Hanna Kumpula, Kari Peisa & Jouko Teeriaho

B

Algorithms for compression and real-time internet services of intelligent road data

LAPLAND UAS PUBLICATIONS
Publication series B. Research reports and
Compliations Publication series 18/2018

Algorithms for compression and real-time internet services of intelligent road data

Hanna Kumpula, Kari Peisa & Jouko Teeriaho

Algorithms for compression and real-time internet services of intelligent road data

Publication series B. Research reports and Compliations 18/2018

Lapland University of Applied Sciences
Rovaniemi 2018

ISBN 978-952-316-258-7 (pdf)
ISSN 2489-2637 (verkkojulkaisu)

Publications of Lapland UAS
Publication series B. Research reports and Compliations Publication series 18/2018

Financiers: Interreg Nord
Authors: Hanna Kumpula, Kari Peisa & Jouko Teeriaho
Layout: Lapland UAS, Communications

Lapland University of Applied Sciences
Jokiväylä 11 C
96300 Rovaniemi, Finland

Tel. + 358 020 798 6000
www.lapinamk.fi/Publications

Table of contents

Introduction

The road weather station (RWS) network in the Northern parts of Finland is too sparse for the purposes of delivering detailed information of the road weather. There may be a hundred kilometers of road between two stations and while these stations deliver information from their locations, the road condition may be completely different half way between them.

The Interreg Nord funded Winter Road Maintenance (WiRMa) project researches the possibility of using large goods vehicles (LGV) as mobile RWSs by equipping them with sensors measuring a number of variables directly on the road. These variables include road condition, water level on road, temperature, and friction data. Figure 1 shows detail of one of the LGVs with the sensors. Furthermore, WiRMa project provides a unique testing platform for different road weather sensors.

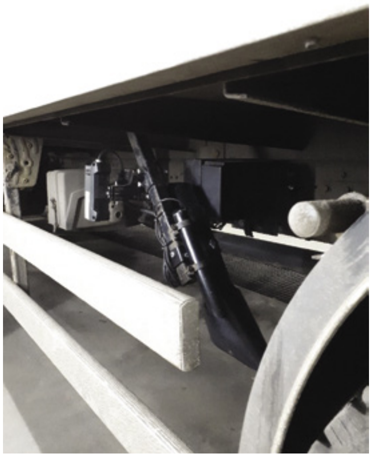

Figure 1 Detail of a vehicle equipped with sensors. (Autioniemi 2018)

The collected sensor data is stored in a cloud storage and delivered to road maintenance personnel through a web based user interface (UI) as well as varied research purposes for road weather predictions through application programming interface (API). Figure 2 shows the main view of the WiRMA UI designed in the project.

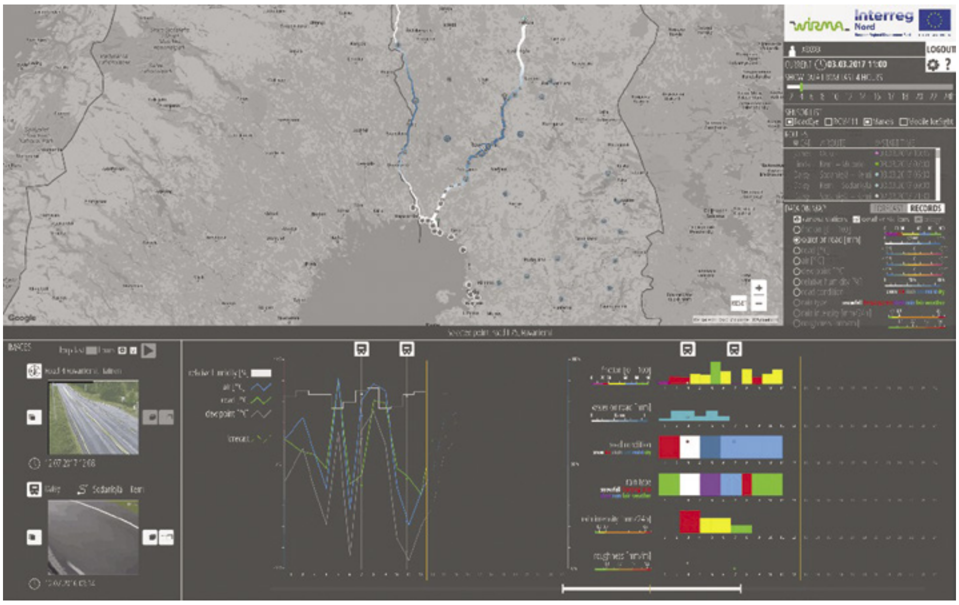
Figure 2 Main view of the WiRMa UI designed in the project.

This article describes the technical points in compressing the collected measurement data. Both the currently implemented algorithm and thoughts on future development are shared.

Chapter 2 shortly explains the required vector mathematics used in these algorithms.

Chapter 3 describes the algorithm developed by Jouko Teeriaho (Lapland UAS) currently implemented in WiRMa project. It explains the mathematics used in the algorithm, pseudo code of the algorithm, and explanation of variables. Implementation using Mathematica and Python as well as the visual explanations are included.

Chapter 4 describes the work of Kari Peisa (Lapland UAS). It concentrates on future and discusses how the algorithm could be improved and taken toward real-time tracking application mainly through the use of R-trees. Implementation of the test runs made using Python are included as well.

1 Vector mathematics

Algorithms are based on vector mathematics. Position on the road is presented as a point (x,y) in the data, where x and y refer to coordinates along the x- and y-axis. Mathematically these points can be called position vectors **r**=(x,y).

For two position vectors a=(a_x, a_y) and **b**=(b**x**, b_y), the operations are

Addition

$$\boldsymbol{a}+\boldsymbol{b}=(a_x+b_x, a_y+b_y) \qquad (1)$$

Multiplication by a constant

$$t\boldsymbol{a}=\left(ta_x, ta_y\right) \qquad (2)$$

Subtraction

$$\boldsymbol{a}-\boldsymbol{b}=\left(a_x-b_x, a_y-b_y\right) \qquad (3)$$

Dot product

$$\boldsymbol{a}\cdot\boldsymbol{b}=a_xb_x+a_yb_y=|\boldsymbol{a}|\left|\boldsymbol{b}\right|\cos\gamma\ , \qquad (4)$$

where g is the angle between the vectors.
Length of a vector

$$|\boldsymbol{a}|=\sqrt{a_x^2+a_y^2} \qquad (5)$$

Angle between two vectors in radians

$$\gamma=\arccos\left(\frac{\boldsymbol{a}\cdot\boldsymbol{b}}{|\boldsymbol{a}|\left|\boldsymbol{b}\right|}\right) \qquad (6)$$

Arc length of a circle

$$s=\gamma r, \qquad (7)$$

where g is the angle in radians and *r* is the circle radius.
Vector projection **a** onto **b**

$$\boldsymbol{a}_{\boldsymbol{b}}=\frac{\boldsymbol{a}\cdot\boldsymbol{b}}{\boldsymbol{b}\cdot\boldsymbol{b}}\boldsymbol{b}=\frac{\boldsymbol{a}\cdot\boldsymbol{b}}{\left|\boldsymbol{b}\right|}\frac{\boldsymbol{b}}{\left|\boldsymbol{b}\right|} \qquad (8)$$

2 Implemented compression algorithm

The LGV sent data must be compressed in such a way that following requirements are fulfilled:

- Geometry of the road is preserved
- Points where the condition parameter changes should be included in the compressed file.

The algorithm goes through the road points in order. First two points determine the road direction and are added to the compressed points. Then points are inspected one by one and added to the compression if either the color changes or lateral deviation from the inspected point is too large from the established road direction. Every time a point is added to the compression, the road direction is determined again.

For two road points a and b, the connection vector **b**-**a** forms the road direction line. The lateral deviation of point c under inspection from the road direction line is calculated using equations (5) to (7)

$$s = |c-a| \arccos\left(\frac{(b-a)\cdot(c-a)}{|b-a||c-a|}\right) \tag{9}$$

2.1 DESCRIPTION OF THE COMPRESSION ALGORITHM

First, the algorithm in explained in pseudocode and then using images.

2.1.1 Written description of the algorithm

```
Variables
    data : [[$x_1$,$y_1$],[$x_2$,$y_2$], … ,[$x_n$,$y_n$]] # points of the road data
    colors : [$color_1$, $color_2$, … ,$color_n$] #colors expressing condition parameter
    c_list : [[$x_1$,$y_1$],[$x_2$,$y_2$], … ] # compressed list of points
    pcolors : [$color_1$, $color_2$, … ] #colors of points for c_list
```

Variables a and b are used to determine the direction of the road, the third pointer c moves forward in array data

a, b, c : pointers of type [a1,a2], [b1,b2], and [c1,c2], respectively

Constants

MAXDEV : parameter giving the maximal lateral deviation of point c from the straight line determined by a and b. If the scale of the map is changed, then MAXDEV should be changed proportional to the scale.

```
Initialisation:
Pointers a and b are set to point to the first points

# Initialisation:
# Pointers a and b are set to point to the first two
# points of the road data
a = array(data[0])
b = array(data[1])

# First two points are added to the c_list and the colors
# to c_colors
c_list = [data[0], data[1]]
pcolors = [colors[0], colors[1]]

# Set index for the while structure
k = 2

# while structure
while (k < len(data)):
    c = array(data[k])
    angle = angleBetween(c-a, b-a)
    s = norm(c-a)*angle

    if(color[k] != color[k-1]):
         if(data[k-1] != c_list(len(c_list)-1):
              c_list.append(data[k-1])
              pcolors.append(colors[k-1])
         c_list.append(data[k])
         pcolors.append(colors[k])
         a = data[k-1]
         b = data[k]
    else:
         if(s > MAXDEV):
              c_list.append(data[k])
              pcolors.append(append(colors[k])
              a = b
              b = c
    k = k+1
```

2.1.2 Visual description of the algorithm

Initialisation:

Pointers a and b are set to point to the first points

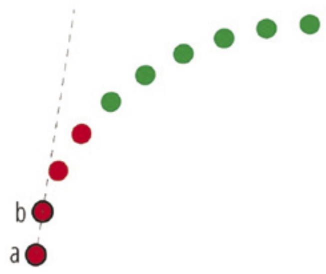

Figure 3 Initialisation phase: set pointers a and b.

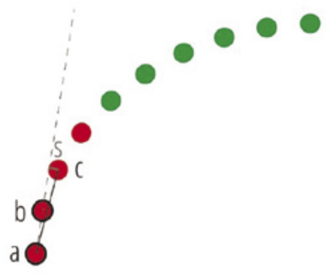

Figure 4 Set pointer c and calculate s. Check if conditions.

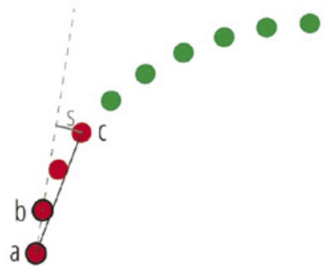

Figure 5 Set pointer c and calculate s. Check if conditions.

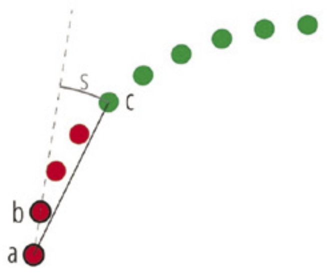

Figure 6 Set pointer c and calculate s. Check if conditions. First if condition catches color change.

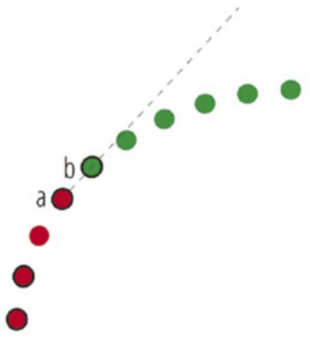

Figure 7 Reposition pointers a and b.

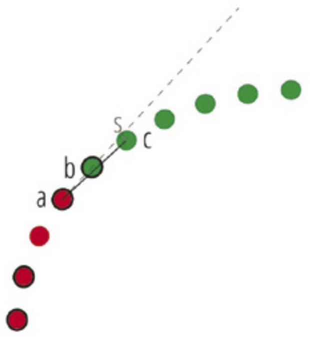

Figure 8 Set pointer c and calculate s. Check if conditions.

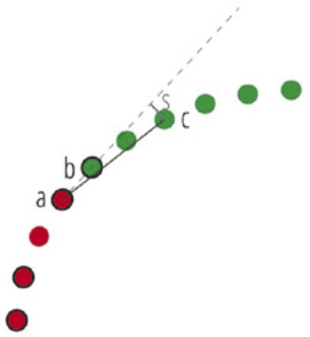

Figure 9 Set pointer c and calculate s. Check if conditions.

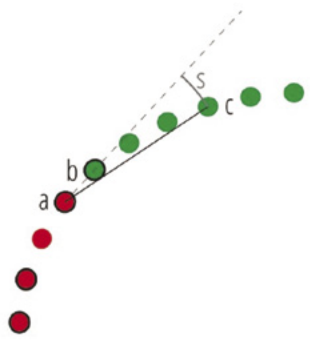

Figure 10 Set pointer c and calculate s. Check if conditions.

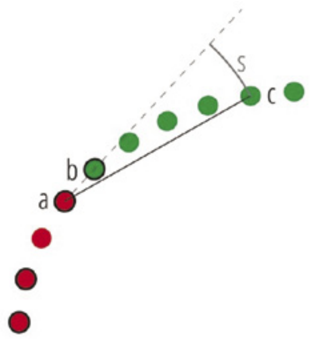

Figure 11 Set pointer c and calculate s. Check if conditions.

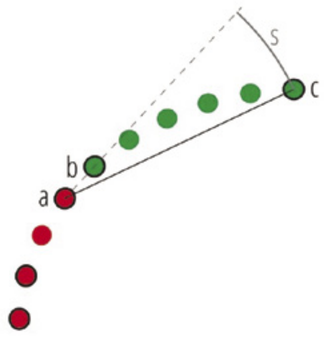

Figure 12 Set pointer c and calculate s. Check if conditions. Second if condition catches s>MAXDEV

2.2 IMPLEMENTATION OF THE ALGORITHM

Implementation of the algorithm in Mathematica and Python is described in detail. Both of the algorithms use the same dataset. Figure 13 shows a scatter plot of 191 points of example road data. The data table is available in Appendix A.

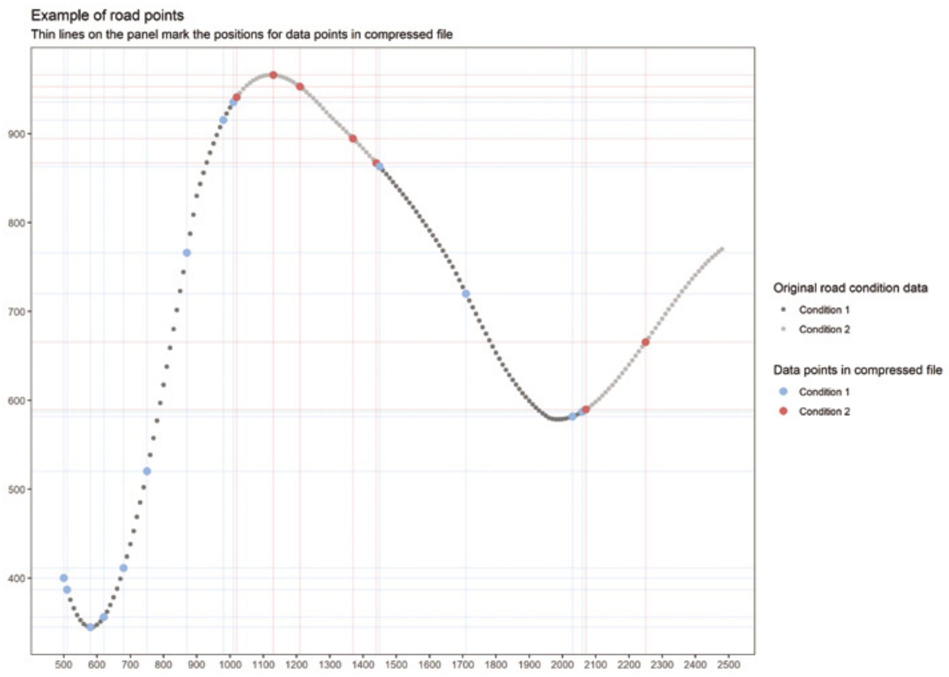

Figure 13 191 points of example road data.

2.2.1 Mathematica implementation

Mathematica has a function called VectorAngle to calculate the angle between two vectors and Norm to calculate the length.

Initialisation of variables.

While

```
maxdev = 30.0;
a = data[[1]];
b = data[[2]];
c_lista = {a, b};
c_colors = {colors[[1]], colors[[2]]};
While -loop
k = 3;
While[k < 191,
     c = data[[k]];
     a = VectorAngle[c-a, b-a];
     d = a*Norm[c-a];

     If[colors[[k-1]]≠colors[[k]],
           If[c_list[[Length[c_list]]]≠data[[k-1]],
                 c_list = Append[c_list, data[[k-1]]];
```

```
                c_colors = Append[c_colors, colors[[k-1]]];];
            c_list = Append[c_list, data[[k]]];
            c_colors = Append[c_colors, colors[[k]];
            a = data[[k-1]];
            b = data[[k]],
            If[d>maxdev, c_list = Append[c_list, c];
                c_colors = Append[c_colors, colors[[k]];
                a = b;
                b = c;];
            ];
        k++;
    ]
```

Compressed list has following points:

```
c_list
{{500., 400.}, {510., 386.721}, {580., 344.659}, {620., 355.779}, {680., 411.073}, {750., 520.}, {870., 765.9}, {980., 915.25}, {1010., 935.548}, {1020., 941.05}, {1130., 966.065}, {1210., 952.912}, {1370., 894.583}, {1440., 867.047}, {1450., 862.886}, {1710., 719.846}, {2030., 581.462}, {2060., 587.009}, {2070., 589.423}, {2250., 665.462}
```

2.2.2 Python implementation

Python's basic version has no vector class. By loading numpy –packages the angle between two vectors can be calculated:

```
from numpy import (array, dot, arccos, subtract)
from numpy.linalg import norm
a = array([4, 2])
b = array([1,6])
# dot(a,b) calculates dot product
# norm(a) calculates the length of vector a
angle = arccos(dot(a,b)/norm(a)/norm(b))
```

```
from numpy import (array, dot, arccos, subtract)
from numpy.linalg import norm

#Move test data points into variables data and colors
```

```
  data =[[500., 400.], [510., 386.721], [520., 375.37], [530.,365.891],
[540., 358.229], [550., 352.33], [560., 348.138], [570.,345.6], [580.,
344.659], [590., 345.261], [600., 347.352], [610.,350.877], [620.,
355.779], [630., 362.006], [640., 369.501], [650.,378.21], [660., 388.079],
[670., 399.051], [680., 411.073], [690.,424.089], [700., 438.045], [710.,
452.886], [720., 468.557], [730.,485.003], [740., 502.169], [750., 520.],
[760., 538.442], [770.,557.439], [780., 576.937], [790., 596.881], [800.,
617.216], [810.,637.887], [820., 658.839], [830., 680.018], [840.,
701.368], [850.,722.835], [860., 744.364], [870., 765.9], [880., 787.388],
[890.,808.773], [900., 830.], [910., 843.352], [920., 855.895], [930.,
867.65], [940., 878.635], [950., 888.869], [960., 898.37], [970.,907.157],
[980., 915.25], [990., 922.667], [1000., 929.427], [1010.,935.548],
[1020.,941.05], [1030., 945.951], [1040., 950.27], [1050., 954.026],
[1060.,957.238], [1070.,959.925], [1080., 962.105], [1090., 963.797],
[1100.,965.021], [1110., 965.794], [1120., 966.136], [1130.,966.065],
[1140.,965.601], [1150., 964.762], [1160.,963.567], [1170., 962.035],
[1180.,960.185], [1190.,958.035], [1200., 955.604], [1210., 952.912],
[1220.,949.976], [1230., 946.816], [1240., 943.451], [1250.,939.899],
[1260.,936.18], [1270., 932.311], [1280.,928.312], [1290., 924.202],
[1300.,920.], [1310., 916.463], [1320.,912.899], [1330., 909.306],
[1340.,905.681], [1350.,902.02], [1360., 898.322], [1370., 894.583],
[1380.,890.801], [1390.,886.973], [1400., 883.096], [1410.,879.168],
[1420.,875.186], [1430.,871.146], [1440., 867.047], [1450., 862.886],
[1460.,858.659], [1470.,854.365], [1480., 850.], [1490., 845.562], [1500.,
841.047], [1510.,836.454], [1520., 831.779], [1530., 827.019], [1540.,
822.173], [1550.,817.237], [1560., 812.208], [1570.,807.084], [1580.,
801.862], [1590.,796.539], [1600., 791.113], [1610., 785.58], [1620.,
779.939], [1630.,774.185], [1640., 768.318], [1650., 762.333],
[1660.,756.228], [1670.,750.], [1680., 742.551], [1690., 735.028], [1700.,
727.453], [1710., 719.846], [1720.,712.228], [1730.,704.62], [1740.,
697.042], [1750., 689.515], [1760., 682.06], [1770.,674.697], [1780.,
667.448], [1790., 660.333], [1800., 653.373], [1810., 646.588], [1820.,640.],
[1830., 634.086], [1840., 628.392], [1850., 622.926], [1860.,617.695],
[1870.,612.707], [1880., 607.97], [1890., 603.491], [1900., 599.278],
[1910., 595.338], [1920.,591.678], [1930., 588.307], [1940., 585.232],
[1950., 582.461], [1960., 580.], [1970.,579.112], [1980., 578.611],
[1990.,578.483], [2000., 578.717], [2010., 579.3], [2020.,580.219],
[2030., 581.462], [2040., 583.016], [2050., 584.869], [2060., 587.009],
[2070.,589.423], [2080., 592.098], [2090., 595.022], [2100., 598.182],
[2110., 601.567], [2120.,605.163], [2130., 608.957], [2140., 612.938],
[2150., 617.094], [2160., 621.41], [2170.,625.876], [2180., 630.478],
[2190., 635.204], [2200., 640.041], [2210., 644.977], [2220.,650.],
[2230., 655.097], [2240.,660.255], [2250., 665.462], [2260., 670.706],
[2270.,675.973], [2280., 681.253], [2290., 686.531], [2300., 691.795],
[2310., 697.034], [2320.,702.234], [2330., 707.383], [2340., 712.469],
[2350., 717.478], [2360., 722.399], [2370.,727.219], [2380., 731.926],
[2390.,736.506], [2400., 740.948], [2410., 745.24], [2420.,749.367], [2430.,
753.319], [2440., 757.082], [2450., 760.644], [2460., 763.993], [2470.,
767.116], [2480., 770.]]

  colors=[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

  # Initialize vectors a and b
  a = array(data[0])
  b = array(data[1])

  # Move first two points to vectors containing compressed
data points
  c_list = [data[0],data[1]]
  c_colors = [colors[0],colors[1]]

  # Define maximum deviation for the compression
  MAXDEV=30
  # Starting position for the index
  k = 2;

  # While loop for selecting data points to the compressed
data
  while (k<len(data)):
       #move next data point for comparison
       c = array(data[k])
       #difference vector c-a
       d1 = subtract(c,a)
       #difference vector c-b
       d2 = subtract(b,a)
       #calculate angle between difference vectors
       angle = arccos(dot(d1,d2)/norm(d1)/norm(d2))
       #calculate deviation of point c from the line ab
       d = angle*norm(d1)
```

```
      if (colors[k-1] != colors[k]): #if color has changed
            if (data[k-1] != c_list[len(c_list)-1]):
                  c_list.append(data[k-1])
                  c_colors.append(colors[k-1])
            #add point c and the previous point to the
compressed data
            c_list.append(data[k])
            c_colors.append(colors[k])
            #reposition vectors a and b
            a = data[k-1]
            b = data[k]
      else:
            if(d>MAXDEV): #if deviation exceeds MAXDEV
                  #add point c in the compressed data
                  c_list.append(data[k])
                  c_colors.append(colors[k])
                  #reposition vectors a and b
                  a = b
                  b = c
      k = k+1 #add index for the while loop

  #print the points, colors, and the number of the points
in the compressed data

  print("Results of the test run: ")
  print("Compressed data points: ",c_list)
  print("\nColors of the compressed data points: ",c_colors)
  print("\nCompressed data contains ",len(c_list)," points")
```

3 Towards real-time tracking of spatial road data

The core of the system's operation is a fixed indexed mesh structure, by which the roads in different countries subject to measurement will be framed. The mesh enables the system to locate and organise the road information that is coming from the sensors. The data consists of road surface condition and weather-related parameters which are stored in a cloud storage as JSON (JavaScript Object Notation) data format. Furthermore, the mesh also makes it possible to present real-time knowledge of road condition to other vehicles travelling on the roads.

The mesh is stored in an R-tree for fast spatial queries. R-trees are powerful tree form data structures for efficient access methods of spatial data. The mesh is made of indexed quadrilaterals, i.e., each quadrilateral has a unique index that includes the identification of the country, the road, and the order number of the quadrilateral in the road. The mesh is an extension to the idea to divide road into linear segments.

3.1 CONSTRUCTING THE MESH

The mesh construction requires the road being represented as a polyline. The interconnected quadrilaterals forming the mesh are constructed around the linear segments of the polyline. Figure 14 shows part of the road data with the mesh.

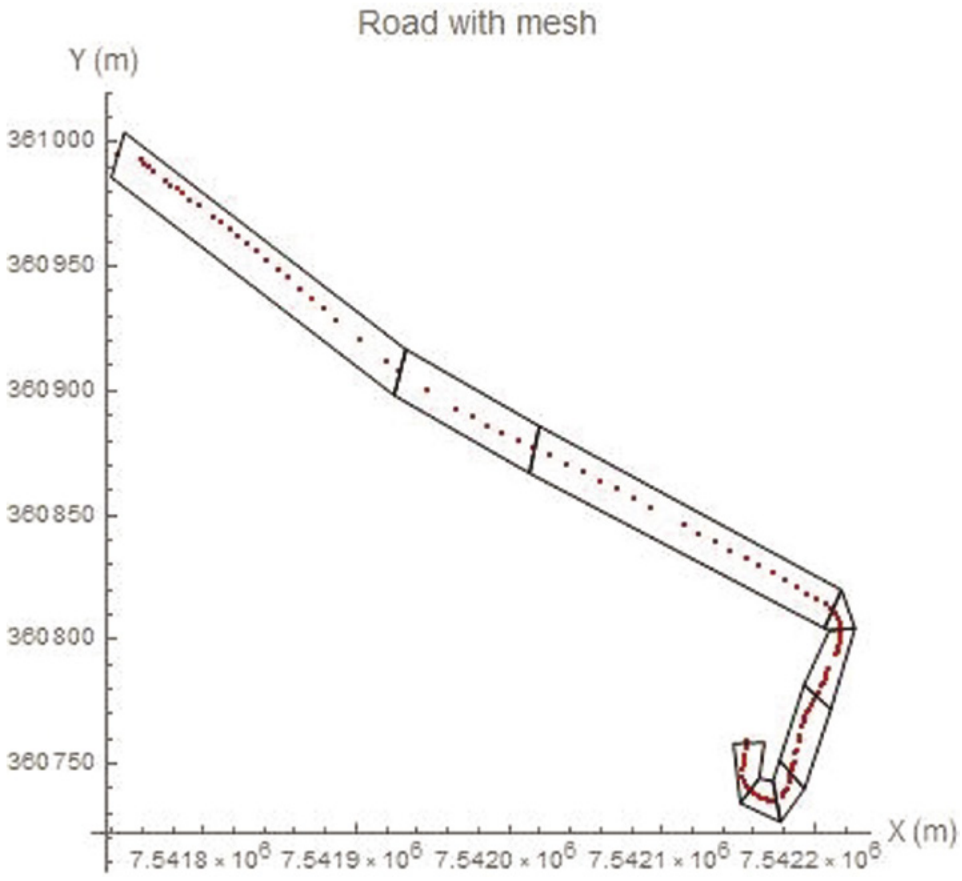

Figure 14 Part of the road data along with the mesh. Mesh width is around 20 m and the maximum deviation of a point inside a linear segment is 5 m.

First, the length of the projection vector can be calculated using equation (8). If vector **b** in said equation is a unit vector $\hat{\mathbf{b}}$ then the length of the projection vector can be obtained efficiently simply using

$$\left|\boldsymbol{a} \cdot \hat{\boldsymbol{b}}\right| \tag{10}$$

Second, as the dot product of two perpendicular vectors is zero, perpendicular unit vectors $\hat{\boldsymbol{n}}$ and $-\hat{n}$ to some unit vector $\hat{\boldsymbol{s}} = (\boldsymbol{s}_x, \boldsymbol{s})$ can be formed as

$$\hat{n} = (-s, s_y) \; and - \hat{n} = (s_x, -s_y) \tag{11}$$

Figure 15 shows the mesh elements that form the quadrilaterals around the road polyline. Road points are handled as position vectors in calculations.

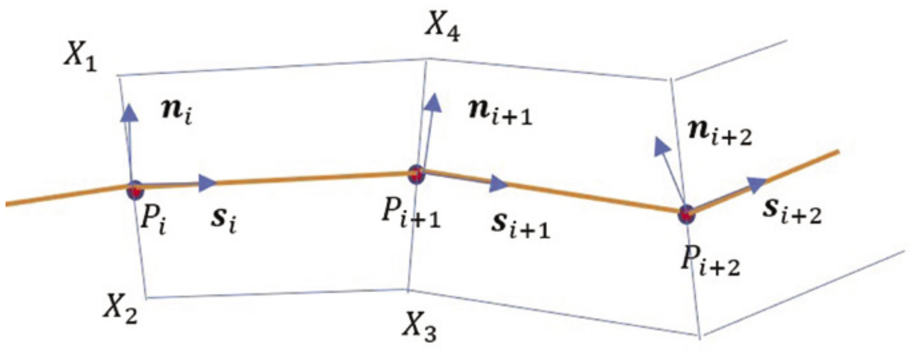

Figure 15 Mesh elements: quadrilateral corners (X_1,...,X_4), unit vectors for road direction ($\hat{s}_i$) and their perpendicular unit vectors ($\hat{n}_i$) in each road point (P_i).

The mesh is constructed from the collected LGV data. First, the polyline for the road is determined by going through the data points, which must be in the correct order and must not contain any discontinuities. Second, the quadrilaterals are formed around the polyline segments.

For the polyline, the first point is added to the polyline and used as a reference point. The direction unit vector of the linear segment as well as the normal to it are calculated using the next point in the data. The algorithm goes through all the successive data points, constructs a trial linear segment, and projects it on the previously calculated normal. The length of this projection is then compared to the maximum deviation. If the maximum deviation is exceeded, then the previous data point is added to the polyline and used as starting point for the next linear segment.

For the quadrilaterals, the polyline points are traversed. At the first and last points the quadrilateral side is set parallel to the normal, but for all other points quadrilateral sides are placed such that they act as the bisector for the two neighboring polyline segments. The polyline goes through the middle of the mesh quadrilaterals.

3.1.1 Algorithm for the mesh construction

Algorithm for constructing the polyline.

```
”””  denote
   xyList  #list of successive points of the road
   polyline #list of vertices in polyline of the road
   nextP++ #nextP is the next point in a point list
   nextP-- #nextP is the previous point in a point list
”””
  startP = xyList[0]
  polyLine=[startP]
  nextP = xyList[1]
```

```
   while nextP in xyList
     s0 = normalize(nextP-startP) # make a unit vector
     n0 = [-s0[1], s0[0]]
     nextP++
     while nextP in xyList and
            dotProduct(nextP-startP,n0) <= maxDeviation
       nextP++
     nextP--
     if nextP in xyList append nextP to polyLine
       startP = nextP
       nextP++
   return polyLine
```

Algorithm for constructing the quadrilaterals.

```
  ””” denote
   segment points = [P0, P1, P2,…,Pn-1] is a list of consecutive
points that form the polyline of the road.
  ”””
  at the beginning i = 0
    s0 = normalize(P1-P0) # make a unit vector
    n0 = [-s0[1], s0[0]]
    X1 = P0 + h* n0
    X2 = P0 - h* n0
  repeat on the following segments i = 1,2,…,n-2
    s1 = normalize(Pi+1-Pi)   # make a unit vector
    n1 = [-s1[1], s1[0]]
    X3 = Pi+1 - ½*h*(n0+ n1) # - for counter clockwise order
    X4 = Pi+1 + ½*h*(n0+ n1) # + for counter clockwise order
    Append [X1, X2, X3, X4] to the mesh
    X1 = X4
    X2 = X3
    n0 = n1
  at the end
    X3 = Pn-1 - h*n1
    X4 = Pn-1 + h*n1
    Append [X1, X2, X3, X4] to the mesh
  return mesh
```

The algorithms described above only work with data that contains data points that form a clean road and do not contain any discontinuities or return trips. If the original data does include anything like these, then points that fulfill the requirements must be extracted for the algorithms.

Furthermore, the algorithms in their current form do not work with data having geodetic (latitude and longitude) coordinates in degrees. Instead, the coordinates must be presented in Cartesian coordinates.

During the development process these challenges were taken care of by, firstly, ensuring that the data set only contained properly successive points. Secondly, the coordinates were converted from geodetic into ETRS-TM35FIN Cartesian coordinates by copying the code from a PHP library (Loukko.net 2013) into the Python program. The conversion was reviewed through the transformation service of the Finnish Geodetic Institute (n.d.).

3.2 TESTING IF THE POINT IS INSIDE A POLYGON

Algorithm for testing whether a point is inside a polygon is based on the general crossing number algorithm also known as even-odd rule algorithm. The implementation of the algorithm described here and presented in section 4.2.1 is based on the work described by Paul Bourke (1987). The underlying concept of the algorithm is to calculate the number of intersections the horizontal line drawn from the query point makes with the polygon edges. In the case of query point being inside the polygon, the number shall be even and odd otherwise. This property is true for both convex and concave polygons. The polygon edges may be traversed in either clockwise or counter-clockwise.

The algorithm goes through the polygon edge-by-edge and checks the parity of the number of intersections from the query point. After a number of checks regarding the location of the query point to the polygon edge, the x-coordinate of the intersection is determined by solving

$$x_{inters} = \frac{(y - p_{1y})(p_{2x} - p_{1x})}{(p_{2y} - p_{1y})} + p_{1x}$$

where is the y-coordinate of the query point, (*P*1x, *P*1y) and (P2x, P2y) are the first and second end point of the polygon edge under inspection, respectively. Figure 16 illustrates the algorithm checking the query point against one of the polygon edges traversed in clockwise direction.

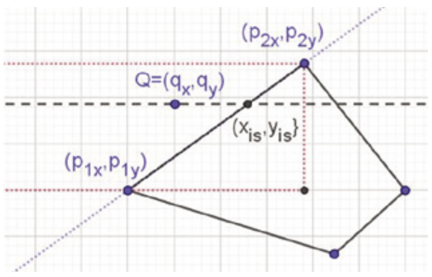

Figure 16 Illustration of the algorithm determining if the queried point Q is inside the polygon.

The crossing number algorithm is efficient and is running in linear time. The implementation of the algorithm falsely categorises some points that lie on an edge or a vertex, therefore, these cases should be checked in a separate process. However, for the purpose of tracking locations with the mesh, cases where the query point hits an edge or a vertex are extremely rare and therefore can be left untracked.

3.2.1 Algorithm for determining if the point is inside the polygon

A Python implementation of the algorithm.

```
def isInsidePolygon(x, y, points):
 n = len(points)
 inside = False
 p1x, p1y = points[0]
 for i in range(1, n + 1):
   p2x, p2y = points[i % n]
   if y > min(p1y, p2y):
     if y <= max(p1y, p2y):
       if x <= max(p1x, p2x):
         if p1y != p2y:
          xinters = (y - p1y)*(p2x - p1x)/(p2y - p1y) + p1x
           if p1x == p2x or x <= xinters:
             inside = not inside
   p1x, p1y = p2x, p2y
  return inside
```

3.3 SYSTEM

The proposed system is based on data in a cloud storage, which is continually updated via mobile sensors.

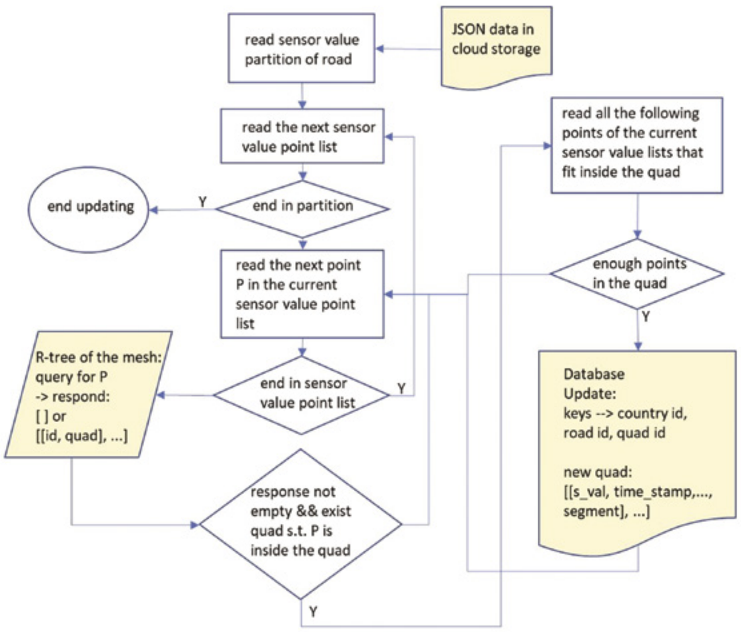
read sensor value partition of road
JSON data in cloud storage
read the next sensor value point list
read all the following points of the current sensor value lists that fit inside the quad
end updating
Y
end in partition
enough points in the quad
read the next point P in the current sensor value point list
Y
R-tree of the mesh: query for P -> respond: [] or [[id, quad], ...]
end in sensor value point list
Y
Database Update: keys --> country id, road id, quad id
new quad: [[s_val, time_stamp,..., segment], ...]
response not empty && exist quad s.t. P is inside the quad
Y

Figure 17 Proposed procedure for real-time tracking of spatial road information.

Figure 17 shows a flow chart describing the proposed procedure for a real-time internet application updating road information.

3.3.1 The usage of R-tree in tracking location from the mesh

R-trees are dynamic tree form data structures, where the indexing is based on multidimensional information, such as spatial data. In dynamic tree form global reorganisation is not required to handle insertions or deletions. The R-tree concept was created by Toni Guttman (1984). After that many variants such as R*-tree or R+-tree have been developed. Here, the use of R-tree is restricted only for the use of 2D geospatial data. Therefore, the indexing of R-tree is based on rectangles (x_{min}, y_{min}, x_{max}, y_{max}) which are inserted into the tree.

When an R-tree is created and new elements are inserted in it, R-tree groups nearby objects and represents them with their minimum bounding rectangle (MBR) in the next higher level of the tree. The inner nodes of an R-tree are called pages, and they have knowledge of the MBR and the connections to the next lower level pages. The MBRs of a higher level include all the MBRs in its subtree. In the lowest level the leaf nodes have three entities: the unique index number in the tree, the MBR, and the data object to be stored. Here, the data objects are the quadrilaterals of the mesh.

When a query including a 2D point or a polygon is made to R-tree, the tree is traversed starting from the root node to find which MBRs intersect, i.e., include, the point or the polygon. Depending on how many intersections are found, a list of hits of leave nodes is returned.

There are two kinds of requests in the WiRMa road service application. First, from the coordinates of the tracked vehicle, the quadrilateral the vehicle is located in needs to be determined, possibly along with the next couple of nearest quadrilaterals as well. Second, real-time sensor value information regarding the segments or the whole road needs to be acquired.

Spatial requests are typical tasks that are handled by the R-tree. Storing all the sensor value data in the R-tree results in multiple queries and updates, which might cause R-tree performance to collapse. Therefore, only the mesh of each road is constructed and inserted into the R-tree and considered fixed. Then, the R-tree is mainly subject to the queries and its performance remains effective. The sensor value data is to be stored in a separate database, where the keys for accessing information consist of the identification of the country, the road, and the mesh quadrilateral. This data is stored in the R-tree as well and returned in the response.

The identification of the country, the road, and the mesh quadrilateral can be combined into one unique index number of an R-tree element. It can be constructed, e.g., by concatenating strings. Each country has a unique identifying number, each road within a country has unique identifying number, and each mesh quadrilateral has order number from 1 to the number of quadrilaterals in the road. These identifying numbers are converted to a certain length of strings and then concatenated. The length of strings can be, e.g., the following:

2 digits for country, max 99 different countries

3 digits for roads in each country, max 999 different roads /country

7 digits for quadrilaterals, max 9 999 999 different segments /road

The index string length is 12, thus including up to 999 999 999 999 possible different index numbers. When necessary, the number is padded with zeros from the beginning to reach the desired length.

When the quadrilateral is inserted into the R-tree, the concatenated index string is converted to long int which is the unique index number of the inserted R-tree element. Here is an example of indexing:

If country unique number = 3, road number within the country = 4, and the order number of the mesh quadrilateral =11, then

3 ->'03'

4 -> '004'

11 -> '0000011'

concatenated bit string is converted to a long integer

'030040000011' -> 30040000011

The long integer index number in R-tree respond can be parsed back to identify the country, road, and the mesh quadrilateral.

3004000001 -> '03004000001'-> ['03','004','0000011']

3.3.2 The details of updating sensor value information

The unit module of the information of the road that is going to be extracted from the JSON data in the cloud storage is a list of road points, which are connected to a certain normalised integer value of the selected sensor type. When the sensor value changes, a new point list will be created. Sensor types are the road features that are presented in road services. There are five sensor types currently in use: water level on the road, road temperature, air temperature, road condition, and friction. The road information of each sensor type can be extracted from JSON file into the following (Python) list format:

```
[[sensor_value, spatial point list], …]
```

In real data, the spatial point lists of the sensor values, or the points in one list, do not necessary form a continuous list of successive points of the road. The vehicle may have stopped for a moment, it may have deviated from its route, it may have made back and forth moving, or there may be other discontinuities. When updating the road information, we need to be sure that JSON data has enough new information of the current section of the road. This can be ensured by using the same technology used in the R-tree, i.e., the MBR.

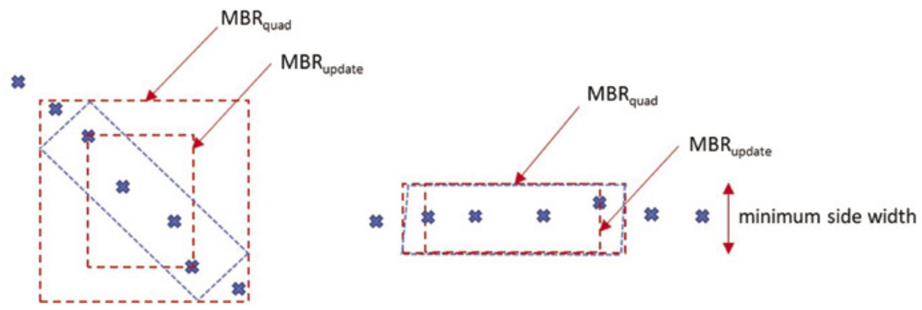

Figure 18 MBR for new data in mesh that is going to be updated.

The road information in the current quadrilateral will be updated in the database if condition

$$\frac{|MBR_{update}|}{|MBR_{quad}|} \geq b,$$

where b is some predefined boundary level and $|MBR|$ the area of the rectangle, is fulfilled. Figure 18 illustrates the setting of a minimum width to the bounding rectangle in order to avoid $|MBR|$ getting near zero in some cases where the x- or y-coordinates are nearly constant.

The database of sensor value information uses the country, the road, and the order number of mesh quadrilateral as the key values ensuring fast access to the data elements. It depends on service application, which information is stored to the database besides the sensor values with their spatial boundaries.

3.3.3 An example of accessing the mesh in an R-tree installed in Python

This example is written for libspatialindex version 1.8.5 and Python version 2.7.9. After installing the libraries libspatialindex and rtree, the R-tree index variable can be created:

```
from rtree import index
idx = index.Index()
```

In the following, the list quads contains the mesh consisting of quadrilaterals [X_1, X_2, X_3, X_4]. The long integer index numbers of the quadrilaterals are determined as described in section 4.3.1 and contained in the list Lint. Function getMBR returns the MBRs as a Python tuple (minx, miny, maxx, maxy), where the smallest and largest coordinates are selected from the vertices of the quadrilateral given as argument. MBR format (x,y,x,y) is required for inserting or making a query of one point (x,y).

The quadrilaterals of a road are inserted into the R-tree as follows:

```
for i in range(0,len(quads)):
     idx.insert(Lint[i],getMBR(quads[i]),obj=quads[i])
```

The query for a point=(x,y), such that we get both the index and the quadrilateral is made as follows:

```
respond=[[n.id,n.object] for n in
    idx.intersection((point[0],point[1],point[0],point[1]),
    objects=True)]
```

The following presents an example of accessing the mesh stored in the standard R-tree and the result of the program run. An example of a road having a mesh already exists. First, an R-tree is created and mesh inserted into it.

```
  # Creating the R-tree index for mesh of the road
  idx = index.Index()

  # Generating long integer indexes for each quadrilateral
of the current road
  print("We have indexes: country=3, road=15, and we generate
the long integer indexes for each quadrilateral of the
mesh")
  country=3;road=15;
  Lint=[]
  for i in range(0,len(quads)):
      ind=indexGener(country,road,i)
      Lint.append(ind)
  print ("The number of quadrilaterals is "+ str(len(quads)))
  print("\nInserting the quadrilaterals into the R-tree...")
  print("")

  # inserting quads into the R-tree
  for i in range(0,len(quads)):
      idx.insert(Lint[i],getMBR(quads[i]), obj=quads[i])
  Then, a query containing an original road point is formed.
  # making a query to the R-tree
  point=xy[1359] # select an original road point
  print("We make a query with a road point "+ str(point))
  print("")

  # the query, where we get both the long int index and the
quadrilateral(s)
  respond=[[n.id,n.object] for n in
  idx.intersection((point[0],point[1],point[0],point[1]),
objects=True)]
```

The respond of R-tree includes one quadrilateral and the query point is checked for being inside it.

```
  print("The respond from R-tree: ")
  print("")
  print("long integer index from the respond: "+ str(respond[0][0]))

  # printing the quadrilateral related to long integer index
  print("The quadrilateral related to long integer index = "+
  str(respond[0][1]))

  # testing if the point is inside the quadrilateral
  print("Result of testing, if the query point is inside the
quadrilateral: " +
  str(isInsidePolygon(point[0],point[1],respond[0][1])))
```

The identifications for the country, the road, and the mesh quadrilateral are parsed from the respond index number. Function fromIndex uses the parsing method described in section 4.3.1.

```
  print("The country, road, and order number of quadrilateral
from the respond:"
  + str(fromIndex(respond[0][0]))
```

The program run:

```
  We have an example of a road which have a mesh. We create
an R-tree, and insert the mesh in it
  We have indexes: country=3, road=15, and we generate the
long integer indexes for each quadrilateral of the mesh
  The number of quadrilaterals is 64

  Inserting the quadrilaterals into the R-tree...

  Making a query with a road point [7307328.103078449,
387951.0093313182]

  The respond from R-tree:
  long integer index from the respond = 30150000029
  The quadrilateral related to long integer index =
[[7307474.769375576, 388032.68673849193], [7307501.23424275,
387936.28092190035], [7306342.843770854, 387648.10530477605],
[7306307.472878471, 387740.87329855404]]
  Result of testing, if the query point is inside the
quadrilateral: True
  The country, road, and mesh order number from the respond:
[3, 15, 29]
```

4 Discussion

This article concentrated on the algorithms behind the mobile RWS data collection. It covered both the currently implemented algorithm and future thoughts on how the algorithm can be further improved in order to develop the system toward a real-time tracking application.

The currently implemented algorithm could be further improved by using triplets containing information on position (x- and y-coordinates) as well as the color instead of storing these data separately.

The R-tree is a balanced search tree like B+-tree, i.e., it reorganises the grouping of MBRs to be such that all leaf nodes are at the same height. Each internal node can contain a maximum number *M* of entries, whereas the minimum number of entries is some number *m* such that $m < \frac{M}{2}$. (Manopoulos 2006.)

In spite of the balanced tree property, there are plenty of issues in improving the effectiveness of R-tree. The many variants of R-tree implementations like R*-tree or R+-tree have been developed in order to improve the effectiveness in different situations. In a query, the tree is traversed starting from the root node to find which MBRs intersect. The traversing continues to the subtrees of a MBR, only if it intersects the bounding rectangle of the query region. In a standard R-tree, the grouped MBRs at the same level usually overlap. Therefore, there is no guarantee on good worst-case traversing. The R+-tree, a variant of the standard R-tree, tries to avoid overlapping of internal nodes by inserting an object into multiple leaves when necessary. This causes a slightly higher construction cost for R+-tree in inserting new data and maintaining the tree than in standard R-trees. On the other hand, the traversing for a query will usually perform better. In WiRMa project, using the R+-tree is justified, as R-tree is primarily subjected to queries.

In addition, many techniques for inserting elements into the R-tree. Different ordering of the inserted elements can result in an ineffective structure of the standard R-tree. To overcome this shortcoming many packing or bulk loading methods have been developed. These methods require advance knowledge of the data to be inserted in the R-tree. R-trees formed through packing or bulk loading methods are called static versions of R-trees. (Manopoulos 2006.) The static R-tree methods can also be utilised in the system for WiRMa project as the data consists of the static mesh structure.

The algorithms described in this article will be further tested and improved during the WiRMa project during year 2019. New follow-up projects taking the results from WiRMa further are being designed. The follow-up projects are planned to concentrate on larger datasets and will hold a larger role for the optimization of data analysis and visualisation.

Bibliography

Autioniemi, M. 2018. Photograph.

Bourke, P. 1987. Determining if a point lies on the interior of a polygon. Polygons and meshes. Accessed 28.5.2018. http://paulbourke.net/geometry/polygonmesh/

Finnish Geodetic Institute. n.d. Koordinaattimuunnokset. Accessed 28.5.2018. http://coordtrans.fgi.fi/transform.jsp

Guttman, T. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. Proceedings ACM SIGMOD Conference on Management of Data. Boston. MA. 47-57

Loukko.net. 2013. ETRS89-TM35FIN –karttaprojektiokonversiot. Loukko.net 29.4.2013. Accessed 28.5.2018. http://www.loukko.net/koord_proj/

Manopoulos, Y. et al. 2006. R-trees: Theory and Applications. ISBN 978-1-85233-977-7.

Appendix A

Data table

x	y	color
500	400,00	1
510	386,721	1
520	375,370	1
530	365,891	1
540	358,229	1
550	352,330	1
560	348,138	1
570	345,600	1
580	344,659	1
590	345,261	1
600	347,352	1
610	350,877	1
620	355,779	1
630	362,006	1
640	369,501	1
650	378,210	1
660	388,079	1
670	399,051	1
680	411,073	1
690	424,089	1
700	438,045	1
710	452,886	1
720	468,557	1
730	485,003	1

740	502,169	1
750	520,000	1
760	538,442	1
770	557,439	1
780	576,937	1
790	596,881	1
800	617,216	1
810	637,887	1
820	658,839	1
830	680,018	1
840	701,368	1
850	722,835	1
860	744,364	1
870	765,900	1
880	787,388	1
890	808,773	1
900	830,000	1
910	843,352	1
920	855,895	1
930	867,650	1
940	878,635	1
950	888,869	1
960	898,370	1
970	907,157	1
980	915,250	1
990	922,667	1
1000	929,427	1
1010	935,548	1
1020	941,050	2
1030	945,951	2
1040	950,270	2
1050	954,026	2
1060	957,238	2
1070	959,925	2
1080	962,105	2

1090	963,797	2
1100	965,021	2
1110	965,794	2
1120	966,136	2
1130	966,065	2
1140	965,601	2
1150	964,762	2
1160	963,567	2
1170	962,035	2
1180	960,185	2
1190	958,035	2
1200	955,604	2
1210	952,912	2
1220	949,976	2
1230	946,816	2
1240	943,451	2
1250	939,899	2
1260	936,180	2
1270	932,311	2
1280	928,312	2
1290	924,202	2
1300	920,000	2
1310	916,463	2
1320	912,899	2
1330	909,306	2
1340	905,681	2
1350	902,020	2
1360	898,322	2
1370	894,583	2
1380	890,801	2
1390	886,973	2
1400	883,096	2
1410	879,168	2
1420	875,186	2
1430	871,146	2

1440	867,047	2
1450	862,886	1
1460	858,659	1
1470	854,365	1
1480	850,000	1
1490	845,562	1
1500	841,047	1
1510	836,454	1
1520	831,779	1
1530	827,019	1
1540	822,173	1
1550	817,237	1
1560	812,208	1
1570	807,084	1
1580	801,862	1
1590	796,539	1
1600	791,113	1
1610	785,580	1
1620	779,939	1
1630	774,185	1
1640	768,318	1
1650	762,333	1
1660	756,228	1
1670	750,000	1
1680	742,551	1
1690	735,028	1
1700	727,453	1
1710	719,846	1
1720	712,228	1
1730	704,620	1
1740	697,042	1
1750	689,515	1
1760	682,060	1
1770	674,697	1
1780	667,448	1

1790	660,333	1
1800	653,373	1
1810	646,588	1
1820	640,000	1
1830	634,086	1
1840	628,392	1
1850	622,926	1
1860	617,695	1
1870	612,707	1
1880	607,970	1
1890	603,491	1
1900	599,278	1
1910	595,338	1
1920	591,678	1
1930	588,307	1
1940	585,232	1
1950	582,461	1
1960	580,000	1
1970	579,112	1
1980	578,611	1
1990	578,483	1
2000	578,717	1
2010	579,300	1
2020	580,219	1
2030	581,462	1
2040	583,016	1
2050	584,869	1
2060	587,009	1
2070	589,423	2
2080	592,098	2
2090	595,022	2
2100	598,182	2
2110	601,567	2
2120	605,163	2
2130	608,957	2

2140	612,938	2
2150	617,094	2
2160	621,410	2
2170	625,876	2
2180	630,478	2
2190	635,204	2
2200	640,041	2
2210	644,977	2
2220	650,000	2
2230	655,097	2
2240	660,255	2
2250	665,462	2
2260	670,706	2
2270	675,973	2
2280	681,253	2
2290	686,531	2
2300	691,795	2
2310	697,034	2
2320	702,234	2
2330	707,383	2
2340	712,469	2
2350	717,478	2
2360	722,399	2
2370	727,219	2
2380	731,926	2
2390	736,506	2
2400	740,948	2
2410	745,240	2
2420	749,367	2
2430	753,319	2
2440	757,082	2
2450	760,644	2
2460	763,993	2
2470	767,116	2
2480	770,000	2

The road weather station (RWS) network in the Northern parts of Finland is too sparse for the purposes of delivering detailed information of the road weather. Therefore, the Interreg Nord funded Winter Road Maintenance (WiRMa) project researches the possibility of using large goods vehicles (LGV) as mobile RWSs by equipping them with sensors measuring a number of variables directly on the road.

The collected sensor data is stored in a cloud storage and delivered to road maintenance personnel through a web based user interface (UI) as well as varied research purposes for road weather predictions through application programming interface (API). This article describes the technical points in compressing the collected measurement data. Both the currently implemented algorithm and thoughts on future development are shared.

WiRMa project is implemented in years 2016-2019. The lead partner is Lapland University of Applied Sciences Ltd. The co-beneficiaries are Luleå University of Technology, Arctic University of Norway, Foreca Ltd., Finnish Meteorological Institute, and Casselgren Innovation AB.

Interreg Nord
European Regional Development Fund

EUROPEAN UNION

REGIONAL COUNCIL OF LAPLAND

WiRMa
real-time road conditions

LAPIN AMK
Lapland University of Applied Sciences

www.lapinamk.fi

ISBN 978-952-316-258-7