

Data Structures Project

Communications Network

Nikolay Vasilev
Department of Computer Science
Henning Lübbers
spring 2010

Table of Contents

Requirements and Design.....	3
Underlying Model.....	3
Data Structures and Algorithms.....	3
Merge Sort.....	3
Kruskal's Algorithm.....	5
UML Diagram.....	7
Implementation.....	8
Introduction.....	8
Analysis of Data Structures and Algorithms.....	8
Merge Sort.....	8
Kruskal's algorithm.....	9
Testing.....	9
Programme Limitations.....	10
Installation and Running Instructions.....	10
Course Feedback and Conclusion/Reflection.....	10
References.....	11

1.Requirements and Design

a)Underlying Model

The objective of this Data Structures project is to produce a piece of software that will provide the end user with an efficient plan for updating a given telecommunications network. The programme will receive a text file as its input, which will contain (an) existing communications grid(s) (might be several) with the names of cities (in pairs), their geographical distance (in kilometres), as well as the renovation cost per kilometre of connection (in Euros or another currency). Based on the data the programme has extracted from the text file, it will compute and output (on another file or the command line, depending on the user's preference) a list of the cities and the links that need to be updated so that the overall cost be as small as possible and every city remain connected to the grid it was initially connected to.

The abstract model behind the communications network is essentially a weighted undirected graph, which may or may not be connected. The result will be a minimum spanning tree for each/the graph in the input. Each city on the network constitutes a node on the graph and the connections between the cities will be the edges between them. The cost to update the entire link will be regarded as the weight of the edge of the graph. Hence the minimum spanning tree will actually show the cheapest rather than the shortest solution (in terms of distance) to the communications network updating problem.

b)Data Structures and Algorithms

In order to produce the desired result the programme will make extensive use of Kruskal's algorithm. Since the algorithm itself makes no assumptions regarding the structure of the graph whatsoever (even that the graph should be connected), no "security checks" will be performed before the Algorithms is harnessed. The data types that will according to preliminary estimates be used are a few programmer defined classes:

1. Vertex (for the vertices of the graph). Each vertex will most probably have a name. Other fields will be added, should the need arise in the programming process.
2. Edge (for each edge of the graph). The edge will have at least the following fields:
 - a) weight – the overall cost to update the entire link
 - b) pointers to each of the two vertices it connects
3. Set (for the needs of the algorithm). The set will mainly consist of vertices.

•Merge Sort

Merge Sort is used for the purposes of Kruskal's algorithm. All of the graph's edges (whether the graph is connected or not) are saved in an ordinary array of pointers, which is later sorted by virtue of the method in question. Since Edge is not a primitive data type, edges are compared based on the values of their *weight* fields. Merge Sort is a recursive divide-and-conquer algorithm. Here's a description given in Thomas Cormen's *Introduction to Algorithms*:

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already sorted.

The key operation of the Merge Sort algorithm is the merging of two sorted sequences in the

“combine” step. To perform the merging, we use an auxiliary procedure Merge(A, p, q, r) (with a bit modified parameters in this application), where A is an array and p, q , and r are indices numbering elements of the array such that $p \leq q < r$. The procedure assumes that the arrays $A[p \dots q]$ and $A[q+1 \dots r]$ are in sorted order. It **merges** them to form a single sorted subarray that replaces the current subarray $A[p \dots q]$.

Our Merge procedure takes time $O(n)$, where $n = r - p + 1$ is the number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are checking just two top cards. Since we perform at most n basic steps, merging takes $O(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. The idea is to put on the bottom of each pile a **sentinel** card, which contains a special value that we use to simplify code (in the actual programme superseded by a loop and variables that keep a record of the indices in the left and right subarray). Here, we use ∞ as the sentinel value, so that whatever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```

Merge( $A, p, q, r$ )
1       $n_1 \leftarrow q - p + 1$ 
2       $n_2 \leftarrow r - q$ 
3      create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
4      for  $i \leftarrow 1$  to  $n_1$ 
5          do  $L[i] \leftarrow A[p + i - 1]$ 
6      for  $j \leftarrow 1$  to  $n_2$ 
7          do  $R[j] \leftarrow A[q + j]$ 
8       $L[n_1 + 1] \leftarrow \infty$ 
9       $R[n_2 + 1] \leftarrow \infty$ 
10      $i \leftarrow 1$ 
11      $j \leftarrow 1$ 
12     for  $k \leftarrow p$  to  $r$ 
13         do if  $L[i] \leq R[j]$ 
14             then  $A[k] \leftarrow L[i]$ 
15                  $i \leftarrow i + 1$ 
16             else  $A[k] \leftarrow R[j]$ 

```

In detail the Merge procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p \dots q]$, and the line 2 computes the length n_2 of the subarray $A[q + 1 \dots r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively in line 3. The **for** loop of lines 4-5 copies the subarray $A[p \dots q]$ into $L[1 \dots n_1]$, and the **for** loop of lines 6-7 copies the subarray $A[q + 1 \dots r]$ into $R[1 \dots n_2]$. Lines 8-9 put the sentinels at the ends of the arrays L and R . Lines 10-17 perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

At the start of each iteration of the **for** loop of lines 12-17, the subarray $A[p \dots k - 1]$ contains the $k - p$ smallest elements of $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A ...

... We can now use the Merge procedure as a subroutine in the merge sort algorithm. The procedure Merge-Sort(A, p, r) sorts the elements in the subarray $A[p \dots r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p \dots r]$ into two subarrays: $A[p \dots q]$, containing $\lfloor n/2 \rfloor$ elements, and $A[q + 1 \dots r]$, containing $\lfloor n/2 \rfloor$ elements.

```

Merge-Sort( $A, p, r$ )
1   if  $p < r$ 
2       then  $p \leftarrow \lfloor (p+r)/2 \rfloor$ 
3       Merge-Sort( $A, p, q$ )
4       Merge-Sort( $A, q+1, r$ )
5       Merge( $A, p, q, r$ )

```

To sort the entire sequence $A = \{A[1], A[2], \dots, A[n]\}$, we make the initial call Merge-Sort($A, 1, \text{length}[A]$), where once again $\text{length}[A] = n$. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n .

•Kruskal's Algorithm

The description of this algorithm is also taken from Thomas Cormen's *Introduction to Algorithms*.

Corollary 23.2

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E . Let A be a subset of E that is included in some minimum spanning tree for G , and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A , then (u, v) is safe for A .

Kruskal's algorithm is based directly on the generic minimum-spanning-tree algorithm given in Section 23.1 (see below)

```

Generic-MST( $G, w$ )
1    $A \leftarrow \emptyset$ 
2   while  $A$  does not form a spanning tree
3       do find an edge  $(u, v)$  that is safe for  $A$ 

```

```

4           $A \leftarrow A \cup \{(u, v)\}$ 
5      return A

```

It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight. Let C_1 and C_2 denote the two trees that are connected by (u, v) . Since (u, v) must be a light edge connecting C_1 to some other tree, Corollary 23.2 implies that (u, v) is safe for C_1 . Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

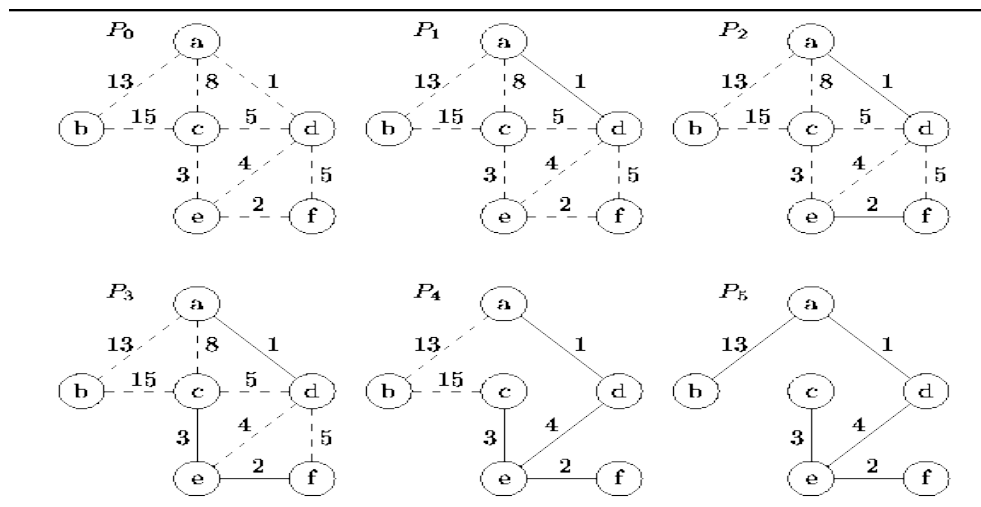
Our implementation of Kruskal's algorithm is like the algorithm to compute connected components from Section 21.1. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. The operation Find-Set(u) returns a representative element from the set that contains u . Thus, we can determine whether two vertices u and v belong to the same tree by testing whether Find-Set(u) equals Find-Set(v). The combining of trees is accomplished by the Union procedure.

```

MST-Kruskal( $G, w$ )
1       $A \leftarrow \emptyset$ 
2      for each vertex  $v \in V[G]$ 
3          do Make-Set( $v$ )
4      sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5      for each edge  $(u, v) \in E$ , taken in nondecreasing order by weight
6          do if Find-Set( $u$ )  $\neq$  Find-Set( $v$ )
7              then  $A \leftarrow A \cup \{(u, v)\}$ 
8                  Union( $u, v$ )
9      return A

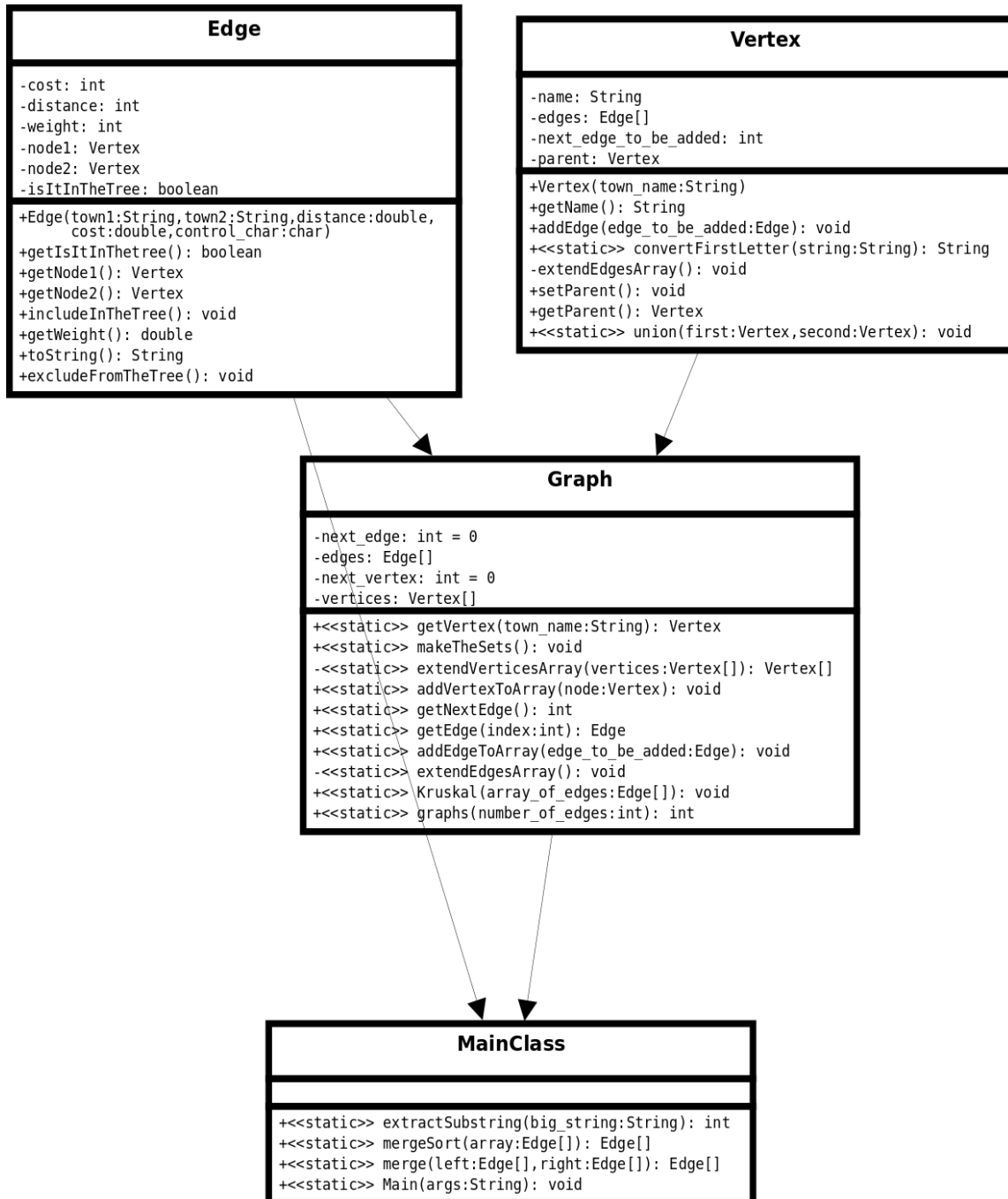
```

Kruskal's algorithm works as shown below. Lines 1-3 initialise the set A to the empty set and create $|V|$ trees, one containing each vertex. The edges in E are sorted into nondecreasing order by weight in line 4. The **for** loop in lines 5-8 checks, for each edge (u, v) , whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, the edge (u, v) is added to A in line 7, and the vertices in the two trees are merged in line 8.



c)UML Diagram

Since the programmer noticed this requirement a bit later, the following diagram reflects the state of the programme as it is now, not how it was designed at first.



2.Implementation

a)Introduction

On the whole the programme fulfils the criteria set forth in the previous section. With the benefit of hindsight now, the programmer contemplates how the algorithms could have been implemented more efficiently by applying "niftier" data structures. For example, edges and vertices are now saved in ordinary lists of objects, whereas a far more efficient alternative would have been a binary search tree. Such an approach would have also rendered Merge Sort obsolete. Looking back now, however, the benefits of the data structure would scarcely offset the complexity the implementation (enormous as it seems now) would have involved.

Another cause for concern is the way the class Edge was designed and coded. Since the underlying algorithm required that an array of edges should be sorted by weight, good programming style would have enjoined that this be done by implementing the *Comparable* interface through its *compareTo* method. Instead the programme carries out the comparisons by directly retrieving the value of each Edge object's *weight* field, which has not as yet evinced any flaws, but does not allow for much further development.

Last but not least, the sets operations are performed in a rather clumsy way. Instead of a separate class that was to take care of the unions and the like (as the instructor had suggested himself), the programmer opted for a far simpler solution, which does the job, if slow and ungainly.

b)Analysis of Data Structures and Algorithms

In this section only Merge Sort and Kruskal's algorithm will be honoured with some attention. Again, the analyses are taken from Thomas Cormen's textbook *Introduction to Algorithms*.

•Merge Sort

Analysis of merge sort

Although the pseudo code for Merge-Sort works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. In Chapter 4 we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

Divide: *The divide step just computes the middle of the subarray, which takes constant time. Thus $D(n) = O(1)$;*

Conquer: *We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.*

Combine: *We have already noted that the Merge procedure of an n -element subarray takes time $O(n)$ so $C(n) = O(n)$.*

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $O(n)$ and a function that is $O(1)$. This sum is a linear function of n , that is $O(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives us the recurrence for the worst case

running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

In Chapter 4, we shall see the "master theorem," which we can use to show that $T(n)$ is $O(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $O(n \lg n)$ running time, outperforms insertion sort, whose running time is $O(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to the above recurrence is $T(n) = O(n \lg n)$. Let us rewrite the recurrence as

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps...

...To compute the total cost represented by the recurrence, we simply add all the costs of all its levels. There are $\lg n + 1$ levels, each costing cn , for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant c gives the result of $O(n \lg n)$.

Space complexity is calculated to be $O(n)$. (Data Structures 2009 lecture slides – page 324)

•Kruskal's algorithm

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structure. We shall assume the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and path-compression heuristics (a little less efficient method used in this particular project, but difference is not dramatic), since it is the asymptotically fastest implementation known. Initialising the set A in line 1 [of Kruskal's algorithm] takes $O(1)$ time, and the time to sort the edges in line 4 is $O(E \lg E)$. (We will account for the cost of the $|V|$ Make-Set operations in the **for** loop of lines 2-3 in a moment.) The **for** loop of lines 5-8 performs $O(E)$ Find-Set and Union operations on the disjoint-set-forest. Along with the $|V|$ Make-Set operations, these take a total of $O((V + E) \alpha(V))$ time, where α is the very slowly growing function defined in Section 21.4. Because G is assumed to be connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E \alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

The algorithm itself does not require any auxiliary variables besides the array of edges. Therefore space complexity can be considered to be zero.

c) Testing

The tests performed outside the standard JUnit tests were directed at the correctness of the result output by the programme and the file opening procedure (including but not limited to the validity of the input format, the file name, etc.). The tests were conducted manually and can be easily reproduced by simply bringing the programme into use and trying different inputs and file names (the instructor's sample input is a good starting point...at least it was in the programmer's tests). The

two test targets, namely the file opening procedure and the correctness of the programme's output, can be checked for example as follows:

- 1) File opening procedure: the test can be reproduced by entering names of files that do not exist in the directory containing the bin folder. Special attention should be paid to the filename extensions. Windows demands that the extension be included in the filename, whereas the programme will complain in such cases when executed on Linux.
- 2) Result validation: the only way this can be tested is by manually computing the minimum spanning tree and comparing it to the programme's output. The Sample file accompanying this package is how the tests were conducted. The input was manipulated to include new edges that were (not) part of the existing graph etc.

All detected bugs and glitches were rectified and there should not be any left (hopefully).

The achieved code coverage was about 90%, which is probably the best this programmer is capable of...at least at this stage.

d)Programme Limitations

Probably the most notorious programme limitation is the time complexity the algorithms add to the overhead incurred by the programmer's mistakes. Kruskal's algorithm takes definitely much more time to perform its task mainly due to the neglected union-by-rank. The fact that the edges are stored in an ordinary list rather than a binary tree has also taken its toll on the programme's performance. This has another negative side effect – the number of edges in the input is artificially limited to 2^{31} as they are indexed by a 32-bit integer.

e)Installation and Running Instructions

The programme can be easily executed by running the class called MainClass.java. No command line arguments are used. The programme asks for a file name and upon successfully opening the file, which field it should regard as the weight of the edge (the distance or the product of cost and distance). Option 'D' stands for distance and anything else will mean cost. Next the programme asks whether the user wants the result on the command line or another file. If the user chooses the latter option, the programme queries the name and closes upon successfully writing the file.

f)Course Feedback and Conclusion/Reflection

One of the reasons why I decided to embark on this course was quite honestly the linguistic benefit. While I realise I do not at present boast enviable Finnish skills, completing this project in German (a language I had once considered myself fluent in and had long thought those times were dead and gone) promised at first to be a greater challenge. Reading the first emails I sent the instructor I was enraged by how many typos I managed to make. I believe he was pretty pessimistic himself about me not only coping with the programming, but overcoming the language barrier that existed between us, as well. I had the feeling he expected me to exclaim any moment entreating: "All right, that's enough, English, please!" This is probably the reason why he refrained from correcting me now and again. It was very frustrating to walk out of a meeting turning our conversation over in my mind and to remember the blatant mistakes I had made, which I thought I would have never made 5 years earlier. So in a nutshell, my advice is to correct and remark on errors whenever they occur. Progress cannot be achieved otherwise, rather the problem is likely to deteriorate into pompous ignorance. I know people might react differently and sometimes even abruptly, but everything comes at a cost, and if a person really wants to learn, they will not take offence at the fact that they

have stood corrected.

3. References

1. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein (2002) *Introduction to Algorithms*. People's Republic of China: The MIT Press
2. Florèen, Patrik (2009) *Tietorakenteet*, Helsinki University
<http://www.cs.helsinki.fi/u/floreen/tira/k09/luennot.pdf>
3. Illustration of Kruskal's algorithm
<http://www.brpreiss.com/books/opus6/html/img2464.gif>