2η Εργαστηριακή Άσκηση στο μάθημα Σχεδιασμός Ενσωματωμένων Συστημάτων

Ομάδα 29

Μέλη: Αριστοτέλης Γρίβας

Σάββας Λεβεντικίδης

Άσκηση 1:

Α) Τα αποτελέσματα τα οποία καταγράφθηκαν κατά την εκτέλεση της εφαρμογής DRR και αφορούν τον αριθμό των προσβάσεων στη μνήμη (*memory accesses*) και το μέγεθος της απαιτούμενης μνήμης (*memory footprint*) για κάθε έναν από τους διαφορετικούς συνδυασμούς υλοποιήσεων των δομών δεδομένων (*clientList, pList*) είναι τα εξής:

MEMORY ACCESSES			
Client \ **Packet**	SLL	DLL	DYN_ARR
SLL	71882119	72554333	470935571
DLL	71894098	72566107	470950605
DYN_ARR	72380778	73065810	471683534

MEMORY FOOTPRINT (KB)			
Client \ **Packet**	SLL	DLL	DYN_ARR
SLL	798.8	980.3	1111.0
DLL	823.0	983.3	1128.0
DYN_ARR	760.2	928.5	1075.0

Ο κάθε συνδυασμός δοκιμάστηκε βγάζοντας από τα σχόλια την εκάστοτε υλοποίηση για τις δομές δεδομένων clientList, pList στην αρχή του αρχείο **drr.h** .

Β) Ο συνδυασμός υλοποιήσεων δομών δεδομένων για τον οποίο η εφαρμογή έχει τον μικρότερο αριθμό προσβάσεων στην μνήμη είναι ο εξής:

- Απλά Συνδεδεμένη Λίστα (SLL) για την λίστα των κόμβων .
- Απλά Συνδεδεμένη Λίστα (SLL) για την λίστα των πακέτων.

} 71882119 προσβάσεις

Τα αποτελέσματα αυτά, οφείλονται στο γεγονός πως στη δομή Single Linked List, κάθε κόμβος περιέχει μόνο ένα pointer στον επόμενο, με αποτέλεσμα να έχουμε λιγότερες προσβάσεις στη μνήμη σε κάθε επανάληψη, σε σύκγριση με τις δύο άλλες δομές οι οποίες χρησιμοποιούν επιπλέον pointers για κάθε κόμβο, κάνοντας περισσότερες προσβάσεις.

Γ) Ο συνδυασμός υλοποιήσεων δομών δεδομένων για τον οποίο η εφαρμογή έχει τις μικρότερες απαιτήσεις σε μνήμη είναι ο εξής:

- Δυναμικός Πίνακας (DYN_ARR) για την λίστα των κόμβων.
- Απλά Συνδεδεμένη Λίστα (SLL) για την λίστα των πακέτων.

} 760.2 KB

Αυτή τη φορά βλέπουμε πως για τη λίστα των κόμβων, η βέλτιστη επιλογή είναι Dynamic Array. Αυτό, πιθανώς, οφείλεται στη ικανότητα του δυναμικού πίνακα να κάνει resize (αποδεσμεύει μνήμη) κάθε φορά που δεν απαιτεί ορισμένο χώρο στη μνήμη, με αποτέλεσμα να μειώνεται το footprint. Ωστόσο, η διαδικασία αυτή είναι αργή, και για μεγαλύτερο εύρος δεδομένων, όπως τα πακέτα (plist), δεν είναι αποδοτική, όπως φαίνεται και από τα αποτελέσματα, επομένως διαλέγουμε SLL για τη λίστα των πακέτων.

Άσκηση 2:

Α) Τα αποτελέσματα από την εκτέλεση της εφαρμογής Dijkstra είναι τα εξής:

```
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
```

Τα αποτελέσματα που παράγει ο αρχικός κώδικας όσον αφορά στις προσβάσεις μνήμης και στις απαιτήσεις μνήμης είναι τα εξής :

- Προσβάσεις στη μνήμη (*memory accesses*) : 113910491
- Απαίτηση μνήμης (*memory footprint*) : 17.68 (KB)

B) Παρατηρούμε πως αρχικά ο κώδικας είναι υλοποιημένος με χρήση ουράς.Συνεπώς, αφότου εισάγουμε και αρχικοποιήσουμε τις δομές δεδομένων που ζητούνται από την εκφώνηση, το μόνο που έχουμε να κάνουμε είναι η υλοποίηση εκ νέου των συναρτήσεων enqueue και dequeue.Έτσι, όπως θα φανεί και παρακάτω, βάζουμε σε σχόλια τις αρχικές συναρτήσεις enqueue και dequeue και προσθέτουμε την νέα μας υλοποίηση όπου αυτές εκτελούνταν (μέσα στη συνάρτηση Dijkstra).Ο κώδικας που προκύπτει μετά την εισαγωγή της βιβλιοθήκης DDTR και της αντικατάστασης των δομών δεδομένων με αυτές της ίδιας βιβλιοθήκης (DDTR) είναι ο εξής:

```
#include <stdio.h>

#define NUM_NODES                           100
#define NONE                                9999



//#define SLL
#define DLL
//#define DYN_ARR

#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


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;

//QITEM *qHead = NULL;

#if defined(SLL)
cdsl_sll *qList;
#elif defined(DLL)
cdsl_dll *qList;
#elif defined(DYN_ARR)
cdsl_dyn_array *qList;
```

```
#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);
}

/* we will implement queue and dequeue in dijkstra() function

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

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

  if (!qLast)
    {
      qHead = qNew;
    }
  else
    {
      while (qLast->qNext) qLast = qLast->qNext;
      qLast->qNext = qNew;
    }
  g_qCount++;

}


void dequeue (int *piNode, int *piDist, int *piPrev)
{
```

```
  QITEM *qKill = qHead;

  if (qHead)
    {

      *piNode = qHead->iNode;
      *piDist = qHead->iDist;
      *piPrev = qHead->iPrev;
      qHead = qHead->qNext;
      free(qKill);
      g_qCount--;
    }
}

*/


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

int dijkstra(int chStart, int chEnd)
{

#if defined(SLL)
qList=cdsl_sll_init();
#elif defined(DLL)
qList=cdsl_dll_init();
#elif defined (DYN_ARR)
qList=cdsl_dyn_array_init();
#endif


  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);

      QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
      qNew->iNode=chStart;
      qNew->iDist=0;
      qNew->iPrev=NONE;
```

```
      qList->enqueue(0, qList, (void*)qNew);
      g_qCount++;


     while (qcount() > 0)
    {
      //dequeue (&iNode, &iDist, &iPrev);

      qNew=qList->dequeue(0,qList);
      iNode=qNew->iNode;
      iDist=qNew->iDist;
      iPrev=qNew->iPrev;
          g_qCount--;



      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);
              QITEM *qNew = (QITEM *) malloc(sizeof(QITEM));
                  qNew->iNode=i;
                  qNew->iDist=iDist+iCost;
                  qNew->iPrev=iNode;
                  qList->enqueue(0,qList, (void*)qNew);
                  g_qCount++;

            }
        }
        }
            free(qNew);
    }

      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");
  }
```

```
  /* 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);
  }
  exit(0);


}
```

Το αποτέλεσμα της εκτέλεσης της εφαρμογής με την υλοποίηση των δομών δεδομένων της DDTR είναι το ίδιο με το ερώτημα (Α).

Όπως και πριν, έτσι και τώρα, η κάθε δομή δεδομένων χρησιμοποιείται βγάζοντας από τα σχόλια την αντίστοιχη λειτουργία στην αρχή του κώδικα (sll, dll, dyn_arr).

Γ) Τα αποτελέσματα τα οποία καταγράφθηκαν κατά την εκτέλεση της εφαρμογής Dijkstra και αφορούν τον αριθμό των προσβάσεων στη μνήμη (*memory accesses*) και το μέγεθος της απαιτούμενης μνήμης (*memory footprint*) για κάθε διαφορετική υλοποίηση είναι τα εξής:

Data Type	Memory Accesses	Memory Footprint (KB)
SLL	101292157	28.05
DLL	101470542	33.52
DYN_ARR	149158503	97.73

Δ) Η υλοποίηση δομής δεδομένων με την οποία η εφαρμογή Dijkstra έχει τον μικρότερο αριθμό προσβάσεων στη μνήμη είναι η εξής:

- Απλά Συνδεδεμένη Λίστα (SLL) ⟶ 101292157 προσβάσεις

Όπως είδαμε και στο πρώτο ερώτημα (DRR), αυτό το αποτέλεσμα είναι λογικό, καθώς η SLL χρειάζεται μόνο ένα pointer για κάθε κόμβο, κάνοντας λιγότερες προσβάσεις στη μνήμη σε κάθε επάναληψη, ενώ οι άλλες δομές χρειάζονται περισσότερους pointers, με κόστος να έχουν χειρότερο memory efficiency.

Ε) Η υλοποίηση δομής δεδομένων με την οποία η εφαρμογή Dijkstra έχει τις μικρότερες απαιτήσεις σε μνήμη είναι η εξής:

- Απλά Συνδεδεμένη Λίστα (SLL) ⟶ 28.05 KB

Σε σύκγριση με το πρώτο ερώτημα (DRR), θα περιμέναμε να έχουμε ως βέλτιστη δομή δεδομένων για το memory footprint το Dynamic Array καθώς έχει την ικανότητα να κάνει resize, αποδεσμεύοντας μνήμη, όταν δε τη χρειάζεται. Ωστόσο, τα αποτελέσματα είναι διαφορετικά και ως βέλτιστη δομή έχουμε την SLL. Αυτό, όπως αναφέραμε παραπάνω, οφείλεται πιθανότητα στο γεγονός πως η διαδικασία για το resize είναι αργή, και το σύνολο των δεδομένων (Qitems στον κώδικα) είναι μεγάλο, το οποίο κάνει τη διαδικασία αυτή όλο και πιο δύσκολη.