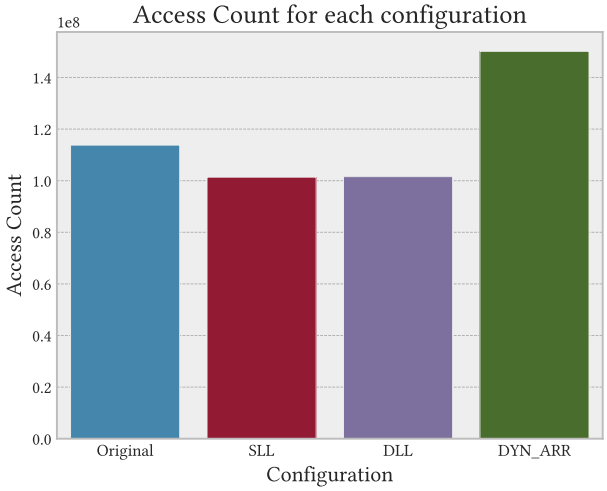
```
    echo "Dijkstra Plain"
    gcc ./src/dijkstra/dijkstra.c -pthread -o dijkstra.exe
     valgrind --log-file="./results/dijkstra/mem_accesses_log_original.txt" --tool=lackey --trace-mem=yes ./dijkstra ./src/dijkstra/input.dat

       grep -c 'I\| L' "./results/dijkstra/mem_accesses_log_original.txt" >"./results/dijkstra/mem_accesses_count_original.txt"
    rm -f "./results/dijkstra/mem_accesses_log_original.txt"

    valgrind --tool=massif ./dijkstra ./src/dijkstra/input.dat &
    MASSIF_PID=$!
    wait $MASSIF_PID
    ms_print massif.out.${MASSIF_PID} >"./results/dijkstra/massif_log_original.txt"
    rm -f massif.out.${MASSIF_PID}
) &

wait

rm ./dijkstra*.exe
```

Αποτελέσματα

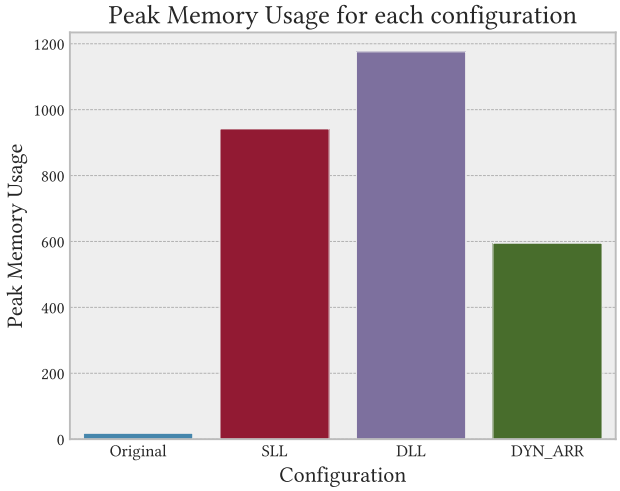
Ως *«original»* επισημαίνεται η αρχική υλοποίηση του αλγορίθμου.

Configuration	Access Count	Peak Memory Usage (kB)
DLL	101576693	1175.552
Original	113791196	17.68
DYN_ARR	150135162	594.6
SLL	101367529	941.6

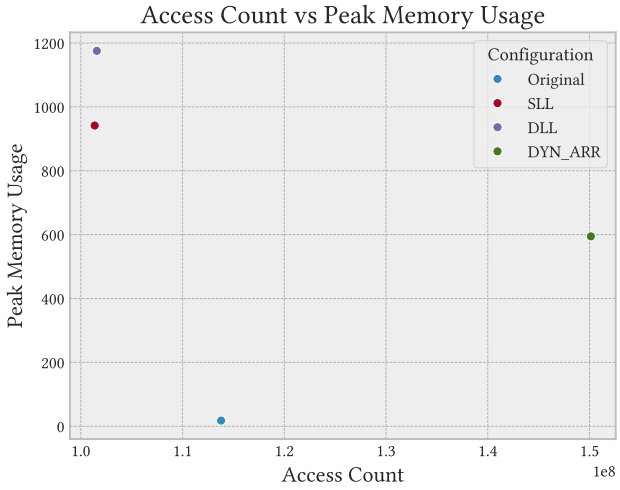
Πίνακας 2: Συγκεντρωτικά Αποτελέσματα Dijkstra



Σχήμα 4: Συνολικός αριθμός προσπελάσεων κάθε συνδυασμοΘ παραμέτρων



Σχήμα 5: Peak memory Usage κάθε συνδυασμού παραμέτρων



Σχήμα 6: Συνολικός αριθμός προσπελάσεων vs Peak memory Usage κάθε συνδυασμού παραμέτρων

Από τις καταγραφές, εντοπίστηκε η υλοποίηση με SLL να είναι αυτή με τον μικρότερο αριθμό προσπελάσεων στην μνήμη (101367529), ενώ η Original εκείνη με το χαμηλότερο memory footprint (17.62 kB). Ωστόσω, επειδή οι συγκρίσεις γίνονται αποκλειστικά μεταξύ δομών της βιβλιοθήκης, καταγράφουμε την υληποίηση DYN_ARRAY με 594.6 kB lowest memory footprint. Οι ερμηνείες των αποτελεσμάτων είναι παρόμοιες με αυτές για τον drr.

Συγκεκριμένα, γίνονται πολλές λειτουργίες pop / push για την εύρεση των προς εξέταση κόμβων, άρα απιτούνται resizings των DYN_ARRAY δομών σε περίπτωση που προστεθούν πολλοί κόμβοι (πιθανό να συμβεί κατά την επέκταση του βέλτιστου μονοπατιού καθώς είναι μια αναδρομική διαδικασία). Επίσης, επειδή ο αλγόριθμος είναι greedy έχουμε μόνο forward steps προς κόμβους που δεν έχουμε ήδη προσπελάσει, άρα η αποθήκευση prev pointers σε μια DLL δομή είναι περιττή. Τέλος, για τους λόγους που αναφέρθηκαν πριν, η πρόσβαση στην SLL για τους γείτονες (που είναι αποθηκευμένοι σειριακά) γίνεται και εδώ σε O(1) εαν χρησιμοποιείται Roving pointer, ο οποίος δείχνει σε κάθε iteration στον τρέχοντα κόμβο.
Το memory footprint είναι και πάλι χαμηλότερο για το DYN_ARRAY καθώς αποθηκεύει μόνο values και όχι pointers όπως εξηγήθηκε πριν.