Algorithms and Data Structure project report

Riccardo Sieve*

* *Facoltà di Informatica, Università degli studi di Torino, Torino, Italia*

20 November 2018

The project is made of four different exercise created upon different tasks (sorting of data, operations on strings, priority queue, graphs). I tried to be as orecise as possible while explaining them.

1 Sorting

The first part of the exercise deals with the implementation of two sorting algorithms:

1. Selection Sort: sorting algorithms with overall complexity of $O(n^2)$
2. Merge Sort: sorting algorithms with overall complexity of $O(logn)$ (in terms of complexity is the best sorting algorithm)

1.1 Implementation

1.1.1 Insertion Sort

Algorithm 1 Insertion Sort pseudo-code

```
for i < ArrayList− > size do
    if ArrayList[i + 1] < ArrayList[i] then
        for j > 0 do
            if ArrayList[j] < ArrayList[j − 1] then
                ArrayList[j − 1] ← ArrayList[j]
            end if
        end for
    end if
end for
```

The implementation of the Selection Sort is rather easy. I start parsing the data structure from the first element; I check that the following element is smaller. In this case I iterate backwards and swap this element if it's smaller than the previous one.

1.1.2 Merge Sort

The merge sort implementation makes use of the Divine and Conquer pattern. I create a recursive function that is called from merge sort that will divide the whole ArrayList into smaller subset, order it and the recreate the ArrayList ordered.

The merge sort function itself is quite short, I first check that I have still have part of the ArrayList to order (I have not reached the last two indexes), if so I set our pivot as $p = l + (r - l)/2$;, I then call the merge sort function twice. The first one from 0 to p, the second one from $p + 1$ up to the last index.
Once I'll have a single element ArrayList I call the merge function.

The merge_sort implementation is the following one:

Algorithm 2 Merge Sort pseudo-code

```
function MERGE_SORT(ArrayList, func, l, r)
    if l<r then
        p = l + (r − l)/2
    MERGE_SORT(arraylist, func, l, p)
    MERGE_SORT(arraylist, func, p+1, r)
    MERGE(arraylist, func, l, p, r)
    end if
end function
```

The merge psudo code is structure as:

Algorithm 3 Merge pseudo-code

```
function MERGE(ArrayList, function, l, m, r)
    fst ← m − l + 1
    snd ← r − m
    for i < fst do
        left[i] ← ArrayList[l + i]
    end for
    for i < snd do
        right[i] ← ArrayList[m + 1 + i]
    end for
    i = j = k = 0
    while i<fst && j<snd do
        if !function(right[j], left[i]) == −1 then
            ArrayList[k] = left[i]
            i + +
        else
            ArrayList[k] = right[j]
            j + +
        end if
        k + +
    end while
    while i < fst do
        ArrayList[k] ← left[i]
        i + +
        k + +
    end while
    while j < snd do
        ArrayList[k] = right[j]
        j + +
        k + +
    end while
end function
```

1.2 Simulation

The first part of the simulation consists in taking an input file — *integers.csv* — (which contains 20 millions numbers) to be ordered with both sorting algorithms

The second part of the simulation takes several steps:

- Open a second file — *sums.txt* — that contains several numbers
- For each number in this file check that it's sum of two elements of the ArrayList
- Create a function with $O(k(logk))$ complexity

1.3 Test

Unit test have been made with the same procedure for both algorithms: I first insert elements inside a temporary ArrayList and then I order them with the corresponding algorithm and test that every element is ordered – I check that each element is not greater than the following one (it can be smaller or equal).

- *test_arraylist_ordered*: I pass 100 random value (from 0 to 100)
- *test_arraylist_reverse_ordered*: I pass 50 elements in reverse order (from 50 to 0) to verify that it order correctly the elements if they're already ordered in reverse order
- *test_arraylist_already_ordered*: I pass 50 elements already ordered and check that it performs correctly

1.3.1 Function implementation

I first call a function that finds the nearest index to the value passed to the ArrayList — this way I'll start from that index.
I then iterate both way — from this index to 0, and from 0 to this index — and check:

- If $ArrayList[i] + ArrayList[index] > value$ I decrement index
- If $ArrayList[i] + ArrayList[index] < value$ I increment i
- If $ArrayList[i] + ArrayList[index] = value$ I got the value

Algorithm 4 Code for sum check

```
for index > 0 do
    for i < index do
        if (ArrayList[i] + ArrayList[j] == value)
then PRINT(Found occurrence)
        else BREAK
        end if
    end for
end for
```

1.3.2 Results

The insertion_sort algorithm failed (it couldn't sort the whole subset in 15 minutes).
The merge_sort algorithm ordered correctly the subset in 20 seconds.

The results are in order with the complexity of the relative algorithms:

- The insertion_sort algorithm has $O(n^n)$ in terms of complexity. Having 20 million records to order can take even days to sort completely
- The merge_sort algorithm instead, is $O(logn)$ in terms of complexity. Ordering the 20 million records con take up to 20,000 clock cycles with a 2 Ghz CPU.

These results are quite expected since the merge_sort has a good complexity. The same simulation was taken with smaller subsets.
With a simple subset – like 1000 or 10000 of elements – both sorting algorithms could get the job done in less than a second. The real difference in the difference of complexity is valuable when a bigger subset – in the

order of millions of records – have to be taken care of. In a real life scenario, I are likely to have a subset of unknown size, using the insertion_sort – as Ill as bubble_sort or selection_sort – could work with small subset – a good sorting algorithm that was developed recently, the Timsort, combine insertion_sort and merge_sort to get the best of both.

1.4 Usage of generics

Since the algorithms have to be implemented with generics, I used opaque type in C with pointers and pointer functions.

The data structure of choice is an ArrayList created in C with opaque type as Ill and have all pre-implemented functions for primitive types. For both insertion_sort and merge_sort a compare function must be passed (the ArrayList implementation already have all compare functions for primitive types).

2 Edit Distance

As part of the second exercise I had to create an algorithm that, given two strings, it would return the number of operation (elimination or insertion) applied to the characters of the first strings in order to obtain in output the second one.

2.1 implementation

The implementation details for the algorithm are as follow:

1. Length check: first of all I check whether or not the two strings have the same length. I furthermore make the following distinction:
   - $length(s1) > length(s2) \rightarrow length(s1) - length(s2)$ eliminations
   - $length(s1) < length(s2) \rightarrow |length(s1) - length(s2)|$ insertions
2. Strings parsing: I iterate over the first string and check, character by character, if there is a pairing one in the second string, if not I will increment the eliminations/insertions counter by one. At the end of the parsing I will know how many insertions and eliminations need to be done.
   I do take care of two possible issues:
   - At the end of the string parsing I will subtract the number found from the $\Delta(length)$ to the eliminations/insertions otherwise I would have a mismatching number of operations.
   - In order to know how many characters from the second string have already been used for the verification, I create an array of $length$ equals to the $length$ of $s2$ filled with $false$ elements. Every time I find an element matching the first string I set the value in the proper position of this array to $true$ so that the corresponding value will not be checked again.

2.2 Simulation

I tested the algorithm with both unit test and the subset given. In both case the algorithm worked smoothly with both the iterative and recursive version, finding several elements with short distance — 0 or 1 — in a reasonable amount of time (the number of elements was quite limited compared to the number of elements I had in the first exercise).

3 Priority Queue

3.1 Data Structure choice in C

For the third exercise, I created the priority queue data structure.
I implemented the structure in C using a Linked List. Even though, normally, it would have been beneficial to use a Binary Heap for this kind of structure, I opted for the List for mainly two reasons:

1. The linked list structure is generally easier to implement and to maintain then a binary heap.
2. Even though a binary heap performs better while deleting a file – its complexity is $O(lgn)$, whereas the complexity of a linked list is $O(n)$ – using a linked list gives us a better performance in the inserting part – $O(1)$ compared to $O(lgn)$ from the binary heap.

As a result, I thought that the benefits of using a linked list where such to make it my implementation choice.

3.2 Implementation in C

The PriorityQueue was implemented by creating a list in which I have the data and the priority associated with it.

When I add an element to the structure it will be appended at the end so that the complexity stays in $O(1)$ and, when removing it, it will parse the list to find the element with the highest priority and then remove it.

3.3 Generics in C

I implemented the generic Priority Queue by having a structure that has a $void*$ element and int as its priority (even though we could have made the priority too as a $void*$ — we might need to use a double as priority but, for the time being, a simple integer is enough).

3.4 Simulation of the algorithm

The data structure have been tested with several primitive type — numbers with single and double precision and strings — to check that it would work with random type too, apart from the unit test. As for the unit test, I developed them to test the Queue with multiple elements of different type — since the element that will be removed is the one with the greatest priority it does not matter the type of the elements.

3.5 Implementation in Java

Since, due to lack of time, I implemented the graph algorithm in Java, I had to implement a Priority Queue to make use of it.

I developed the Queue in Java by making use of a Fibonacci Heap. The idea of creating the Priority Queue with a Heap was to minimise the complexity of the algorithm by making the complexity, in the worst case, of $O(lgn)$ while removing.
It was mostly needed for the graph, since it has several records, therefore computing the whole execution with the structure created in C, whose complexity was $O(n)$ would have made a huge impact on the execution time.

The Heap is made of nodes whose elements are the generic couple $< E, P >$ where:

- E: is the generic element of the queue
- P: is the generic priority associated with the nodes of the queue

Since the heap is more structured than the Linked List, I had to make a set of methods that would aid me in the process:

1. $getElement$ is the method that allows me to get the element at the specified position (since the base structure in which we store data is the ArrayList its complexity is trivial — $O(1)$)
2. $getMinimumElement$ returns the element in position 0
3. $getParent$ returns the parent node to the current one — the index for the parent node is calculated with $getParentPosition$ and it's $(i-1)/2$ in case $i > 0$ (otherwise an exception will be thrown)
4. I also make of use of functions to get the value of the leftChild and the rightChild — both methods have the helper function to calculate the correct position give by the $isSmaller(i, 1)?((2*i)+1) : i$ for the leftChild and $isSmaller(i, 2)?((2*2)+1) : i$ for the rightChild

I make large use of those methods throughout the execution of the main methods of the PriorityQueue: *insert* and *remove*.

As extra methods, I added two to change the key and priority of the nodes in case I need to rearrange the structure.

4 Graph and Prim Algorithm

The fourth and last exercise was about creating a graph and, with it, apply the Prim Algorithm returning a forest as Minimum Spanning Tree.

As mentioned before, I switched from C to Java for this last exercise mainly due to lack of time to complete in a reasonable way in C. Also, as a result of having done the Priority Queue with a Linked List, the drawback of a huge complexity ($O(n)$) in the removing part, would have made the algorithm quite slow.

4.1 Implementation

The implementation of the graph algorithm have made a substantial refactor to add all element needed in the specifications.

At first the algorithm used to iterate over an ArrayList — list of adjacency — and, with the help of a Priority Queue, get all the values to evaluate the minimum total Weight of the Graph and generate the minimum spanning tree with a pseudo code close to this:

Algorithm 5 Prim Algorithm

```
function PRIM(Vertex v)
    unvisitedVremove(v)
    while !unvisitedV.isEmpty() do
        for Iterator i == v.getAdj().iterator() do
            Edgee = i.next()
            if unvisitedV.contains(e.getDestV()) then
                heap.add(e)
            end if
        end for
        e = heap.remove()
        spanningTree.add(e)
        totalCost+ = e.getWeight()
        v = e.getDestVertex()
        unvisited.remove(v)
    end while
end function
```

The main problem with this code, even though it was functional, was the absence of the custom structure of the PriorityQueue and the Minimum Spanning Tree itself — I was just creating a simple tree and not a complete forest.
The last element that was incorrect with the specifications given was to put all given input into an Undirected Graph. This was erroneous into a real-life context: if we take as an example a Network in which some of the nodes are one-way only, I would have

forced them to go in both direction allowing a possible security issue.

The algorithm itself — excluding the two "errors" mentioned before — worked smoothly and with a complexity up to a $E * O(lgV) + O(lgV)$. As I previously wanted to do in C, using a HashMap to store Data instead of a simple ArrayList could have improved performance as well. Into a HashMap, the lookup time is trivial — $O(1)$ — when well formed.

Therefore, while refactoring the code, I took into account the HashMap for the storage of the Data and the Sets as return value for the getter to slightly improve performance.
As a last element, I created the code for both Directed and Undirected Graphs.

The new Prim algorithm was:

Algorithm 6 Prim Algorithm

```
function PRIM(Vertex v)
    map = new HashMap
    forest = new UndirectedGraph
    queue = new PriorityQueue
    for Vertex i: getVertices() do
        i.setKey(∞)
        i.setParent(null)
    end for
    v.setKey(0)
    for Vertex i: getVertices() do
        node = new Node
        map.put(v, node)
    end for
    while !queue.isEmpty() do
        minV = queue.remove().getElem()
        forest.addVertex(minV)
        if minV.getParent() != null then
            forest.addEdge(minV.getParent
             minV
             getWeight(minV().getElem())
        end if
        for Vertex i: getNeighbours(minV)) do
            if queue.contains(map.get(v)) then
                if v.getKey() > getWeight(minV, v)
then
                    v.setKey(getWeight(minV, v))
                    v.setParent(minV)
                    queue.changePrio(map.get(v)
                     getWeight(minimumVertex, v))
                end if
            end if
        end for
    end while
end function
```

Apart from the initial state of the elements to be more correct in accord to the Prim Algorithm, the algorithm itself does not differ that much from its first version. What really changes in the structure of this excercise is the presence of the HashMap that improves the overall performance of the algorithm and the creation of the forest that was requested in the assignment.

4.2 Generics

Both Directed and Undirected Classes extends from Graph and, as such, both implement generics as a couple $< V, E >$ where V is the generic element for the vertices of the graph, and E is the generic element for the edge of the graph.

During the implementation of the Usage file these generics have been implemented as, respectively, Strings and Double.

4.3 Usage and Test

The Graph and the Prim Algorithm have been tested with JUnit5 (just like the Priority Queue) testing the graph with no element, one element or multiple elements.

As per the usage part, I used the dataset given to put all elements iin the graph with the couple $< String, Double >$ for the vertex and weight. All the values in the dataset have been placed into an Undirected graph to apply the Prim Algorithm over it. As a final print out, I print both values displaying:

- *Number of Vertices*: number of vertices
- *Number of Edges*: number of edges
- *Total Weight*: weight calculated all over the structure

before and after the creation of the forest. As a result, the number of vertices remained the same — this is trivial since we don't remove any vertex from the graph — whereas the number of edges is lowered from 48055 to 18637 and the total weight from $437381401.867\ km$ to $89939.913\ km$