Laboratory report Algotirhms and data structures

Index

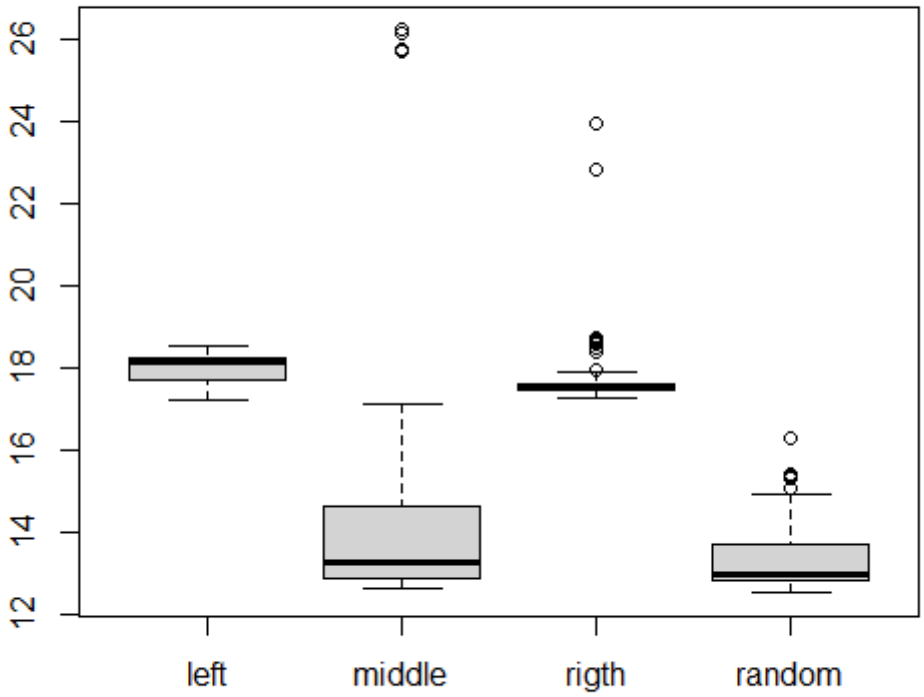
Exercise 1

The first exercise implements a library for using the quicksort and insertion sort algorithms. All of the code is contained in the src folder. The structure of the library is divided into a part allowing for generic data implementation(generic_data.c and corresponding header file) and a second part which implements the algorithms themselves(sorting_lib.c and corresponding header file), as well as a sorting_main.c file which contains a use example of the algorithms which sorts a specific csv file, and finally some unit tests using Unity. The comments in the header files explain the role of all of the used functions.

The binary insertion algorithm works by considering the array to sort in two parts a sorted and unsorted one, as the first part is already sorted, a binary search can be used to determine the position of the next element in the unsorted part. This process, while more efficient than searching by brute force, remains a very slow process compared to the quicksort algorithm.

Pivot

Regarding the quicksort algorithm, two implementation questions were to be answered. choosing the pivot element to partition around and how the partition actually takes place. For the pivot element we identified four possibilities: two nearly identical leftmost and rightmost, as well as random and middle element. The measurements for each of these are as follows:



Pivot test performed on 20 million elements (strings) on a Windows Subsystem for Linux

As can be seen from the boxplots, the random pivot was found to be the most effective
In addition it allows to minimize the encounters of the worst-case performance $O(n^2)$

Partition

As for partition methods, which are known as Hoare and Lomuto partitions, we chose Hoare's partition.
The advantages over Lomuto are:

- Does three times fewer swaps on average than Lomuto
- Creates efficient partitions even when all values are equal

Since the left value is used as a pivot, if we want to implement the random pivot, we must exchange the extracted value with the left

Comparison Quick & Insertion

Number of elements	time(s) of quick sort	time(s) of insertoin sort
10	0.00063	0.000004
100	0.00011	0.000015
1000	0.000235	0.000539
10000	0.002002	0.045521
100000	0.024793	3.937356
1000000	0.332223	238.5105
10000000	2.990886	not available

Tast performed on a bare metal linux installation with the string dataset

Exercise 2

The second exercise implements a particular version of the skip list data structure which can skip certain elements based on an aleatory distribution. The exercise is compiled through the makefile, the library is contained in the src directory, object files are contained in the obj directory and the build directory contains the executables. The library the skiplist data structure wich is linked list where each node as different pointer to other nodes of an abritrary distance and which height is determined by an aleatory variable.

Max Height

As for performance anlysis the variable is represented by the max height value wich determines how many pointers can a single node conatain. If this value is 1 the skiplist will beahave as a simple linked list. It should be noted that for any max heaight value lesser than 6 run time exceedes 10 minutes results will as such be ignored. From the following measures we can deduce that the higher the value of max height is the better the performance, yet the amount gained is quicly amortized and close to none once we pass the 20 mark.

Max Height	Time(s)
6	328.1265
7	116.8988
8	40.9284

Max Height	Time(s)
10	9.5526
15	2.4686
20	2.3896
40	2.4041
60	2.403
100	2.3895
150	2.4052
200	2.394

Tast performed on a bare metal linux installation

Exercise 3

The Heap data structure is a priority queue which represents an array as a tree where the child is in a greater or lesser relation with its parent. In this case the Heap is minimum as such the relation is lesser or equal. Compilation is handled through the Buildjava.sh script which generates all object files within the class directory.

HashTable

In order to maintain the required execution times, the use of hash tables from java libraries was necessary in order to allow every element to be mapped to a certain key within the structure. Through this key parent and child elements can be easily obtained.

As for the extraction, reduction of an element and inserting, the nature of the structure allows log(n) executions through the moveUp and moveDown methods.

Exercise 4

The graph data structure was implemented through three classes representing nodes, arches and the whole graph. The connections are represented by the use of a hash table used as a parameter:

- Graph
  - The graph object holds all the nodes in a HashTable that use the node's value as key
  - UndirectedGraph
    - extension of the graph that allows to represent undirected graphs through an override of 2 metods "addArc" and "removeArc".

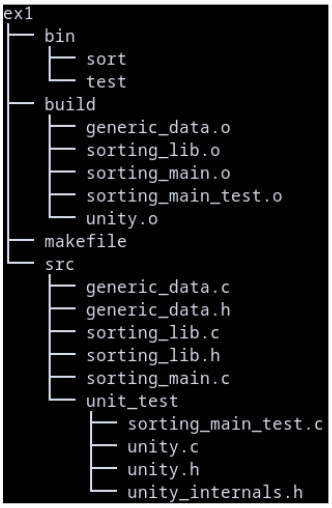
- Node
    - Every node hold an hashtable witch contains all the arcs that start form itself (and it use the value of the arrival node as a key)
- Arc
    - Every arc contain 2 nodes (departure and arrival nodes) and the value of the label

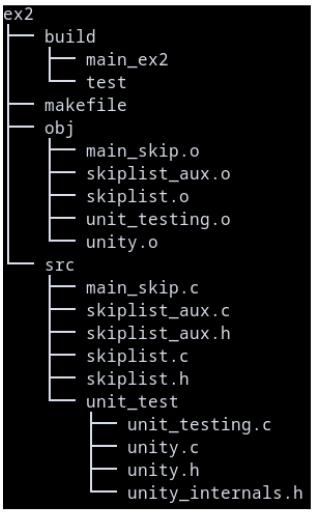
Mapping through hash tables allows for the requested execution times. As for the userspace, all methods are used through the graph class, and use the key of nodes as well as arch labels, to identify the objects if for example a node was to be removed with key 20, graphName.removeNode(20) would be the appropriate way.

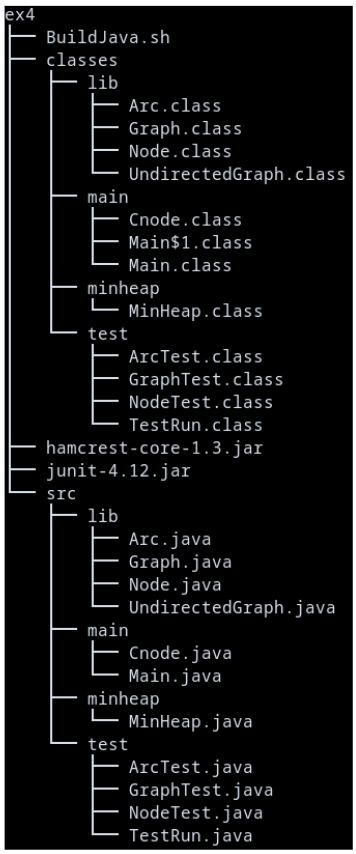
Dijkstra

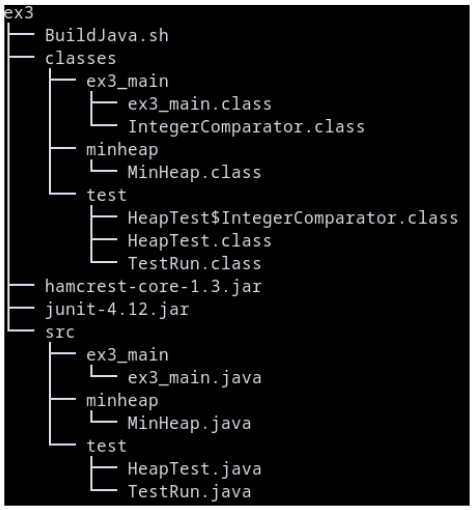
As for the Dijkstra algorithm we use a graph of Strings as keys and Float as labels. Its implementation requires obtaining a certain node's adjacents from the hash table which are returned as Enumerations of arcs. We then iterate over the enumeration and check the labels. We also use a CNode structure containing a Node reference,predecessor reference and cost. If the node’s cost plus the label is lesser than the arrival CNode’s cost is lesser we then it’s cost and predecessor to the newly obtained values. The Cnodes are all contained in the exercise 3 Min heap which allows each time to extract the node with the lesser relative cost.
In the end of the execution every Cnode will contain the shortest path from the origin point, we can then trace the path using the predecessors.

Folder tree


ex1
bin
sort
test
build
generic_data.o
sorting_lib.o
sorting_main.o
sorting_main_test.o
unity.o
makefile
src
generic_data.c
generic_data.h
sorting_lib.c
sorting_lib.h
sorting_main.c
unit_test
sorting_main_test.c
unity.c
unity.h
unity_internals.h


ex2
build
main_ex2
test
makefile
obj
main_skip.o
skiplist_aux.o
skiplist.o
unit_testing.o
unity.o
src
main_skip.c
skiplist_aux.c
skiplist_aux.h
skiplist.c
skiplist.h
unit_test
unit_testing.c
unity.c
unity.h
unity_internals.h


ex4
BuildJava.sh
classes
lib
Arc.class
Graph.class
Node.class
UndirectedGraph.class
main
Cnode.class
Main$1.class
Main.class
minheap
MinHeap.class
test
ArcTest.class
GraphTest.class
NodeTest.class
TestRun.class
hamcrest-core-1.3.jar
junit-4.12.jar
src
lib
Arc.java
Graph.java
Node.java
UndirectedGraph.java
main
Cnode.java
Main.java
minheap
MinHeap.java
test
ArcTest.java
GraphTest.java
NodeTest.java
TestRun.java


ex3
BuildJava.sh
classes
ex3_main
ex3_main.class
IntegerComparator.class
minheap
MinHeap.class
test
HeapTest$IntegerComparator.class
HeapTest.class
TestRun.class
hamcrest-core-1.3.jar
junit-4.12.jar
src
ex3_main
ex3_main.java
minheap
MinHeap.java
test
HeapTest.java
TestRun.java