National Technical University of Athens
Electrical and Computer Engineering

Embedded Systems

2nd Lab

Dynamic Data Structure Optimization Exercises (Dynamic Data Type Refinement (DDTR)

Nanos Georgios

nanosgiwrgos1997@gmail.com

Task 1: Optimizing dynamic data structures of the algorithm DRR

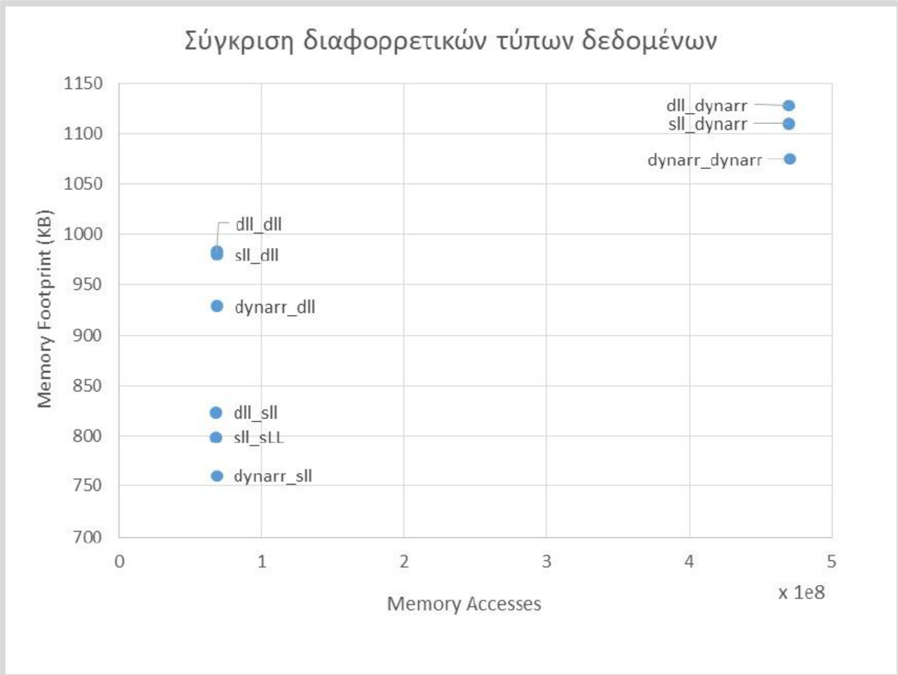
The DRR source code has already passed the DDTR library.
a) First we ran the application with all the different combinations implementations of data structures for the list of packages and the list of nodes. This was made possible by placing in comments each time the definitions that for two data structures, and leaving out the structure that we were studying. Below are the commands through which we did Compilation of the DRR application, for 9 different combinations.

```
gcc-4.8 drr.c -o drr_sll_sll -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_sll_dynarr -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_dll_sll -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_dll_dll -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_dll_dynarr -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_dynarr_sll -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_dynarr_dll -pthread -lcdsl -L./../synch_implementations
-I./../synch_implementations
gcc-4.8 drr.c -o drr_dynarr_dynarr -pthread -lcdsl -L./../
    synch_implementations
-I./../synch_implementations
```

We used script.sh and for each combination we recorded the its results on the number of accesses to memory (memory accesses) and the size of the required memory (memory footprint).

Node Struct	Packet Struct	Memory Accesses	Memory Footprint	
SLL	SLL	67698178	798.8	KB
SLL	DLL	68335279	980.3	KB
SLL	DYN_ARR	469412656	1.110	MB
DLL	SLL	67710604	823.3	KB
DLL	DLL	68347313	983.3	KB
DLL	DYN_ARR	469428855	1.128	MB
DYN_ARR	SLL	68225743	760.2	KB
DYN_ARR	DLL	68884212	928.5	KB
DYN_ARR	DYN_ARR	470133885	1.075	MB

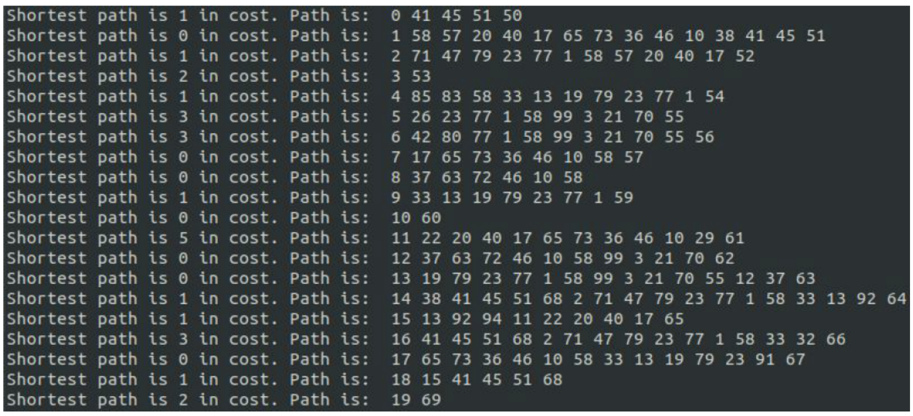
We also present the results in a scatter plot diagram.



(b) The combination of data structure implementations with which the application has the smallest number of memory accesses (minimum number of memory accesses) is o SLL-SLL. (c) The combination of data structure implementations with which the application has the smallest memory footprint is o DYNARR-SLL.

Task 2: Optimizing dynamic data structures of the algorithm Dijkstra

The dijkstra algorithm finds the shortest path in a table of size 100x100. The nodes that it examines and that make up the shortest path are stored in a list. We apply the DDTR methodology for this implementation: a) We ran the application and recorded the results:



```
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
```

b) After importing the library into the application, we replaced its data structure with the data structures of the library. More specifically, the changes/additions made were as follows: i) Add to the beginning the necessary definitions of the DDTR library.

```
//#define SLL
//#define DLL
#define DYN_ARR
#if defined(SLL)
```

```c
#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
```

ii) We replaced the command:

```c
QITEM *qHead = NULL;
```

with the commands:

```c
#if defined(SLL)
cdsl_sll *qHead;
#endif
#if defined(DLL)
cdsl_dll *qHead;
#endif
#if defined(DYN_ARR)
cdsl_dyn_array *qHead;
#endif
```

so that, depending on the data structure we use, the appropriate index in it. iii) In main we added the following commands:

```c
#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
```

to initialize the appropriate data structure based on the define. iv)Part of the enque function, which created a node based on values of the definitions and added it at the end, shown below:

```c
QITEM *qLast = qHead;
qNew->qNext = NULL;
if (!qLast)
{
    qHead = qNew;
}
else
{
    while (qLast->qNext) qLast = qLast->qNext;
    qLast->qNext = qNew;
}
```

We replaced the above commands with the library command:

```c
qHead->enqueue(0, qHead, (void *)qNew);
```

which does the same node insertion, regardless of the data structure we use. v)In the deque function, we replaced the following command, which created a pointer, which pointed to the same point (same address memory address) that qHead points to,

```c
QITEM *qKill = qHead;
```

with the commands,

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

that perform the same function regardless of the data structure (defined pointer qKill pointing, where qHead points) Finally, in order to delete the first node, it was necessary to replace the following commands:

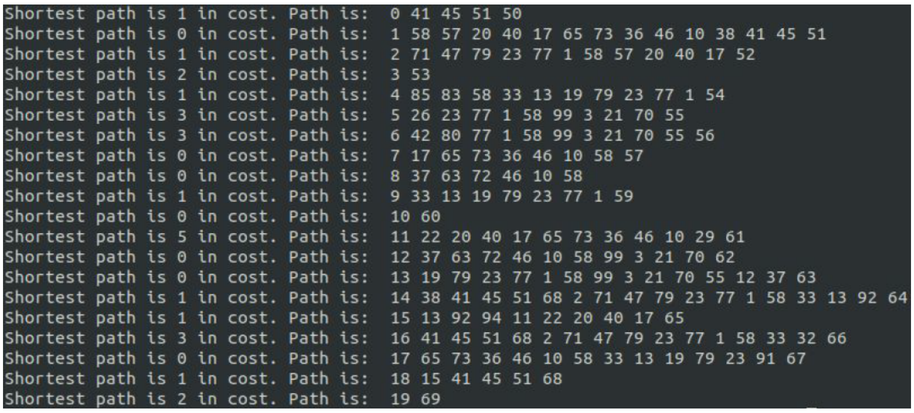
```
qHead = qHead->qNext;
free(qKill);
```

with the library command:

```
qHead = qHead->qNext;
qHead->remove(0, qHead, qKill);
```

We ran the application and verified that the library has been imported correctly, comparing the results produced with those recorded in question (a). (Obviously they are exactly the same).

```
gcc-4.8 dijkstra_opt.c -o dijkstra_opt_sll -pthread -lcdsl
-L./../synch_implementations -I./../synch_implementations
gcc-4.8 dijkstra_opt.c -o dijkstra_opt_dll -pthread -lcdsl
-L./../synch_implementations -I./../synch_implementations
gcc-4.8 dijkstra_opt.c -o dijkstra_opt_dynarr -pthread -lcdsl
-L./../synch_implementations -I./../synch_implementations
```

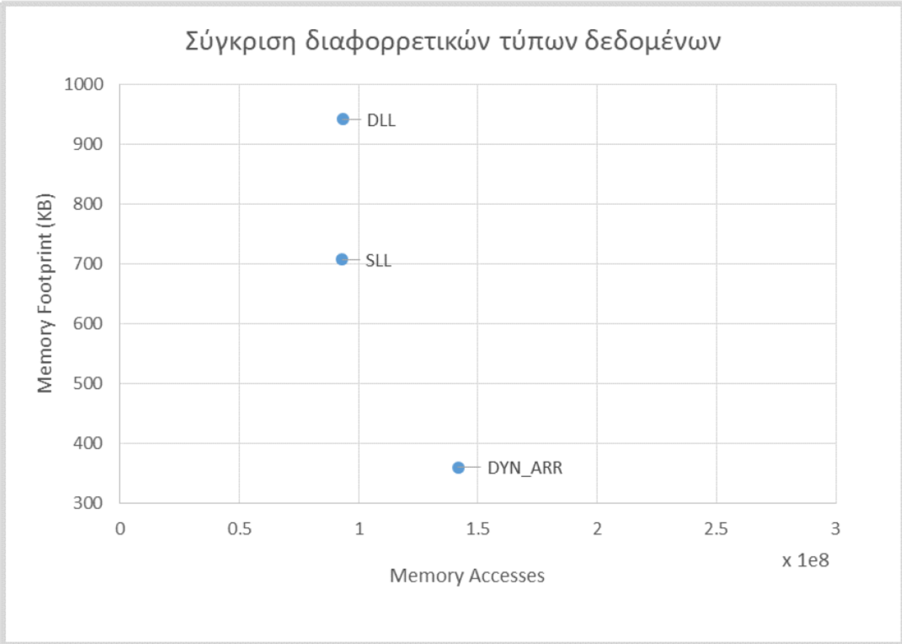


```
Shortest path is 1 in cost. Path is:  0 41 45 51 50
Shortest path is 0 in cost. Path is:  1 58 57 20 40 17 65 73 36 46 10 38 41 45 51
Shortest path is 1 in cost. Path is:  2 71 47 79 23 77 1 58 57 20 40 17 52
Shortest path is 2 in cost. Path is:  3 53
Shortest path is 1 in cost. Path is:  4 85 83 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
```

c) We executed with the script.sh file the application for the following dynamics data structures. Double Linked List - Double Linked List (DLL) and Dynamic Table - Dynamic Table - Double Linked List (DLL). Dynamic Array (DYN_ARR). memory accesses and the maximum size memory required for each implementation we tested (memory footprint).

Struct	Memory Accesses	Memory Footprint (KB)
SLL	92993000	707.7
DLL	93199871	941.8
DYN_ARR	141741903	360.7

We also present the results in a scatter plot diagram.



d) The implementation of a data structure with which the application has the the lowest number of memory accesses (lowest number of memory accesses) is the SLL. e) The implementation of a data structure in which the application has the lowest number of memory latencies. requirements (lowest memory footprint) is DYN_ARR.