Final report

PhD-course: Automate your GIS - Scripting in Python (NGEO306)

Author: Ana Luisa Maffini

Date: April 2023

---

1. Introduction

For the final project of the NGEO306 PhD course I chose to develop a QGIS processing plugin that could calculate the Potential Movement Difference (PMD) metrics that I have been developing with a colleague in my PhD.
The proposed plugin is called PMD Index. It was created using Plugin Builder in QGIS.

The PMD Index plugin is constituted, so far, of a help folder, an icons folder, a pycache folder, an icon.png image, a metadata.txt file, a readme.md file, and four python files:

- __init__.py
- pmd_index.py
- pmd_index_algorithm.py
- pmd_index_provider.py

Because this report aims to present the main parts of the code that were written by me, it will focus mostly on the PMD algorithm script. There were some alterations in the code in the other files that required me to write in the scripts, but it is the algorithm file that contains what the plugin must calculate.
The report is structure in four parts: first, this brief introduction; then, the PMD plugin, where the metrics are explained and the plugin is explained; next is the PMD algorithm, where the code is explained; and finally, a conclusion for the report.

2. The PMD plugin

PMD Index is a set of urban network metrics that consider the urban space as a network graph for its analysis.

Potential Movement (PM) is the frequency with which streets in the urban street network belong to the shortest paths of a population when heading to urban facilities, acting as an indicator of population flows.

The PMD Index considers **origins** and **destinations** to estimate potential movements. It also uses **weights** according to the size (or attractiveness) of destination locations and the size of origin locations (number of people or households).

The index calculates the differences in the Potential Movement values of specific populations' groups when compared to a General value of Potential Movement.

Currently, the PMD Index plugin calculates eight measures:

- The General Potential Movement (GPM)
- The Potential Movement for a Group A of the Population (PMa)
- The Potential Movement for a Group B of the Population (PMb)
- The normalized value of GPM
- The normalized value of PMa
- The normalized value of PMb
- The Potential Movement Difference for Group A (PMDa)
- The Potential Movement Difference for Group B (PMDb)

2.1 The PMD metrics

2.1.1Potential Movement:

The **PM** is a centrality-based network measure that is restricted to ordered pairs of nodes (origins and destinations). Only the pairs with these opposite attributes are considered in the calculation.

For the **origin** nodes, the PM uses the location of the households weighted by the number of

residents.

For **destination** nodes, several different activity spaces can be considered. They are usually weighted by the number of establishments in the node.

The PM metric is defined by the equation:

$$PM_{ij}(k) = \sum_{k=1}^{i,j \epsilon G} \frac{W_i \cdot W_j}{d_{i,j}} \cdot \frac{g_{i.j}^{(k)}}{g_{i,j}}$$

Where, $PM_{ij}(k)$ is the PM of node K, where all the nodes *i* contain origins and all the nodes *j* contain destinations. W_i is the weight of node *i* and W_j is the weight of node *j*, d_ij is the length of the shortest path between *i* and *j* (distance), $g_ij(k)$ is the number of shortest paths between *i* and *j* that pass through k, and g_ij is the total number of shortest paths between *i* and *j*.

Currently in the plugin, the PM metric is calculated three times. When the PM is calculated for the entire population of the city, it is called **General Potential Movement (GPM)**, it is a reference value for what the PM could be if the entire population of the city was of the same group. When the PM is calculated for a specific population group, it is called Potential Movement of group *x* (PMx). At the moment in the plugin there are only two population groups being considered, A and B, and their sum dos not need to equal the entire population.

2.1.2 Normalized Potential Movement

After computing the GPM and the two PM metrics (PMa and PMb), the plugin normalizes the values by the total amount of population that was considered in the metric. For the GPM that number is the total amount of the city's population, whereas for the PMa and PMb is the total amount of that specific population group. The equation of the normalized PM is:

$$PMNorm(i) = \frac{PM_{ij}(k)}{\sum W_i}$$

Where, $PMNorm(i)$ is the normalized value of PM for the group *i*, $PM_{ij}(k)$ is the PM value of node_k_ for that population's PM, and W_i is the sum of all the weight values of population *i*.

2.1.3 Potential Movement Difference

PMD is the value of the difference between the population's group PM and the GPM. It indicates how different the PM values of that population are when compared to what they might be if the entire population of the city belonged to that population's group. The PMD metric is:

$$PMD_{(K)}^{S} = PM_{(k)}^{S} - GPM_{(k)}$$

Where, $PMD_{(K)}^{S}$ is the PMD of node k for the population group S, $PM_{(k)}^{S}$ is the PM of node k for the population group S, and $GPM_{(k)}$ is the GPM value of node k.

2.2 The PMD Index plugin

The PM Index plugin is a processing plugin in QGIS. Once installed it is located in the processing toolbar.

When started, the plugin interface requires some inputs from the user. Figure 1 shows the plugin's interface. The user is required to select a vector layer as input. The vector layer must be of a LineString geometry type and can have any CRS, although the user must be aware that for geometric calculations the CRS must be a projected CRS.

Once the input vector layer is selected, the user must define the analysis type, the connecting rule and the radius for the analysis. There are two types of analysis, they can be topological or geometric. For the connecting rule, there are three options available to the user, "overlapping vertices", "crossing lines", and "overlapping vertices + crossing lines". The radius is set at 0, which means a global analysis of the metrics, but the user can define a different number and perform a local analysis as well.

After these definitions, the user must select the field columns from the input vector layer attribute table for the impedance, the destinations, the origin of the entire population, the origin of population group A, and the origin of population group B. The impedance is an optional field, if the user does not select any the algorithm considers the value as 1. The

impedance is a value that multiplies the distance in the calculations. It is used to increase or decrease the distance.

Finally, the user must inform the output vector layer. Currently, the option for saving in a temporary layer is not working.

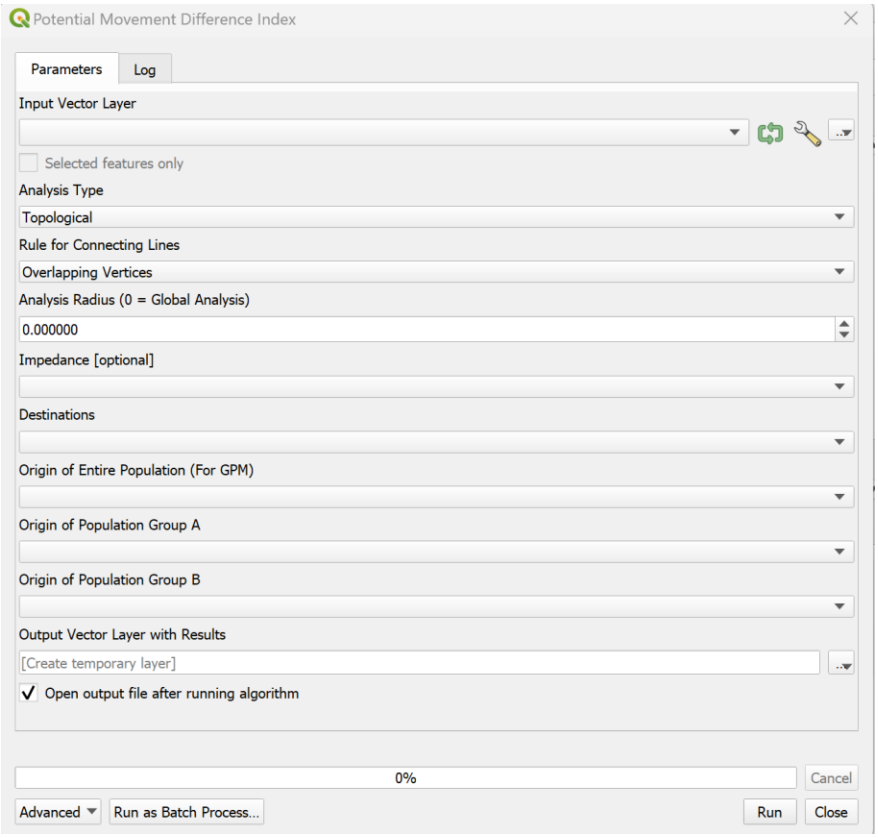

Figure 1 – PMD Index plugin interface in QGIS.

After the calculations are done, the plugin will return a new LineString vector layer with the same field columns from the input layer plus the new ones that are generated. The input vector layer is not altered.

3. The PMD algorithm

To calculate the PMD Index metrics, the PMD algorithm's code is structured in nine main steps:

1. The initial imports
2. Definition of constants
3. Addition of parameters
4. Start of the algorithm and definition of the input parameters
5. Start the Edge Object
6. Definition of the object parameters
7. Start of the Edge processing
    1. Computation of shortest paths
        1. Heap creation
        2. Heapsort
    2. Calculation of the metrics
8. Update the attribute table
9. Save the output file

3.1 The PMD Index algotithm's code

1. The initial imports:

```
import os
from qgis.PyQt.QtGui import QIcon
from qgis.PyQt.QtCore import QCoreApplication, QVariant
from qgis.core import (NULL, QgsProcessing, QgsProcessingAlgorithm, QgsProject, QgsVectorFileWriter,
QgsDistanceArea, QgsField,  QgsProcessingParameterField, QgsProcessingParameterNumber, QgsProcessi
ngParameterEnum, QgsProcessing,  QgsProcessingAlgorithm, QgsProcessingParameterFeatureSource, Qgs
ProcessingParameterFeatureSink, QgsField, QgsProject)
```

2. Definition of constants:

```
class PMDIndexAlgorithm(QgsProcessingAlgorithm):

    INPUT_VECTOR_LAYER = 'IMPUT_VECTOR_LAYER'
    ANALYSIS = 'ANALYSIS'
    GEOMRULE = 'GEOMRULE'
    ID_FIELD = 'ID_FIELD'
    RADIUS = 'RADIUS'
    IMPEDANCE = 'IMPEDANCE'
    DESTINATION = 'DESTINATION'
    ORIGIN_GLOBAL = 'ORIGIN_GLOBAL'
    ORIGIN_A = 'ORIGIN_A'
    ORIGIN_B = 'ORIGIN_B'
    OUTPUT_VECTOR_LAYER = 'OUTPUT_VECTOR_LAYER'
```

3. Addition of parameters:

a) The first parameter that is added is for the input vector layer:

```
def initAlgorithm(self, config):

    self.addParameter(
        QgsProcessingParameterFeatureSource(
            self.INPUT_VECTOR_LAYER,
            self.tr('Input Vector Layer'),
            [QgsProcessing.TypeVectorAnyGeometry]))
```

b) Next, we add the parameter for the analysis type. In here, there are two options (topological or geometric) that the user can chose from:

```
    self.addParameter(
        QgsProcessingParameterEnum(
            self.ANALYSIS,
            self.tr('Analysis Type'),
            ['Topological', 'Geometric'],
            defaultValue=[0]))
```

c) Next, we add the parameter for the rule for connecting lines. In here, there are three rules for how the algorithm is suposed to consider the connection of the lines. The the user must choose one:

```
    self.addParameter(
        QgsProcessingParameterEnum(
            self.GEOMRULE,
            self.tr('Rule for Connecting Lines'),
            ['Overlapping Vertices','Crossing Lines', 'Overlapping Vertices + Crossing Lines'],
            defaultValue=[0]))
```

d) Next, we add the parameter for the analysis radius. In here, if no number is typed by the user, the algorithm considers it as 0, and performs a global analysis:

```
    self.addParameter(
        QgsProcessingParameterNumber(
            self.RADIUS,
            self.tr('Analysis Radius (0 = Global Analysis)'),
            QgsProcessingParameterNumber.Double))
```

e) Next, we add the impedance parameter. This parameter is an optional one and can be selected by the user from one of the input vector field columns:

```
        self.addParameter(
            QgsProcessingParameterField(
                self.IMPEDANCE,
                self.tr('Impedance'),
                QgsProcessingParameterField.Numeric,
                parentLayerParameterName='IMPUT_VECTOR_LAYER',
                allowMultiple = False,
                optional = True))
```

f) Next, we add the destination parameter. This parameter must be selected by the user from one of the input vector field columns:

```
        self.addParameter(
            QgsProcessingParameterField(
                self.DESTINATION,
                self.tr('Destinations'),
                QgsProcessingParameterField.Numeric,
                parentLayerParameterName='IMPUT_VECTOR_LAYER',
                allowMultiple = False,
                optional = False))
```

g) Next, we add the origins of the entire population parameter. This parameter must be selected by the user from one of the input vector field columns:

```
        self.addParameter(
            QgsProcessingParameterField(
                self.ORIGIN_GLOBAL,
                self.tr('Origin of Entire Population (For GPM)'),
                QgsProcessingParameterField.Numeric,
                parentLayerParameterName='IMPUT_VECTOR_LAYER',
                allowMultiple = False,
                optional = False))
```

h) Next, we add the origin of population group A parameter. This parameter must be selected by the user from one of the input vector field columns:

```
        self.addParameter(
            QgsProcessingParameterField(
                self.ORIGIN_A,
                self.tr('Origin of Population Group A'),
                QgsProcessingParameterField.Numeric,
                parentLayerParameterName='IMPUT_VECTOR_LAYER',
                allowMultiple = False,
                optional = False))
```

i) Next, we add the origin of population group B parameter. This parameter must be selected by the user from one of the input vector field columns:

```
        self.addParameter(
            QgsProcessingParameterField(
                self.ORIGIN_B,
                self.tr('Origin of Population Group B'),
                QgsProcessingParameterField.Numeric,
                parentLayerParameterName='IMPUT_VECTOR_LAYER',
                allowMultiple = False,
                optional = False))
```

j) Finally, we add the output vector parameter:

```
        self.addParameter(
            QgsProcessingParameterFeatureSink(
                self.OUTPUT_VECTOR_LAYER,
                self.tr('Output Vector Layer with Results'),
                type=QgsProcessing.TypeVectorAnyGeometry,
                createByDefault=True,
                supportsAppend=True,
                defaultValue=None))
```

4. Now we start the processing algorithm and define the input parameters:

```
    def processAlgorithm(self, parameters, context, feedback):
        inputEdges = self.parameterAsVectorLayer(parameters, self.INPUT_VECTOR_LAYER, context)
        metricsL = [0,1,2,3,4, 5, 6, 7]
        impField = self.parameterAsFields(parameters, self.IMPEDANCE, context)
        destinationField = self.parameterAsFields(parameters, self.DESTINATION, context)
        originGField = self.parameterAsFields(parameters, self.ORIGIN_GLOBAL, context)
        originAField = self.parameterAsFields(parameters, self.ORIGIN_A, context)
        originBField = self.parameterAsFields(parameters, self.ORIGIN_B, context)
        analysisType = self.parameterAsEnum(parameters, self.ANALYSIS, context)
        radius = self.parameterAsDouble(parameters, self.RADIUS, context)
        geomR = self.parameterAsEnum(parameters, self.GEOMRULE, context)
        outPath = self.parameterAsOutputLayer(parameters, self.OUTPUT_VECTOR_LAYER, context)
```

5. We then start the Edge Object (edges of the network):

```
        class EdgeObj:
            def __init__(self, featCount, feat, destinationF, originGF, originAF, originBF, impF, analysisType, metri
csL):
                self.id = feat.id()
                self.heapPos = -1 #current position of the edge inside the heap
                self.neighA = [] #list of connected edges
                self.geom = feat.geometry()
                self.length = QgsDistanceArea().measureLength(feat.geometry()) if analysisType == 1 else 1
```

6. We define the object parameters.

a) First, the metrics that will be calculated in the algorithm:

```
        if 0 in metricsL: self.gpm = 0
        if 1 in metricsL: self.pma = 0
        if 2 in metricsL: self.pmb = 0
        if 3 in metricsL: self.gpmNorm = 0
        if 4 in metricsL: self.pmaNorm = 0
        if 5 in metricsL: self.pmbNorm = 0
        if 6 in metricsL: self.pmda = 0
        if 7 in metricsL: self.pmdb = 0
```

b) Then, the destination, origin, originA, originB, and impedance values. We also define how the algorithm should consider them:

```
        self.destination, self.originG, self.originA, self.originB, self.imp = 0,0,0,0,0
        for i in range(len(destinationF)):
          if feat.attribute(destinationF[i]) != NULL: self.destination += feat.attribute(destinationF[i])
        for i in range(len(originGF)):
          if feat.attribute(originGF[i]) != NULL: self.originG += feat.attribute(originGF[i])
          else: 0
        for i in range(len(originAF)):
          if feat.attribute(originAF[i]) != NULL: self.originA += feat.attribute(originAF[i])
          else: 0
        for i in range(len(originBF)):
          if feat.attribute(originBF[i]) != NULL: self.originB += feat.attribute(originBF[i])
          else: 0
        if impF == ['0']: self.imp = 1
        else:
          for i in range(len(impF)):
            if feat.attribute(impF[i]) != NULL: self.imp += feat.attribute(impF[i])
```

c) Next we verify if highest id number is lower than number of features in the object in order to avoid potential conflicts with matrices' size:

```
    def verifyFeatCount(inputFeat):
      featCount = inputFeat.featureCount()
      for feat in inputFeat.getFeatures(): featCount = max(featCount, feat.id()+1)
      return featCount
```

7. Then, we start the edges processing:

```
    edgesCount = verifyFeatCount(inputEdges)
    edgesA = [] #array that stores network edges
    for edge in inputEdges.getFeatures():
      if edge.id() % 50 == 0: feedback.pushInfo(f'Initializing Edge {edge.id()}')
      edgesA.append(EdgeObj(edgesCount, edge, destinationField, originGField, originAField, originBFiel
d, impField, analysisType, metricsL))
      for i in range(len(edgesA)-1):
        if (geomR==0 and edgesA[-1].geom.touches(edgesA[i].geom)) or (geomR==1 and edgesA[-1].geo
m.crosses(edgesA[i].geom)) or (geomR==2 and (edgesA[-1].geom.crosses(edgesA[i].geom) or edgesA[-1].
geom.touches(edgesA[i].geom))):
          dist = (edgesA[-1].imp*edgesA[-1].length + edgesA[i].imp*edgesA[i].length)/2
          if dist <= radius or radius == 0.0:
            edgesA[-1].neighA.append([edgesA[i], dist])
            edgesA[i].neighA.append([edgesA[-1], dist])
```

7. Next, we start the Edge processing. First, we compute the shortest paths using the djikstra algorithm with binary heap as priority queue. This process is structured in two parts: first the heap creation, then the heapsort. Because this part of the code was retrieved from another script that also calculates urban network metrics, and that code was written by a different person, I will not go into much detail for this part.

7.1.1. The heap creation:

```
  for source in edgesA:
    if source.id % 50 == 0: feedback.pushInfo(f'Shortest Paths Edge {source.id}')
    finitePos = 0
    costA = [99999999999999 for i in range(edgesCount)]
    costA[source.id] = 0 #distance from the source edge to itself is zero
    for ind in range(len(source.neighA)): costA[source.neighA[ind][0].id] = source.neighA[ind][1]
    heap = [edgesA[0] for i in range(len(source.neighA) + 1)]
    for destin in edgesA:
      if feedback.isCanceled():
        break
      if costA[destin.id] == 99999999999999:
        heap.append(destin)
        destin.heapPos = len(heap) - 1
      else:
        heap[finitePos] = destin
        destin.heapPos = finitePos
        n = finitePos
        finitePos += 1
        parent = int((n-1)/2)
        while n !=0 and costA[heap[n].id] < costA[heap[parent].id]:
          heap[n].heapPos, heap[parent].heapPos = parent, n
          heap[n], heap[parent] = heap[parent], heap[n]
          n = parent
          parent = int((n-1)/2)
```

7.1.2. The heapsort can be separated in four parts. a) The first one:

```
pivotA = [[] for i in range(edgesCount)] #array of pivot edges in shortest paths
level = [99999999999999 for i in range(edgesCount)]
sortedA = []
numSP = [0 for i in range(edgesCount)]
numSP[source.id], level[source.id] = 1,0
for ind in range(len(source.neighA)):
  numSP[source.neighA[ind][0].id] = 1
  level[source.neighA[ind][0].id] = 1
while heap != []:
  closest = heap[0]
  if costA[closest.id] <= radius or radius == 0.0: sortedA.append(closest)
  if finitePos > 0:
    heap[0].heapPos, heap[finitePos-1].heapPos = finitePos-1, 0
    heap[0], heap[finitePos-1] = heap[finitePos-1], heap[0]
    heap[finitePos-1].heapPos, heap[-1].heapPos = len(heap)-1, finitePos-1
    heap[finitePos-1], heap[-1] = heap[-1], heap[finitePos-1]
    finitePos -= 1
  heap.pop(len(heap)-1)
```

b) The second part:

```
n = 0
lh = finitePos
posChild1, posChild2 = n*2+1, n*2+2
if posChild2 <= lh-1:
  costChild1, costChild2 = costA[heap[n*2+1].id], costA[heap[n*2+2].id]
  if any(x < costA[heap[n].id] for x in [costChild1,costChild2]):
    if costChild1 <= costChild2: sc = posChild1
    else: sc = posChild2
  else: sc = -1
elif posChild2 == lh:
  if costA[heap[n*2+1].id] < costA[heap[n].id]: sc = posChild1
  else: sc = -1
else: sc = -1
```

c) The third part:

```
while sc >= 0:
  heap[n].heapPos, heap[sc].heapPos = sc, n
  heap[n], heap[sc] = heap[sc], heap[n]
  n = sc
  lh = len(heap)
  posChild1, posChild2 = n*2+1, n*2+2
  if posChild2 <= lh-1:
    costChild1, costChild2 = costA[heap[n*2+1].id], costA[heap[n*2+2].id]
    if any(x < costA[heap[n].id] for x in [costChild1,costChild2]):
      if costChild1 <= costChild2: sc = posChild1
      else: sc = posChild2
    else: sc = -1
  elif posChild2 == lh:
    if costA[heap[n*2+1].id] < costA[heap[n].id]: sc = posChild1
    else: sc = -1
  else: sc = -1
```

d) The fourth part:

```
            for ind in range(len(closest.neighA)):
              if closest.neighA[ind][0].heapPos < len(heap):
                cost = costA[closest.id] + closest.neighA[ind][1]
                prevCost = costA[closest.neighA[ind][0].id]
                if prevCost > cost and (radius == 0.0 or cost <= radius):
                  costA[closest.neighA[ind][0].id], level[closest.neighA[ind][0].id] = cost, level[closest.id] + 1
                  pivotA[closest.neighA[ind][0].id] = []
                  pivotA[closest.neighA[ind][0].id].append(closest)
                  numSP[closest.neighA[ind][0].id] += numSP[closest.id]

                  n = closest.neighA[ind][0].heapPos
                  if prevCost == 99999999999999:
                    heap[finitePos].heapPos, closest.neighA[ind][0].heapPos = n, finitePos
                    heap[n], heap[finitePos] = heap[finitePos], closest.neighA[ind][0]
                    n = finitePos
                    finitePos += 1
                  parent = int((n-1)/2)
                  while n !=0 and costA[heap[n].id] < costA[heap[parent].id]:
                    heap[n].heapPos, heap[parent].heapPos = parent, n
                    heap[n], heap[parent] = heap[parent], heap[n]
                    n = parent
                    parent = int((n-1)/2)

                elif source.id != closest.id and costA[closest.neighA[ind][0].id] == cost and (radius == 0.0 or c
ost <= radius):
                  pivotA[closest.neighA[ind][0].id].append(closest)
                  numSP[closest.neighA[ind][0].id] += numSP[closest.id]
```

7.2. Next, we begin the calculations for the metrics.

a) We start with defining the the temporary variables for the metrics:

```
        if 0 in metricsL: gpmTemp = [0 for i in range(edgesCount)]
        if 1 in metricsL: pmaTemp = [0 for i in range(edgesCount)]
        if 2 in metricsL: pmbTemp = [0 for i in range(edgesCount)]
        if 3 in metricsL: gpmNormTemp = [0 for i in range(edgesCount)]
        if 4 in metricsL: pmaNormTemp = [0 for i in range(edgesCount)]
        if 5 in metricsL: pmbNormTemp = [0 for i in range(edgesCount)]
        if 6 in metricsL: pmdaTemp = [0 for i in range(edgesCount)]
        if 7 in metricsL: pmdbTemp = [0 for i in range(edgesCount)]
```

b) We need to sum the values of originG, originA, and originB for the gpmNorm, pmaNorm, and pmbNorm metrics:

```
        originGtot = sum([edge.originG for edge in edgesA])
        originAtot = sum([edge.originA for edge in edgesA])
        originBtot = sum([edge.originB for edge in edgesA])
```

c) We define the tension variables that will be used in the metrics:

```
        while sortedA != []:
            farest = sortedA[-1]
            cost = costA[farest.id]
            sortedA.pop(len(sortedA)-1)
            tensionG = source.destination * farest.originG
            tensionA = source.destination * farest.originA
            tensionB = source.destination * farest.originB
```

d) We then start the metrics calculations. First, we calculate the PM values for the connected edges:

```
            for neigh in pivotA[farest.id]:
                if feedback.isCanceled():
                    break
                if numSP[farest.id] > 0 and (radius == 0.0 or cost <= radius):
                    if 0 in metricsL: gpmTemp[neigh.id] += (numSP[neigh.id]/numSP[farest.id])*((tensionG/(lev
el[farest.id]+1))+gpmTemp[farest.id])
                    if 1 in metricsL: pmaTemp[neigh.id] += (numSP[neigh.id]/numSP[farest.id])*((tensionA/(lev
el[farest.id]+1))+pmaTemp[farest.id])
                    if 2 in metricsL: pmbTemp[neigh.id] += (numSP[neigh.id]/numSP[farest.id])*((tensionB/(lev
el[farest.id]+1))+pmbTemp[farest.id])
```

e) Next, we calculate the PM values for the source edges:

```
            if pivotA[farest.id] == [] and level[farest.id] == 1 and (radius == 0.0 or cost <= radius):
                if 0 in metricsL: gpmTemp[source.id] += (numSP[neigh.id]/numSP[farest.id])*((tensionG/(level
[farest.id]+1))+gpmTemp[farest.id])
                if 1 in metricsL: pmaTemp[source.id] += (numSP[neigh.id]/numSP[farest.id])*((tensionA/(leve
l[farest.id]+1))+pmaTemp[farest.id])
                if 2 in metricsL: pmbTemp[source.id] += (numSP[neigh.id]/numSP[farest.id])*((tensionB/(leve
l[farest.id]+1))+pmbTemp[farest.id])
```

f) Then, we calculate the metrics for the edges in the shortest paths:

```
        if (0 in metricsL) and (radius == 0.0 or cost <= radius): gpmTemp[farest.id] += tensionG/(level[fa
rest.id]+1)
        if (1 in metricsL) and (radius == 0.0 or cost <= radius): pmaTemp[farest.id] += tensionA/(level[fa
rest.id]+1)
        if (2 in metricsL) and (radius == 0.0 or cost <= radius): pmbTemp[farest.id] += tensionB/(level[fa
rest.id]+1)
        if (3 in metricsL) and (radius == 0.0 or cost <= radius): gpmNormTemp[farest.id] += gpmTemp[fa
rest.id]/originGtot
        if (4 in metricsL) and (radius == 0.0 or cost <= radius): pmaNormTemp[farest.id] += pmaTemp[fa
rest.id]/originAtot
        if (5 in metricsL) and (radius == 0.0 or cost <= radius): pmbNormTemp[farest.id] += pmbTemp[f
arest.id]/originBtot
        if (6 in metricsL) and (radius == 0.0 or cost <= radius): pmdaTemp[farest.id] += (pmaNormTemp
[farest.id] - gpmNormTemp[farest.id])
        if (7 in metricsL) and (radius == 0.0 or cost <= radius): pmdbTemp[farest.id] += (pmbNormTemp
[farest.id] - gpmNormTemp[farest.id])
```

g) Finally, we update the metrics values:

```
        if 0 in metricsL: farest.gpm += gpmTemp[farest.id]
        if 1 in metricsL: farest.pma += pmaTemp[farest.id]
        if 2 in metricsL: farest.pmb += pmbTemp[farest.id]
        if 3 in metricsL: farest.gpmNorm += gpmNormTemp[farest.id]
        if 4 in metricsL: farest.pmaNorm += pmaNormTemp[farest.id]
        if 5 in metricsL: farest.pmbNorm += pmbNormTemp[farest.id]
        if 6 in metricsL: farest.pmda += pmdaTemp[farest.id]
        if 7 in metricsL: farest.pmdb += pmdbTemp[farest.id]
```

8. After computing the metrics, we update the table of contents.

a) To do so, we start creating new attributes fields for the computed metrics:

```
    strBegin = "T" if analysisType == 0 else "G"
    strMid = "g" if radius == 0.0 else str(int(radius))
    if len(strMid) > 5: strBegin += strMid[0:5]
    else: strBegin += strMid

    if 0 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "GPM")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "GPM",QVariant.Double)])
       inputEdges.updateFields()
       gpmIndex = inputEdges.fields().indexFromName(strBegin + "GPM")
    if 1 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "PMa")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "PMa",QVariant.Double)])
       inputEdges.updateFields()
       pmaIndex = inputEdges.fields().indexFromName(strBegin + "PMa")
    if 2 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "PMb")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "PMb" ,QVariant.Double)])
       inputEdges.updateFields()
       pmbIndex = inputEdges.fields().indexFromName(strBegin + "PMb")
    if 3 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "GPMNorm")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "GPMNorm",QVariant.Double)])
       inputEdges.updateFields()
       gpmNormIndex = inputEdges.fields().indexFromName(strBegin + "GPMNorm")
    if 4 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "PMaNorm")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "PMaNorm",QVariant.Double)])
       inputEdges.updateFields()
       pmaNormIndex = inputEdges.fields().indexFromName(strBegin + "PMaNorm")
    if 5 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "PMbNorm")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "PMbNorm",QVariant.Double)])
       inputEdges.updateFields()
       pmbNormIndex = inputEdges.fields().indexFromName(strBegin + "PMbNorm")
    if 6 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "PMDa")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "PMDa",QVariant.Double)])
       inputEdges.updateFields()
       pmdaIndex = inputEdges.fields().indexFromName(strBegin + "PMDa")
    if 7 in metricsL:
       inputEdges.fields().indexFromName(strBegin + "PMDb")
       inputEdges.dataProvider().addAttributes([QgsField(strBegin + "PMDb",QVariant.Double)])
       inputEdges.updateFields()
       pmdbIndex = inputEdges.fields().indexFromName(strBegin + "PMDb")
```

b) Then, we define the dictionary with the metrics variables and update the table of contents:

```
    for edge in edgesA:
        metricsD = {}
        if 0 in metricsL: metricsD[gpmIndex] = edge.gpm
        if 1 in metricsL: metricsD[pmaIndex] = edge.pma
        if 2 in metricsL: metricsD[pmbIndex] = edge.pmb
        if 3 in metricsL: metricsD[gpmNormIndex]= edge.gpmNorm
        if 4 in metricsL: metricsD[pmaNormIndex] = edge.pmaNorm
        if 5 in metricsL: metricsD[pmbNormIndex] = edge.pmbNorm
        if 6 in metricsL: metricsD[pmdaIndex] = edge.pmda
        if 7 in metricsL: metricsD[pmdbIndex] = edge.pmdb

        inputEdges.dataProvider().changeAttributeValues({edge.id : metricsD})
```

9. Finally, we save the output vector layer, which is a new vector layer, and delete the metrics attribute fields from the input vector layer.

```
    if outPath != "":
        crs = QgsProject.instance().crs()
        save_options = QgsVectorFileWriter.SaveVectorOptions()
        save_options.driverName = "ESRI Shapefile"
        save_options.fileEncoding = "System"
        context = QgsProject.instance().transformContext()
        QgsVectorFileWriter.writeAsVectorFormat(inputEdges, outPath, "System", crs, "ESRI Shapefile")
        metricsOut = []
        if 0 in metricsL: metricsOut.append(gpmIndex)
        if 1 in metricsL: metricsOut.append(pmaIndex)
        if 2 in metricsL: metricsOut.append(pmbIndex)
        if 3 in metricsL: metricsOut.append(gpmNormIndex)
        if 4 in metricsL: metricsOut.append(pmaNormIndex)
        if 5 in metricsL: metricsOut.append(pmbNormIndex)
        if 6 in metricsL: metricsOut.append(pmdaIndex)
        if 7 in metricsL: metricsOut.append(pmdbIndex)
        inputEdges.dataProvider().deleteAttributes(metricsOut)
        inputEdges.updateFields()

    # Return the results of the algorithm
    return {self.OUTPUT_VECTOR_LAYER: outPath}
```

4. Conclusion

The PMD Index plugin calculates the PMD metrics and returns a new vector layer with the output results. The python script for the algorithm works, calculating the eight metrics (GPM, PMa, PMb, GPMNormalized, PMaNormalized, PMbNormalized, PMDa, and PMDb).

At its current state, the plugin is already operational and can be used to calculate the PMD index for two different population groups. For a city of approximately 300,000 inhabitants, with a street network of 10,000 lines, the plugin takes approximately 30 minutes to finish its processing. That is acceptable but can be a problem when running the calculations for bigger cities, or metropolitan regions, where systems of over 150,000 lines can be found.

Future developments for the plugin include a) to allow the output file to be saved as a temporary layer; b) to allow for the selection of any number of population groups for comparison; c) to improve the code for the shortest path calculation, so that the algorithm can run faster; and d) to launch it for QGIS.