Comparing minimum spanning tree algorithms

Igor Podsechin

Tampereen lyseon lukio

Tietotekniikka

Abstract

This essay compares three different minimum spanning tree algorithms, namely Prim’s, Kruskal’s and Boruvka’s. It introduces the relevant graph theory concepts that are needed and the working principles behind the algorithms are explained briefly. Two different data structures for graphs are considered and their weaknesses and strengths evaluated. The answer to the question which one is superior is not definite. Implementations of the three algorithms that are written in Java are then presented and the various techniques that were used to achieve the aims are covered. Next an experiment is conducted in which the order of the starting graph is changed and each algorithm’s runtime is then measured. As a result it is possible to draw an order versus time graph. It becomes apparent that the algorithms follow their established time complexities fairly well and that out of the three, Prim’s algorithm is the slowest, while Kruskal’s and Boruvka’s are very close with each other on performance. Finally the efforts are summarized and hybrid algorithm is proposed.

Tiivistelmä

Essee vertailee kolmea algoritmia, joiden tarkoitus on löytää pienin mahdollinen virityspuu. Nämä ovat Primin, Kruskalin ja Boruvkan algoritmit. Aluksi essee johdattelee graafiteoriaan ja käy läpi algoritmeille olennaisia käsitteitä lyhyesti. Jotta informaatio voitaisiin säilyttää tehokkaasti vertaillaan kahta erilaista tietostruktuuria, mutta mikään niistä ei ole yksiselitteisesti tehokkaampi kuin toinen. Tämän jälkeen esitetellään kuinka kolmea algoritmia voidaan toteutta Java ohjelmointikielellä. Antamalla ohjelmalle syötteitä, voimme vertailla jokaista algoritmia ja sen käyttäytymistä erikokoisten graafien kanssa mittaamalla käytetyn ajan. Saaduista tuloksista on mahdollista piirtää kaavio jossa y ja x akseleina ovat aika millisekunteina ja graafin suuruus. Selviää että kaikki algoritmit noudattavat aikavaativuusluokkaa suhteellisen hyvin ja lisäksi todetaan että Prim on algoritmeistä hitain, kun taas Kruskal ja Boruvka ovat hyvin lähellä toisiaan suorituskyvyltään. Lopuksi on yhteenveto ja myös ehdotus hybridialgoritmille.

Table of contents

1. Introduction

Graph theory is an important field of mathematics that has been studied carefully ever since its inception, which is commonly regarded as the 18th century. It was Euler who famously introduced his historical problem of “Seven Bridges of Königsberg”, in which the goal is to cross every bridge in the town only once. In his paper he came to the conclusion that no solution exists and this consequently laid the foundations to the study of graphs. The subject has spawned many other fascinating problems nearly all of which are rather concrete in their nature and quite easily comprehensible, but the different algorithms for solving them pose an interesting field of study offering a source for analysis and research. These examples include the travelling salesman, the four colours problem and the minimum spanning tree question, which will be the topic of research in this paper.

After the introduction of computing machines in the twentieth century it became possible to represent mathematical ideas and concepts as bits of data. With the great increase in computing power and storage capacity, it is now possible to perform very large calculations, which would take extremely long periods of time if done by hand, in mere fractions of a second. This has given the opportunity to implement previously established algorithms, but has also given rise to completely new ones, which previously had no applications. This is also the case with the theory of graphs, but algorithms that were made to search for the minimum spanning tree were already developed before the introduction of computers.

In fact the first minimum spanning tree algorithm was hypothesised by a Czech mathematician named Otakar Boruvka in his paper published in 1926[1]. His purpose was to connect the Moravian electric grid as efficiently as possible. Today there are a number of these algorithms and new ones are being worked on as well in the hope of finding more efficient solutions. Minimum spanning tree algorithms are covered in the graph theory chapter of almost any high school mathematics textbook. For example, the IB textbook presents two different algorithms for solving the problem, but does little to actually clarify: which one of them is more efficient? This sparked the author’s interest and the purpose of this paper is to compare the three algorithms and hopefully find an answer to this question.

Graph theory has been applied in various areas of studies, such as dealing with electronic circuitry, links between entities, such as airline paths and also quite importantly in computer science,

[1] http://citeseer.ist.psu.edu/old/413530.html

especially in the representation of different kinds of networks. One example could be the making a connection between computers in a network with the minimal cost. This is why studying algorithms concerning graphs is a relevant topic today.

2. The research question and structure

The aim is to study three minimum spanning tree algorithms, namely Prim's, Kruskal's and Boruvka's algorithms all of which are named eponymously. First of all, we will define the problem which these algorithms have been created to solve. Next, the theory behind graphs will be briefly addressed in relation to the problem, defining key terms that are used in the subject. Then the algorithms will be explained in more detail and the basics of their principle of work will also be covered. In addition an implementation of the three algorithms will be presented in Java and their running time and also their efficiency will be discussed and compared. In the end we will hopefully arrive at a meaningful answer and find out whether any of these algorithms is superior to the others.

3. The problem

There are many concrete examples with which we can represent the minimum spanning tree problem. Let's take the one that inspired Boruvka to create his algorithm. Suppose you have a country in which there are a number of cities scattered around and the distances between the cities vary. Unfortunately the country does not possess an electric grid, which would enable the citizens to use electric appliances. There are number of possible connections which can be made between the cities, but it is important to keep the cost, in this case proportional to the total length of wire, as low as possible. So the question is how to connect all of the cities with a minimum amount of resources?

4. Definitions and terms

Such problems are primarily solved using graphs. A graph is defined as a set of points, or more formally called vertices, some or all of which are connected with a set of lines, or edges. A graph G is denoted as follows:

$$G = \{V, E\}$$

Where *V* is the set of vertices and *E* is the set of edges. Often vertices are also called nodes. There are many forms of graphs, which may be connected in different ways, but in this case we are interested in a special class of graphs that are called trees. A tree is a connected, simple graph with no circuits or cycles, which means it is possible to traverse from a vertex to any other by a sequence of one or more edges. But it is important to mention that there must be exactly one such path. This is because if there would be multiple paths, then a contradiction would arise, since the definition of a tree states that there must be no cycles.

Another property of a tree is that if it has an order of *n* and by order we mean the total number of vertices, then it must have $n - 1$ edges. It is logical when you think about it, e.g. to connect $n = 3$ vertices once we require $n - 1$ edges, which is two. This is a useful property when solving minimum spanning tree problems. Furthermore, if we were to add an edge to a tree it would cease to be a tree, because this would inevitably create a cycle within the graph. Similarly if we were to remove an edge we would end up with a lone vertex and the tree would cease to be connected.

The graphs used in the problem are undirected graphs, so the edges do not bear a specific direction value and thus the statements going from vertex *u* to vertex *v* and going from *v* to *u* are equally valid. On the other hand, the edges do have another attribute, more specifically their weight, which is central to the problem. The weight attribute is analogous to the distance between the vertices and it is the value that is compared when trying to find the optimal solution.

If we are given a weighted undirected graph and asked to find the minimum spanning tree, then we must include all the vertices that are present in the original graph. This is why the resulting subgraph is called a spanning tree, it "spans" the entire original graph.

5. The algorithms

An algorithm is defined as “a step-by-step procedure for accomplishing some end.”[2] Minimum spanning tree algorithms generally follow a similar pattern, which can be described as follows. First an empty set T is created after which we start adding edges to the set. To be added they must be suitable and fill in some conditions, such as not making the tree cyclic. This is continued until a spanning tree is formed and the loop can be then stopped. All of the three considered algorithms also follow this structure. The difference is how the edges are selected. In addition all of them are greedy algorithms, meaning they choose the best option available at each stage.

In Kruskal’s case we begin by taking the edge with the smallest weight. This will be the first part of the spanning tree. Then we continue by selecting the next smallest edge. If it does not complete a cycle we add it to the set T, otherwise we discard it. We keep selecting new edges this way until we have $n - 1$ successfully chosen edges. As a result we will have a minimum spanning tree.

In Prim’s algorithm we start at any vertex. This works, because in a minimum spanning tree all of the vertices, although not necessarily all the edges, are part of the tree. The aim is to “grow” a tree which is carried out by adding edges to the vertex. We choose the edge with the smallest weight that is not part of T and add it to the set. As more vertices are part of the subgraph, there are also more edges available to be compared and chosen. If there happens to be two edges of the same weight, we choose either one of them. This step is repeated over and over until all of the vertices are part of T.

Boruvka’s algorithm is slightly different in its nature and it is perhaps the most difficult one to visualize. While Kruskal’s and Prim’s essentially work in a step-by-step fashion, in Borukva’s this is not required[3]. The algorithm repeats a series of steps until the minimum spanning tree is found. The steps go as follow: For each vertex, an edge which has the smallest weight is chosen. Depending on the graph, sometimes the first step is enough to connect all edges into a minimum spanning tree, but the more likely outcome is that we end up with a number of trees that are part of the graph, but which are not connected. We then continue by merging these trees with each other into larger trees. This is done by iterating through the edges that connect these components together and choose the ones with the least weight. When this is repeated enough times and all of the components are combined, we have the minimum spanning tree.

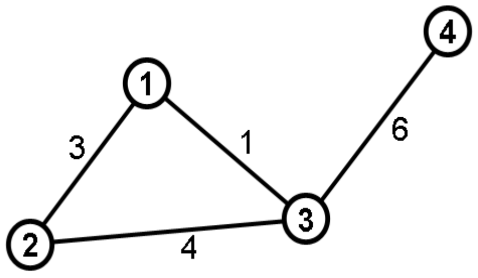
[2] http://www.brpreiss.com/books/opus5/html/book.html

[3] http://www.cs.umu.se/~jopsi/dinf504/chap14.shtml

6. Choosing a graph data structure

In order to solve the problem computationally we require a way to store the graph's information in a data structure which holds the edges and vertices. The two primary methods are the cost adjacency matrix and the adjacency list.[4]

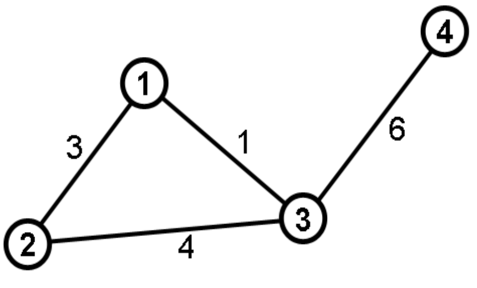
The adjacency matrix is an $n \times n$ square matrix, in which each element represents a connection between two vertices. If no such connection exists the element is left empty. In our case, the graph is undirected, which implies that there is a connection between from vertex u to v and vice versa. This means that the entries a_{ij} and a_{ji} are equal and so the matrix becomes symmetrical. In addition the weights of the connections must be included in the matrix. This is done quite simply by setting the weight as the entry value.



$$\textit{To}$$
$$\textit{From}\quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} \mathrm{x} & \mathbf{3} & \mathbf{1} & \mathrm{x} \\ \mathbf{3} & \mathrm{x} & \mathbf{4} & \mathrm{x} \\ \mathbf{1} & \mathbf{4} & \mathrm{x} & \mathbf{6} \\ \mathrm{x} & \mathrm{x} & \mathbf{6} & \mathrm{x} \end{bmatrix} \end{array}$$

A simple graph with its adjacency matrix

The other data structure for graphs is the adjacency list. It is a collection of all the vertices and each of them has a list of its adjacent vertices. In this case, once again, the weight must also be included into the lists' values.



1 is adjacent to 2(3), 3(1)
2 is adjacent to 1(3), 3(4)
3 is adjacent to 1(1), 2(4), 6(4)
4 is adjacent to 3(6)

[4] http:/ phs/graphs

Same graph and its adjacency list with weights in parentheses

It is clear that the two data structures are rather different in their nature and also perform better at different tasks. In all of the three algorithms there are steps to check if a vertex is connected to another vertex or to obtain one or all of its adjacent vertices. The matrix approach is better at determining whether a connection exists and if so what the weight is. This is because to do so all that is required is to check the correct entry's value. The same task requires a bit more work for the adjacency list especially if the vertex has many edges, because a connection between two vertices is not immediately visible. Instead, one must loop through the list to find the appropriate incident vertex. The adjacency list is superior when it comes to finding adjacent vertices, since that is how the information is essentially stored in the list. The matrix, on the other hand, has to check all of the graph's vertices by going through an entire row. If the graph's order is large then this procedure is costly.

The amount of space that the different representations take up also varies. Even though the graph given in the example happens to store the same amount of variables for both data structures, that is sixteen, this is rarely the case. The main drawback of the adjacency matrix is that graphs most of the time do not use up all of the entries. In the example on the previous page only half of the matrix's entries are used, while the adjacency list does not waste any space. Generally the denser the graph, meaning the more edges it contains, the more appropriate it is to use the adjacency matrix. The list structure has its own weaknesses. All of the edges appear twice, since it is appointed for both vertices at the ends of an edge. The values in the matrix also appear twice but it cannot be said that this is redundant, because they represent the different ways of traversing an edge from one vertex to another. One more drawback of the adjacency list is that the weight value has to be stored separately, whereas in an adjacency matrix the entry itself can be conveniently used to store it.

The more efficient data structure is perhaps the matrix, but I decided to use the adjacency list in my implementation on the basis that it is easier to visualize, when studying the algorithms. It is difficult to predict how the choice affected the speed of the algorithms. It very well may be that one of them gained an advantage over the others, but even if so, that advantage should not be too great to affect the overall order in performance.

7. Implementation details

The program code was written in Java, with an object oriented approach in mind. The graph is initially entered as a text file, in which each line represents an edge, for example "`4 6 10`" is an edge that spans from vertex 4 to 6 and has a weight of 10. After this, the information is processed and added to a Graph object. The Graph holds an array of Vertex objects and each of the vertices contains an expanding array of Edge objects. An edge contains two variables; the other vertex and the weight. There are no empty entries in the Vertex array.

The graph is passed to one of the three algorithms as arguments. The algorithm class then creates a new graph T which contains the minimum spanning tree and it can be output as text. The code is located in the appendix at the end of this essay.

8. A closer look at the algorithms

In this section of the essay the algorithms will be explained in greater detail and in addition the ways of performing the tasks related to graph theory using computer language will be addressed. Also the algorithms' time complexities will be presented.

8.1 Kruskal's algorithm

In Kruskal's algorithm the first thing to do is to sort all the edges into order starting from smallest first. If this wasn't done it would take far too much time to iterate though all of the edges every time. This is the only algorithm out of the three which requires sorting. In this paper, bubble sort is used, which is considered a relatively slow sorting algorithm. This means that the efficiency of this part of the algorithm could be easily improved by applying another sorting algorithm, but in reality its role is not that significant as it does not take up a big fraction of the runtime.

The adjacency list is not well suited for sorting edges and that is why it is easier to create a simpler data structure which is made up of objects that contain all the information, the vertices and the weight, but are not connected to anything and the values are easy to access. That is what the unlinkedEdge class is for.

After the values have been sorted, the actual loop starts and runs *n – 1* times, which is the number of edges in the minimum spanning tree. For each edge we check if its addition to the current situation creates a cycle. Suppose we wish to check if an edge that has vertices *u* and *v* creates a cycle. If there is another way of getting from vertex *u* to vertex *v* by using the already added edges then adding the edge would create a cycle. A simple way to do this is by using the depth first search algorithm[5]. It traverses through all of the vertices in the tree and marks them as it does so, in order to avoid going through the same vertex multiple times. A common way to carry out the depth first search, which is also used in this implementation, is a recursive function. It starts from *u* and traverses all of the edges that are in its tree recursively by calling itself. If it happens to encounter *v* it changes a global boolean, which lets the other parts of the program know that the edge is already connected.

It has been shown that the time complexity of Kruskal's algorithm is O(E log E) or equivalently O(E log V), where E is the number of edges in the graph and V is the number of vertices. E is how many times we loop on the edges and this is multiplied by log V, which is how long it takes to perform the check for cycle task for each edge.[6]

8.2 Prim's algorithm

Prim's algorithm is perhaps the simplest of the three to implement. It also has the least number of code lines. While in Kruskal's algorithm edges are added, in Prim's on the other hand it is the vertices. In this algorithm the useful tool is the array which holds the vertices that have been added. During every iteration all of the vertices that are part of the array are checked and the lightest edge is selected and then added to the array. It is quite clear that after each round the number of vertices to be checked increases and since each vertex has a number of edges, we have a loop within a loop. For the last iteration the function must go through almost the entire graph! This greatly hinders the runtime of the algorithm and it was also visible during experimentation.

The time complexity for Prim's algorithm is $O(V^2)$ and O(E + V logV) depending on data structure.[7]

8.3 Boruvka's algorithm

[5] http://www.nist.gov/dads/HTML/depthfirst.html

[6] http://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[7] http://en.wikipedia.org/wiki/Prim%27s_algorithm

The final algorithm is perhaps the most interesting one. It works in rounds and the number of iterations depends on the structure of the graph. The first step is to select the least weight edge for each vertex. It is possible for two vertices to select the same edge; therefore we must check that the edge is not already in T. Usually the first step will connect most of the edges into a few trees in smaller graphs. The subgraphs are identified by the recursive depth first search function, which was also used in Kruskal's algorithm. It is an efficient method to quickly go through a tree and does not take up a lot of resources.

After this, for each subgraph its smallest weight is selected, ensuring that the edge is not connected to the subgraph itself. It may seem that the algorithm's efficiency would be close to Prim's, because during every round all of the graph's vertices and their edges are inspected causing multiple loops, but this is not actually the case. The truth is that the number of subgraphs in Boruvka's algorithm decreases very rapidly after each iteration and thus only a handful of them are required even for large graphs.

For example, when tested on a randomly generated graph with an order of 100 and containing 100 edges, it took only three rounds to find the minimum spanning tree. After the first round there were 25 subgraphs, after the second it was only 7 and finally it was just 2. Even when the graph was increased to 200 vertices and 450 edges, the number of iterations stayed at three, only the number of subgraphs after each round changed to 58, 15 and 2.

The time complexity of the algorithm is O(E log V), because it can be shown that it takes O(log V) iterations for the outer loop.[8] It has the same time complexity as Kruskal's, so in comparison these two algorithms should be fairly equal.

9. Graphing runtime versus order

After ensuring that the algorithms do indeed give out the correct minimum spanning tree, an experiment was conducted to see the connection between the runtime in milliseconds and the order of a randomly generated graph. This is essentially the time complexity curve. In the experiment a random graph with order varying order was created and the minimum spanning tree was found by using each of the three algorithms. The number of edges was set to be equal to the order of the graph.

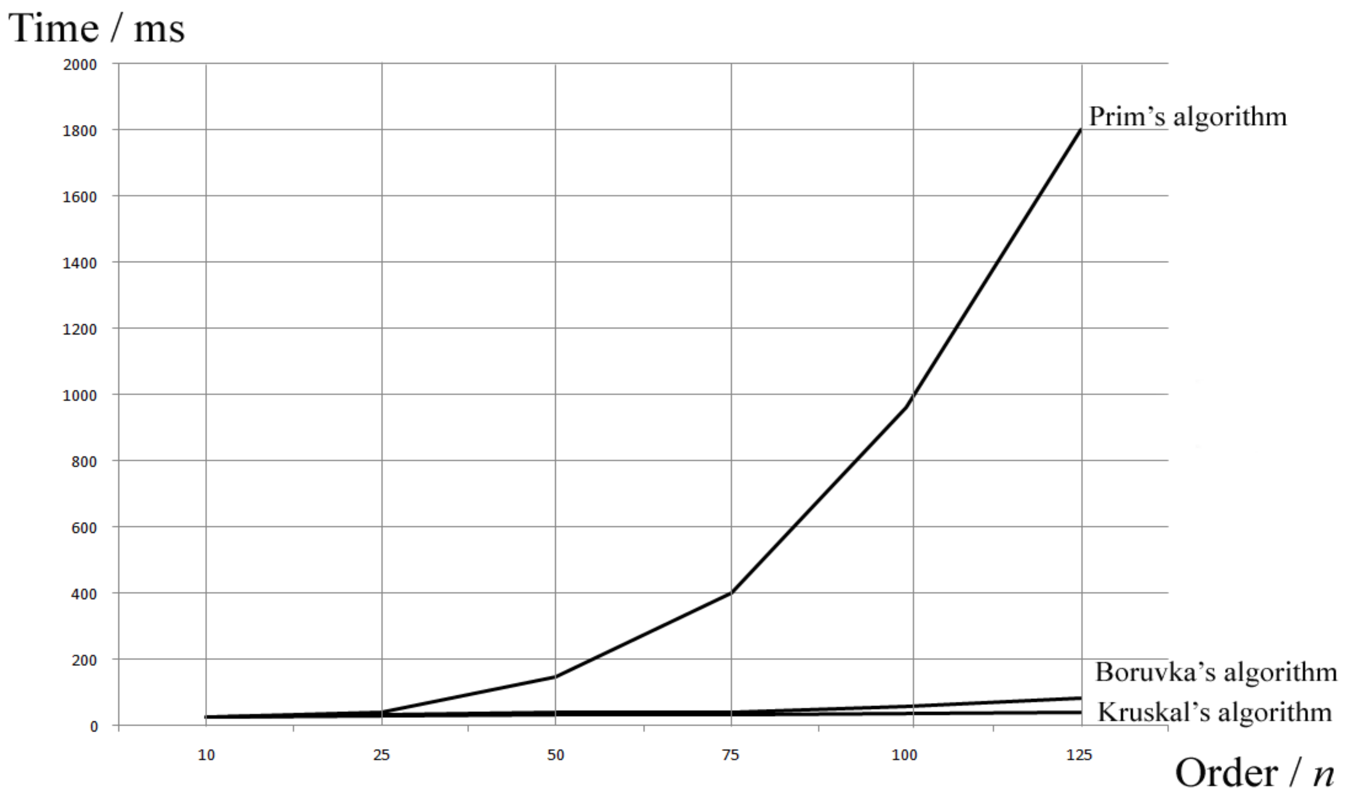
[8] http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Some experimentation was conducted beforehand and it became clear the number of vertices was more important than the number of edges. Doubling the edges increased runtime only by a few milliseconds. It is also evident from the time complexities that V has a greater effect. For each item, the test was run two or three times and the average was taken, although the deviations were nominal. As a result the following data table was produced:

Order *n*	*n* =10	*n* =25	*n* =50	*n* =75	*n* =100	*n* =125
Prim's algorithm	27	41	150	400	960	1800
Boruvka's	27	35	40	42	60	84
Kruskal's	27	30	32	34	36	40

For a graph with an order of 10 the results were almost identical. When the order was increased to 25, slight differences became apparent and it appeared that Kruskal's algorithm was the fastest. For an order of 50, Prim's algorithm took more than five times the time than during an order of 10, while Kruskal's had increased by just five milliseconds. Boruvka's growth is slightly greater than Kruskal's. The next columns proceed in the same fashion and in the final graph test of an order of 125 vertices it took Prim's algorithm nearly two seconds to complete, while Kruskal's algorithm completed the same task in just 40 milliseconds.

If drawn on a graph, with order in the x-axis and also being the independent variable and y-axis as the dependant variable time, the plot would look as follows:The time complexity of Prim's



algorithm clearly appears exponential just as it was given O(V^2). In addition Boruvka's and Kruskal's algorithms had the same time complexity and the graph clearly shows this trend, although Kruskal's gradient is slightly lower. The difference probably lies in the implementation.

10. Conclusion

After examining the three different minimum spanning trees it can be stated that all of them lead to the same results, but use very different approaches. Boruvka's and Kruskal's algorithms are clearly more useful if applied to the real world, while Prim's runtime grows far too quickly with the order of the graph to be of use in a serial processing environment. It is definitely possible to improve on all three implementations, for example by changing the data structure in favour of something more efficient or then decreasing the number of loops.

Moore's law is often misunderstood as the doubling of performance, but in reality it is the amount of transistors on a chip that is doubled. As the upper limits of clock speeds are being reached, hopes are largely placed on multiple processor cores and parallel computing. Parallel computing is in essence the allocation of calculations into smaller parts and running them in different locations simultaneously, such as cores or computers on a network. With that in mind, the study of graph algorithms and the development of new algorithms should also focus on parallel computing.

Out of the three algorithms, Boruvka's holds most promise when parallel computing is considered. It is parallelizable by design and involves searching locally for the smallest edge and then combining the resulting trees after each step. Division of tasks between multiple computer processing nodes would be the logical extension of Boruvka's algorithm. However, as can be seen from this paper, Kruskal's algorithm is much more efficient in a serial environment. Ideally, one would create a hybrid algorithm, combining Kruskal's for local processing and Boruvka for merging subgraphs and parallel processing.

List of sources and references

- Data Structures and Algorithms with Object-Oriented Design Patterns in Java, Bruno R. Preiss, Canada, 1998
- Mathematics for international student, Haese & Harris Publications, Australia, 2005
- http://www.cs.umu.se/~jopsi/dinf504/ - A collection of lecture notes from the course "Algorithms and Data structures"
- http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node5.html - Algorithm analysis course notes
- http://www.nist.gov/ - National Institute of Standards and Technology
- www.wikipedia.org – Online encyclopaedia

The websites were accessed on 20.11.2008. All diagrams and pictures were created by the author of this document.

Appendix - source code

```java
/*
 * Graph.java
 * Igor Podsechin
 * The class which defines and holds a graph
 */

import java.io.*;
import java.util.*;

public class Graph {
    int x = 5000;
    //This array holds the vertices
    Vertex[] vertex = new Vertex[x];
    Vertex[] tree = new Vertex[x];
    //number of vertices
    int n = 0;
    //numer of edges
    int e = 0;

    //Contructor initalizes the vertices
    public Graph() {
        for (int j = 0; j < x; j++) {
            vertex[j] = new Vertex();
        }
    }

    public void addVertices(int v1, int v2, int weight) {
        //Here we assign the edges to vertices
        vertex[v1].addEdge(v2, weight);
        vertex[v2].addEdge(v1, weight);

        //The following lines are for checking for the largest vertice
        //index and assigning it to n.
        if (v1 > n) {
            n = v1;
        }
        if (v2 > n) {
            n = v2;
        }
    }

    //This class is for handling the input file
    public void readInput() {
        //Temporary variables for holding values
        int v1;
        int v2;
        int weight;
        StringTokenizer st;

        //Read the text file line by line
        try {
            BufferedReader in = new BufferedReader(new FileReader("input.txt"));
            String str;
            while ((str = in.readLine()) != null) {
                e++;
                st = new StringTokenizer(str);

                v1 = Integer.parseInt(st.nextToken());
                v2 = Integer.parseInt(st.nextToken());
                weight = Integer.parseInt(st.nextToken());

                addVertices(v1, v2, weight);
            }
            in.close();
        } catch (IOException e) {
        }
    }

    //Outputs the graph as text
    public void printGraph() {
        //Iterate through the vertices
        for (int i = 1; i < n + 1; i++) {
            System.out.println("Vertex " + (i));
            //Iterate through the edges it is connected to
            for (int j = 0; j < vertex[i].getNumberOfEdges(); j++) {
                System.out.println("edge " + vertex[i].getEdge(j).getVertex() + ", w: " +
vertex[i].getEdge(j).getWeight());
            }
        }
    }

    //Returns an array of a non-adjacency list of dges that represents the graph
    public UnlinkedEdge[] getUnlinkedEdgeArray() {
        UnlinkedEdge[] edges = new UnlinkedEdge[e];
        int iterator = 0;
        int v1;
        int v2;
        int weight;
        StringTokenizer st;

        //Read the text file line by line
        try {
            BufferedReader in = new BufferedReader(new FileReader("input.txt"));
            String str;
            while ((str = in.readLine()) != null) {
                st = new StringTokenizer(str);

                v1 = Integer.parseInt(st.nextToken());
                v2 = Integer.parseInt(st.nextToken());
                weight = Integer.parseInt(st.nextToken());

                edges[iterator++] = new UnlinkedEdge(v1, v2, weight);
            }
            in.close();
        } catch (IOException e) {
        }
        return edges;
    }

    public Vertex getVertex(int index) {
        return vertex[index];
    }

    public int getOrder() {
        return n;
    }

    public int getNumOfEdges() {
        return e;
    }
}

/*
 * UnlinkedEdge.java
 * Igor Podsechin
 * This class is needed in Kruskal's and Boruvka's algorithms
 */

public class UnlinkedEdge {

    int v1;
    int v2;
    int w;

    public UnlinkedEdge(int vertex1, int vertex2) {
        v1 = vertex1;
        v2 = vertex2;
    }

    public UnlinkedEdge(int vertex1, int vertex2, int weight) {
        v1 = vertex1;
        v2 = vertex2;
        w = weight;
    }

    public int getWeight() {
        return w;
    }

    public void printEdge() {
        System.out.println("w:" + w + " v:" + v1 + " v:" + v2);
    }

    public int getVertice1() {
        return v1;
    }

    public int getVertice2() {
        return v2;
    }
}
/*
 * Vertex.java
 * Igor Podsechin
 * The class which defines a vertex
 */

import java.util.*;

public class Vertex {

    //Growable array of edges
    Vector<Edge> edges = new Vector<Edge>();

    public void addEdge(int vertex, int weight) {
        edges.add(new Edge(vertex, weight));
    }

    public Edge getEdge(int index) {
        return edges.get(index);
    }

    public Vector printEdges() {
        return edges;
    }

    public int getNumberOfEdges() {
        return edges.size();
    }

    //find a vertices smallest edge by looping through all of them
    //Used in Boruvka's algorithm
    public Edge getSmallestEdge() {
        int smallest = Integer.MAX_VALUE;
        int indexSmallest = -1;

        for (int i = 0; i < edges.size(); i++) {
            if (getEdge(i).getWeight() < smallest) {
                smallest = getEdge(i).getWeight();
                indexSmallest = i;
            }
        }
        return getEdge(indexSmallest);
    }

    //Finds smallest available edge that is not in the argument array, otherwise returns null
    //Used in Prim's and Boruvka's Algorithms
    public Edge getSmallestFreeEdge(int[] arrayOfTakenVertices, int itemsInArray) {
        int smallest = Integer.MAX_VALUE;
        int indexSmallest = -1;
        boolean taken = false;

        //We loop through the edges that the vertex is connected to
        for (int i = 0; i < edges.size(); i++) {
            //Check if the vertex at the end of the edge is part of the argument array
            for (int j = 0; j < itemsInArray; j++) {
                if (arrayOfTakenVertices[j] == getEdge(i).getVertex()) {
                    taken = true;
                }
            }
            //If it is not part of the array and also has the smallest weight then save its index
            if (getEdge(i).getWeight() <= smallest && taken == false) {
                smallest = getEdge(i).getWeight();
                indexSmallest = i;
            }
            taken = false;
        }
        //Return its index, but if no such vertex exists then returns null
        if (indexSmallest != -1) {
            return getEdge(indexSmallest);
        }
        return null;
    }
}



/*
 * Edge.java
 * Igor Podsechin
 * The class which defines and holds an edge
 */
public class Edge {

    int vertex;
    int weight;

    public Edge(int v, int w) {
        vertex = v;
        weight = w;
    }

    public int getWeight() {
        return weight;
    }

    public int getVertex() {
        return vertex;
    }
}


/*
 * KruskalsAlgorithm.java
 * Igor Podsechin
 */

public class KruskalsAlgorithm {

    Graph graph;
    //MST graph
    Graph T;
    //Contains all of g's edges
    UnlinkedEdge[] edges;
    int iterator = 0;
    int verticesAdded = 0;
    boolean connected = false;
    //Array which is used for checking if vertices are connected
    boolean[] visited;

    public KruskalsAlgorithm(Graph g) {
        graph = g;
        T = new Graph();
        edges = g.getUnlinkedEdgeArray();
        sortEdges();

        visited = new boolean[g.getOrder() + 1];

         //Loop through the edges, starting from the smallest one, until there is n-1 of them added
to T
        while (verticesAdded < g.getOrder() - 1) {
            //Clear the visited array
            for (int x = 0; x < visited.length; x++) {
                visited[x] = false;
            }
            //Check if the two vertices are connected, if no add to T
            checkIfConnected(edges[iterator].getVertice1(), edges[iterator].getVertice2());
            if (connected == false) {
                         T.addVertices(edges[iterator].getVertice1(),  edges[iterator].getVertice2(),
edges[iterator].getWeight());
                verticesAdded++;
            }
            connected = false;
            iterator++;
        }
    }

    //Sorts the edges array based on weights using bubblesort
    private void sortEdges() {
        boolean swapped = false;
        UnlinkedEdge temp;
        do {
            swapped = false;
            for (int i = 0; i < edges.length - 1; i++) {
                if (edges[i].getWeight() > edges[i + 1].getWeight()) {
                    temp = edges[i];
                    edges[i] = edges[i + 1];
                    edges[i + 1] = temp;
                    swapped = true;
                }
            }
        } while (swapped);
    }
    //Recursive function which determines if vertices v1 and v2 are connected
    private void checkIfConnected(int v1, int v2) {
        visited[v1] = true;
        if (v1 == v2) {
            connected = true;
        }
        if (T.getVertex(v1).getNumberOfEdges() > 0) {
            //Loop through edges
            for (int i = 0; i < T.getVertex(v1).getNumberOfEdges(); i++) {
                if (visited[T.getVertex(v1).getEdge(i).getVertex()] == false) {
                    checkIfConnected(T.getVertex(v1).getEdge(i).getVertex(), v2);
                }
            }
        }
    }

    public void printMST() {
        T.printGraph();
    }
}
/*
 * PrimsAlgorithm.java
 * Igor Podsechin
 */
public class PrimsAlgorithm {

    Graph T;
    //number of vertices
    int n;
    //This array contains the vertices that have been added
    int[] addedVertexIndex;
    //We use this variable to count how many vertices have been added
    int iterator = 0;

    public PrimsAlgorithm(Graph g) {
        Edge e = new Edge(0, 0);
        int smallestWeight = Integer.MAX_VALUE;
        int indexSmallest = -1;

        //The size of the array is the same as in the order of the original graph
        addedVertexIndex = new int[g.getOrder()];
        //The graph where the MST will be stored
        T = new Graph();

        //Add the first vertex into the array, could be any vertex
        addedVertexIndex[0] = 1;

        //Loop n-1 times (just as many as there are edges in the minimum spanning tree)
        while (iterator++ < addedVertexIndex.length-1) {

            //Loop through the already added vertices and see which one has smallest edge
            for (int j = 0; j < iterator; j++) {
                          e  =  g.getVertex(addedVertexIndex[j]).getSmallestFreeEdge(addedVertexIndex,
iterator);

                //Select the vertex which has the smallest weight
                if(e != null && e.getWeight() <= smallestWeight) {
                    smallestWeight = e.getWeight();
                    indexSmallest = j;
                }
            }
            //Add the edge to T
              e = g.getVertex(addedVertexIndex[indexSmallest]).getSmallestFreeEdge(addedVertexIndex,
iterator);
            T.addVertices(addedVertexIndex[indexSmallest], e.getVertex(), e.getWeight());

            addedVertexIndex[iterator] = e.getVertex();

            indexSmallest = -1;
            smallestWeight = Integer.MAX_VALUE;
        }
    }

    public void printMST() {
        T.printGraph();
    }
}
/*
 * BoruvkasAlgorithm.java
 * Igor Podsechin
 */
import java.util.*;

public class BoruvkasAlgorithm {
    //MST graph
    Graph T;
    int edgeIterator = 0;
    UnlinkedEdge[] edges;
    //Holds the subgraphs that are generated
    Vector[] subgraph;
    int subIterator;
    //Used for deapth first search
    boolean[] visited;
    int visitedIterator = 0;

    public BoruvkasAlgorithm(Graph g) {
        Edge e = new Edge(0, 0);
        T = new Graph();
        edges = new UnlinkedEdge[g.getNumOfEdges()];
        subgraph = new Vector[g.getOrder()];
        visited = new boolean[g.getOrder() + 1];

        //Find smallest edge for each vertice and add it to T
        for (int i = 1; i < g.getOrder() + 1; i++) {
            if (notAlreadyPartOfT(i, g.getVertex(i).getSmallestEdge().getVertex())) {
                                         T.addVertices(i,  g.getVertex(i).getSmallestEdge().getVertex(),
g.getVertex(i).getSmallestEdge().getWeight());
            }
        }

        //The loop which iterates until one subgraph is present
        while (true) {
            subIterator = 0;
            //Sort the vertices into subgraphs,
            for (int j = 1; j < visited.length; j++) {
                if (visited[j] == false) {
                    subgraph[subIterator] = new Vector(1);
                    deapthFirstSearch(j);
                    subIterator++;
                }

            }
            System.out.println(subIterator);
            //End loop when all subgraphs are joined
            if (subIterator == 1) {
                break;
            }

            //Now find the shortest edge for each subgraph, so loop through subgraphs
            for (int x = 0; x < subIterator; x++) {
                int[] subArray = new int[subgraph[x].size()];
                int smallestWeight = Integer.MAX_VALUE;
                int indexSmallest = -1;
```

```
                    //Make an array out of the subgraph vector, since getSmallestFreeEdge() function
accepts only arrays
                for (int i = 0; i < subgraph[x].size(); i++) {
                    Integer temp = (Integer) subgraph[x].elementAt(i);
                    subArray[i] = temp.intValue();
                }

                //loop through edges of subgraph and find least heavy edge that goes out of subgraph
                for (int y = 0; y < subArray.length; y++) {
                    e = g.getVertex(subArray[y]).getSmallestFreeEdge(subArray, subArray.length);
                    if (e != null && e.getWeight() <= smallestWeight) {
                        smallestWeight = e.getWeight();
                        indexSmallest = y;

                    }
                }
                //Add the smallest Edge to T, if it hasn't already been added
                            e  =  g.getVertex(subArray[indexSmallest]).getSmallestFreeEdge(subArray,
subArray.length);
                if (notAlreadyPartOfT(subArray[indexSmallest], e.getVertex())) {
                    T.addVertices(subArray[indexSmallest], e.getVertex(), e.getWeight());
                }

                smallestWeight = Integer.MAX_VALUE;
                indexSmallest = -1;
                resetVisitedArray();
            }
        }
    }

    //Clear the visited array
    private void resetVisitedArray() {
        for (int z = 0; z < visited.length; z++) {
            visited[z] = false;
        }
    }

    private boolean notAlreadyPartOfT(int v1, int v2) {
        for (int x = 0; x < edgeIterator; x++) {
                        if  ((edges[x].getVertice1()  ==  v1  &&  edges[x].getVertice2()  ==  v2)  ||
(edges[x].getVertice1() == v2 && edges[x].getVertice2() == v1)) {
                return false;
            }
        }
        edges[edgeIterator] = new UnlinkedEdge(v1, v2);
        edgeIterator++;
        return true;
    }

    //Recursive function which looks for subgraphs
    private void deapthFirstSearch(int num) {
        subgraph[subIterator].add(new Integer(num));
        visited[num] = true;
        if (T.getVertex(num).getNumberOfEdges() > 0) {
            //Loop through edges
            for (int i = 0; i < T.getVertex(num).getNumberOfEdges(); i++) {
                if (visited[T.getVertex(num).getEdge(i).getVertex()] == false) {
                    deapthFirstSearch(T.getVertex(num).getEdge(i).getVertex());
                }
            }
        }
    }

    public void printMST() {
        T.printGraph() }}
```