Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

March 7, 2014

1 Programming Assignment 01

1.1 Introduction

The purpose of this assignment is to experience some of the problems involved with implementing an algorithm (in this case, a minimum spanning tree algorithm) in practice. As an added benefit, we will explore how minimum spanning trees behave in random graphs.

For this assignment, the programming language used was Java SE 1.7 due to the familiarity with the language and also some data structures are already implemented which will make the development process a little faster

We will be considering complete, undirected graphs. A graph with n vertices is complete if all possible $\binom{n}{2} = \left(\frac{n!}{2!(n-2)!}\right) = \left(\frac{n(n-1)(n-2)!}{2(n-2)!}\right) = \left(\frac{n(n-1)}{2}\right)$ edges are present in the graph.

During this assignment the following types of graphs are considered:

- Complete graphs on n vertices, where the weight of each edge is a real number chosen uniformly at random on $[0, 1]$.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit square. (That is, the points are (x, y), with x and y each a real number chosen uniformly at random from $[0, 1]$.) The weight of an edge is just the Euclidean distance between its endpoints.
- Complete graphs on n vertices where the vertices are points chosen uniformly at random inside the unit cube (3 dimensions) and hypercube (4 dimension). As with the unit square case above, the weight of an edge is just the Euclidean distance between its endpoints.

The first goal is to determine in each case how the expected (average) weight of the minimum spanning tree (not of individual edges, the whole MST) grows as a function of n. This will require implementing an MST, as well as procedures that generate the appropriate random graphs. (It is important to check what sorts of random number generators are available on the implementing system, and determine how to seed them, say with a value from the machine's clock.) Any MST algorithm (or algorithms) may be implemented, however, the choice must be done carefully.

For each type of graph, several values of n must be chosen to test. For each value of n, the code must be run on several randomly chosen instances of the same size n, and compute the average value for the runs. The values must be plot vs. n, and interpret the results by given a simple function $f(n)$ that describes the plot. For example, the answer might be $f(n) = \log n$, $f(n) = 1.5\sqrt{n}$ or $f(n) = \frac{2n}{\log n}$.

Even though the Krushkal or Prim's algorithms run in polynomial time; with big values of n the running time might not be feasible for a home desktop computer. Section 2 briefly explains the main non optimized algorithm and other other algorithms that were used for the implementation. Section 3 explains the problems with using the non optimized version and the running that had to be done in order to compute the MST of graphs with n vertices, where $n > 2^13$. Section 4 presents how the experiments were run the their results. Section 5 tries to deduce the simple function $f(n)$ that describes the results. Finally section 6 presents the interesting experiences with this assignment.

1.2 Algorithms used

First to generate the graph we need to generate the vertices and then the edges. For simplicity, the name assigned to each vertex is a number. Also, to create a vertex (node in the code) its corresponding coordinate must be provided. The generation of this coordinate is very simple if we create an array of size *dimension* and then we just iterate over each bucket in the array to assign a random number in $[0, \ldots, 1]$.
The random numbers were generated using the class java.util.Random, which generates pseudorandom, uniformly distributed values between 0 and 1. The seed is taken partially from the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

After the vertices have been generated, we proceed to generate the edges. The vertices and edges are going to be stored in a Set, to make the code clearer when implementing the algorithm. To generate the edges, we need to create only $n(n-1)/2$ edges. This is done by generating an array from the set of vertices, so we can have a notion of position or index of an element, and then we iterate in a nested loop to generate all the require edges.
To generate the $n(n-1)/2$ edges, we just take the first element and then generate an edge with the others $n-1$ elements, so the first loop will go from 1 to $n-1$, because the last vertex will already have an edge to every other vertex before it.
When the edge is created, we iterate over the coordinates of each vertex so we calculate the Euclidean distance.

Due to the advantages of having several data structures already programmed in Java, Krushkal's algorithm is simple to implement (as many greedy algorithms). The program receives an object of type *Graph* and sorts the edges, then it just greedily iterates over all the edges and assign them to a set as long as not both vertices are already in the set. The loop finishes when the number of edges is equal to the number of vertices minus one.

Just for academic purposes, the MergeSort algorithm was implemented to compare performance to sorting mechanism that Java already implements for arrays, which is also used for lists.

1.3 Running time problems

For thie experiments, I work with a MacBook Aluminum late 2008, with a 2 GHz Intel Core 2 Duo processor and 8 GB 1067 MHz DDR3 in RAM.

All the experiments run for this assignment are with a complete graph, so generating $|V|(|V|-1)/2$ edges might not feasible for certain values. The number of nodes the program is expected to handle is shown in Table 1, which helps to determine what values can be computed straightforward and what values required special treatment. If we consider $n = 2^k$, then the number of edges is $2^k(2^k-1)/2 < 2^k \cdot 2^k/2 = 2^{2k}/2 = 2^{2k-1}$, then we have an upper bound for the number of edges as a power of 2.

Table 1: Cardinality of nodes and edges

Nodes		Edge	Upper bound	
16	2^4	120	128	2^7
32	2^5	496	512	2^9
64	2^6	2,016	2,048	2^{11}
128	2^7	8,128	8,192	2^{13}
256	2^8	32,640	32,768	2^{15}
512	2^9	130,816	131,072	2^{17}
1,024	2^{10}	523,776	524,288	2^{19}
2,048	2^{11}	2,096,128	2,097,152	2^{21}
4,096	2^{12}	8,386,560	8,388,608	2^{23}
8,192	2^{13}	33,550,336	33,554,432	2^{25}
16,384	2^{14}	134,209,536	134,217,728	2^{27}
32,678	2^{15}	533,909,503	536,870,912	2^{29}

Considering how much RAM I have, we have that 8 GB = 8,192 MB = 8(1,024) MB = 8,388,608 KB = $8(1,024)^2$ KB = 8,589,934,592 B = $8(1,024)^3$ B = $8(2^{10})^3 B = 8(2^{30})$ B, so this gives an idea of how much I can compute.

Instead of running the algorithm and how it will slow down as n increases up to a point that computing $n = 2^1 3$ might take too long, I decided for a top down approach. This way, I can run my experiments when I am sure that they will finish in a reasonable time so I can write this report.

if I assign 4,096 MB for memory heap and I tried to create a complete graph with $16,384$ nodes in $\mathbb{R}^4$, I get an OutOfMemoryError while trying to create the edge number 38, 293, 002. If I increase the memory heap to 6,144 MB, then I get the error while creating the edge $50,331,651$. Hence it is not going to be possible to create $134,217,728$ edges and keep all of them at the same time in memory.

So for the purposes of the experiments I will assume that somehow we are able to retrieve the graph along with its edges. This simulation can be performed by implementing the *List*, *Iterator* or *Iterable* interface and then generating the Edges on the go.
Even though the edges are generated as requested, instead of generating all of them at once, if I just keep retrieving them and storing them in another data structure, I will have the same problem that I had at the beginning.

The approach to solve the memory problem (which later would become in a processing capability problem due to swapping) I decided to retrieve $1,000,000$ edges and then applied the non optimized version that I described in section 2. After the non optimized version finishes, we have reduce the number of edges from $1,000,000$ to just $n-1$. At the end of the loop, I called again the non optimized version for the remaining $< 1,000,000$ edges.
The above tuning is very similar to how QuickSort or MergeSort work. We are retrieving the $n-1$ edges with least minimum weight for each block of $1,000,000+n-1$ elements.

The edges are generated in order, so after the first $n-1$ edges retrieve, the loop could stop and we would have a spanning tree T. If we did not have any more edges, then the tree T would become a minimum spanning tree.

Even though the above approach might seem to work, the correct data structures should be used. Before we reach $1,000,000$ edges, those edges have to be stored in some data structure. First I chose an *ArrayList*, but it had several problems. First, after $1,000,000$ have been added, then this list have to be sorted. Sorting a list of $1,000,000$ took too much to consider it an option for the experiments when $n > 2^{12}$.
To avoid the problem of sorting, I decided to use a *TreeSet* instead, that way, as each element is added it will already be placed in the correct spot. The only inconvenient is that my non optimized version of Krushkal is expecting a *List* instead of a *Set*, so before calling the non optimized version of Krushkal, I assign the elements in the *TreeSet* to an *ArrayList*.
The above solution might have worked but an *ArrayList* is implemented using a regular array, and when the non optimized version of Krushkal is removing always the first element of the list. This causes a problem because when the number of elements in the array drops below certain threshold, to avoid wasting memory, a new shorted array is created and the elements are copied to this new array. This might make look that deleting or

adding n elements to an *ArrayList* is in order of $O(n^2)$, but copying to a new shorter (or greater) array does not happen all the time, so using the amortized analysis, the running time of the operations are in $O(n)$.

Even though the amortized analysis is correct, the constant factor that distributes the worst case was impacting the execution time over all the program, so more tuning had to be done.
Hence instead of using an *ArrayList*, I chose to use a *LinkedList*, which is very fast for insertions and deletions which are going to be in constant time since I always delete only the first element of the list.

Therefore, after all the above changes I would expect the code to be fast enough, but still after some tests, it was not. After running some tests I notice that the time between blocks was not constant, it was increasing as the number of blocks treated increases. This was because I was not clearing the sets and lists containing the edges that I had already used. For instance, I assume that I have $1,000,000$ and my non optimized version of Krushkal gets the $n-1$ require edges from the first $n-1$ edges in the list, then when retrieving the next $1,000,000$, instead of searching through only $1,000,000$ elements, it was working with $1,000,000-(n-1)+1,000,000+n+1 = 2,000,000$. So after a while, the program would behave just like the non optimized version with a huge set of edges.
Therefore, the final tuning was to clear the *TreeSet* where I kept the sorted edges and the *LinkedList* where I kept the elements for the non optimized version of Krushkal.

Finally, I ran a text with all the expected values of n to be used in the experiments with dimension 4 to see the running time and verify that all the test cases will finish in a time less than the age of the universe.
For the experiments, I consider $2,048$ as threshold to differentiate non-big graphs from big graphs. So any graph with more than $2,048$ edges, is going to used the optimized version. The Table 2 shows the running time of the experiments for different values of n.

Notice that the running time for $32,678$ was around $2,191$ seconds, which means that it took around 37 minutes to finish. If I run the test 5 times, it would take around 150 minutes and if I do that for dimensions 2, 3 and 4, then it would take around 450 minutes, which is a little more than 7 hours. Actually during experiments, must of the time was spent for the case where $n > 4096$.

1.4 Experiments

All of the experiments were defined as test cases using JUnit 4. This helped me keep track of what experiments I had done and was missing.
Also, since the experiments were expected to run during several hours but no more than a day, I used SL4J and Log4J to log the require information into a file.
To ease how the required jars are downloaded, as well as how the project is compile and test cases run, I decided to create a Maven project.

Table 2: Running time with different values of n

n	Time in milliseconds
16	0
32	3
64	5
128	9
256	239
512	436
1,024	1,800
2,048	14,371
4,096	39,714
8,192	177,060
16,384	466,158
32,678	2,190,921

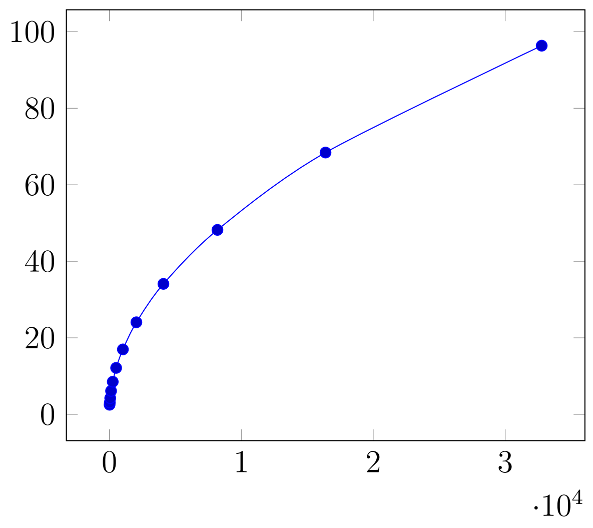

Figure 1: Plot of values for dimension 2

1.5 Deducing $f(n)$

1.6 Conclusion

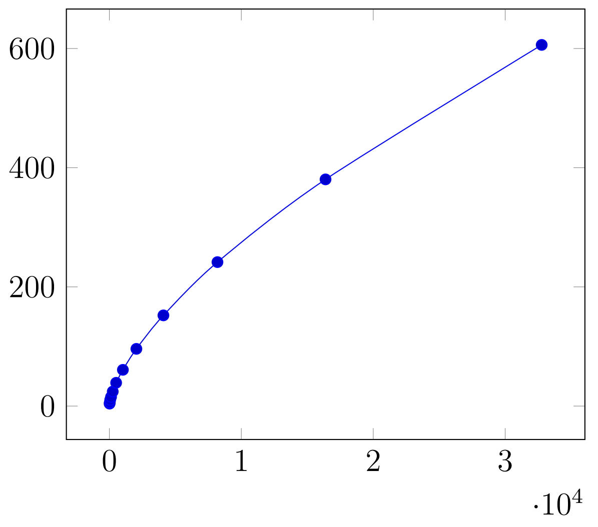

Figure 2: Plot of values for dimension 3

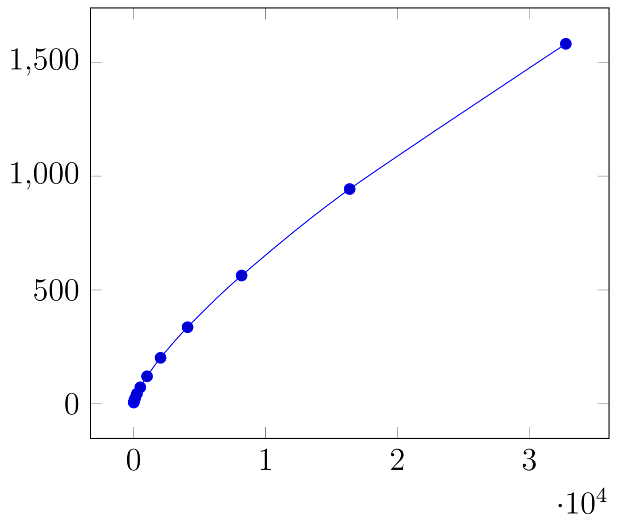

Figure 3: Plot of values for dimension 4