Graph matching and alignment

Xavier Dupré
http://www.xavierdupre.fr/

3 mai 2013

Résumé

How to compute a distance between two graphs ? How to find what modifications were introduced in a graph ? This paper gives an answer to those questions for graphs without cycles. The outcome is a way to match two graphs altogether and to build an alignment between edges and vertices from both graphs. This method is based on Levenstein's edit distance.

0.1 Problem definition

This *graph distance* aims at computing a distance between graphs but also to align two graphs and to merge them into a single one. For example, let's consider the graphs of Figure 1. We would like to merge them and to know which vertices were merged, which ones were added and deleted.

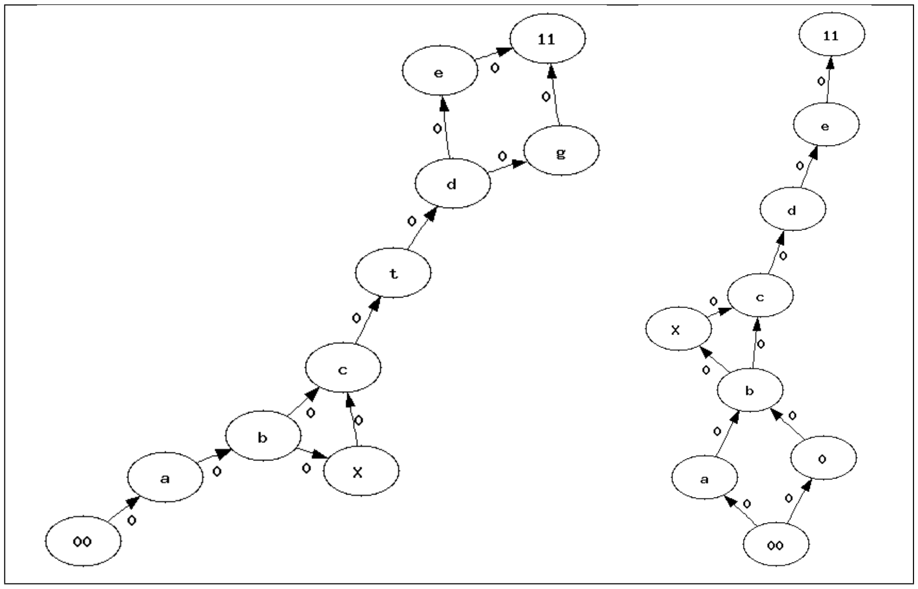

FIGURE 1 : *Two graphs we would like to merge into a single one. Vertices 00 and 11 represents the only root and the only leave.*

The following ideas and algorithm are only applicable on graphs without cycles. To simplify, we assume there are only one root and one leave. If there are mulitple, we then create a single root we connect to all the existing ones. We do the same for the unique leave we create if there are multiple. It will have all the existing ones as predecessors.

We also assume each vertex and each edge holds a label used during the matching. It is better to match vertices or edges holding the same label. A weight can be introduced to give more important to some elements (vertex, edge).

0.2 First approach

Step 1 : edit distance

The main idea consists in using Levenstein's edit distance. This algorithm applies on sequences but not on graphs. But because both graphs do not contain any cycle, we can extract all paths from them. Every path starts with the same vertex - the only root - and ends with the same one - the only leave -. We also consider each edge or vertex as an element of the sequence. Before describing the edit distance, let's denote p_1 as a path from the first graph, p_2 as a path from the second one. $p_k(i)$ is the element i of this sequence. Following Levenstein description, we denote $d(i, j)$ as the distance between the two subsequences $p_1(1..i), p_2(1..j)$. Based on that, we use an edit distance defined as follows :

$$d(i,j) = \min \left\{ \begin{array}{l} d(i-1,j) + insertion(p_1(i)) \\ d(i,j-1) + insertion(p_2(j)) \\ d(i-1,j-1) + comparison(p_1(i), p_2(j)) \end{array} \right. \tag{1}$$

First of all, we are not only interested in the distance but also in the alignment which would imply to keep which element was chosen as a minimum for each $d(i, j)$. If we denote n_k the length of path k, then $d(n_1, n_2)$ is the distance we are looking for.

Second, if two paths do not have the same length, it implies some elements could be compared between each others even if one is an edge and the other one is a vertex. This case is unavoidable if two paths have different lengths.

Third, the weight we use for the edit distance will be involved in a kind of tradeof : do we prefer to boost the structure or the label when we merge the graphs. Those weights should depend on the task, whether or not it is better to align vertices with the same label or to keep the structure. Here are the chosen weights :

operation	**weight**	**condition**
$insertion(c)$	$w(c)$, weight held by the edge or the vertex	
$comparison(a, b)$	0	if vertices a and b share the same label
$comparison(a, b)$	0	if edges a and b share the same label and if vertices at both ends share the same label
$comparison(a, b)$	$w(a) + w(b)$	if edges a and b share the same label and if vertices at both ends do not share the same label
$comparison(a, b)$	$\frac{w(a)+w(b)}{2}$	if a and b do not share the same kind
$comparison(a, b)$	$\frac{3(w(a)+w(b))}{2}$	if a and b share the same kind but not the label

Kind means in this context edge or vertex. In that case, we think that sharing the same kind but not the same label is the worst case scenario. Those weights avoid having multiples time the same distance between two random paths which will be important during the next step. In fact, because the two graphs do not contain cycles, they have a finite number of paths. We will need to compute all distances between all possible pairs. The more distinct values we have for a distance between two paths, the better it is.

Step 2 : Kruskal kind (bijection on paths)

Among all possible distances we compute between two paths, some of them might be irrelevant. If for some reasons, there is an edge which connects the root to the leave, computing the edit distance between this short path and any other one seems weird. That's why we need to consider a kind of paths association. We need to associate a path from a graph to another from the other graph and the association needs to be a bijection assuming two close paths will have a low distance.

After the first step, we ended up with a matrix containing all possible distances. We convert this matrix into a graph where each path is a vertex, each distance is a weighted edge. We use a kind of Kruskal algorithm to remove heavy weighted edges until we get a kind of bijection :

1. We sort all edges by weight (reverse order).
2. We remove the first ones until we get an injection on both sides : a path from a graph must be associated to only one path.

Basically, some paths from the bigger graph will not be teamed up with another path.

Step 3 : Matching

Now that we have a kind of bijection between paths, it also means we have a series of alignments between paths : one from the first graph, one from the second graph and an alignment between them computed during the step. We build two matrices, one for the edges M_e, one for the vertices M_v defined as follows :

1. $M_e(i, j)$ contains the number of times edge i from graph 1 is associated to edge j from graph 2 among all paths associated by the previous step.
2. $M_v(i, j)$ contains the same for the vertices.

Step 4 : Kruskal kind the return (bijection on edges and vertices

We now have two matrices which contains pretty much the same information as we have in step 2 : each element is the number of times an edge or a vertex was associated with an edge or a vertex of the other graph. We use the same algorithm until we get a kind of bijection between vertices or edges from both matrices.

Step 5 : Meging the two graphs

Once we finalized the previous steps, we know which vertices and edges will be associated with vertices and edges from the other graph. What's left is to add the left over to the picture which is shown by Figure 2.

The main drawback of this algorithm is its speed. It is very time consuming. We need to compute distances between all paths which is ok when graphs are small but very long when graphs are bigger. Many paths share the same beginning and we could certainly avoid wasting time computing edit distances between those paths.

Distance between graphs

We defined a distance between two sequences based on the sum of operations needed to switch from the first sequence to the second one, we can follow the same way here. The alignment we were able to build between two graphs shows insertions, deletions and comparisons of different edges of vertices. By giving a weight of each kind, we can sum them to build the distance we are looking for. We use the same weights we defined to compute the alignment between two paths from both graphs. Let's

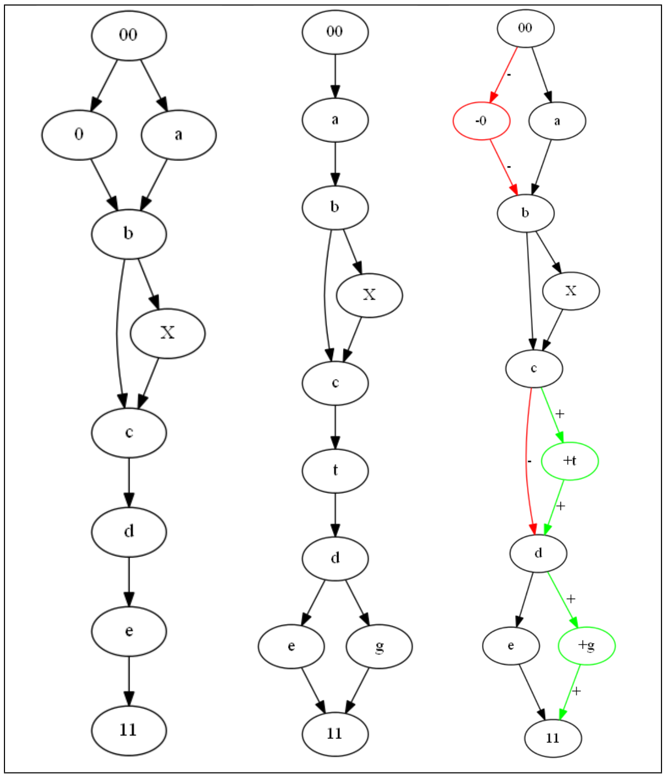

FIGURE 2 : *The graph we get by merging graph from Figure 1. Red and symbol - means deleted from graph 1 and not present in graph 2. Green and symbol + means not present in graph 1 and added in graph 2. The black pieces remains unchanged.*

defined an aligned graph $G = (a, b)$, G is the set of edges and vertices of the final graph, a, b are an edge of a vertex from the first graph for a and from the second graph for b. a or b can be null. We also defined $insertion(a) = comparison(\emptyset, a)$.

$$d(G_1, G_2) = \sum_{\substack{a \in G_1 \cup \emptyset \\ b \in G_2 \cup \emptyset}} comparison(a, b) \mathbb{1}_{\{(a,b) \in G\}} \tag{2}$$

It is obvioulsy symmetric. To proove it verifies $d(G_1, G_2) = 0 \iff G_1 = G_2$, we could proove that every path from G_1 will be associated to itself during the first step. It is not necessarily true because two different paths could share the same sequence of labels. Let's consider the following example :

$$\begin{tikzcd}
begin \arrow[r] \arrow[dr] & 1, a \arrow[r] & 2, b \arrow[r] \arrow[dl] & end \\
& 3, a \arrow[r] & 4, b \arrow[ur] &
\end{tikzcd}$$

This graph contains three paths :

path 1	1,2	ab
path 2	3,4	ab
path 3	1,2,3,4	abab

The matrix distance between paths will give $(x > 0)$:

$$\begin{pmatrix} 0 & \mathbf{0}. & x \\ \mathbf{0}. & 0 & x \\ x & x & \mathbf{0}. \end{pmatrix}$$

The bolded values **0**. represent one possible association between paths which could lead to the possible association between vertices :

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

In that particular case, the algorithm will not return a null distance mostly because while aligning sequences, we do not pay too much attention to the local structure. One edge could be missing from the alignment. We could try to correct that by adding some cost when two vertices do not have the number of input or output edges instead of considering only the labels.

Correction

0.3 Code for the first approach

```
#coding:latin-1


import os, sys, copy, re

if sys.version_info.major >= 3 :
    fileprint = print
else :
    def pprint(*x) : print (x)
    fileprint = pprint

try :
    import importme
except ImportError :
    pass



class Vertex :

    def __init__ (self, nb, label, weight) :
        self.nb     = nb         # kind of id
        self.label  = label      # label
        self.pair   = (None,None)
        self.edges  = { }
        self.predE  = { }
        self.succE  = { }
        self.weight = weight

    def __str__ (self) :
        li = [ "V %s l:%s nbSucc:%d" % (str(self.nb), self.label, len (self.succE)) ]
        if self.pair != (None,None) :
            li.append (   "    (%s,%s)" % (str(self.pair[0]), str (self.pair[1])) )
        if self.edges != None :
            li.extend ( [ "    %s->%s" % (k,str(v)) for k,v in self.edges.items() ] )
        if self.predE != None :
            li.extend ( [ "    prev %s %s" % (k,"") for k,v in self.predE.items() ] )
        return "\n".join(li)

    def isVertex (self) : return True
    def isEdge (self) : return False
    def Label(self) : return self.label
    def __str__(self) : return self.Label()

class Edge :

    def __init__ (self,from_, to, label, weight) :
        self.from_, self.to = from_,to
        self.nb         = from_, to
        self.label      = label
        self.pair       = (None,None)
        self.weight     = weight
        if self.from_ == "00" and self.to == "00" : raise Exception ("should not happen")
        if self.from_ == "11" and self.to == "11" : raise Exception ("should not happen")

    def __str__ (self) :
        li = [ "E %s->%s l:%s" % (str(self.from_), str(self.to), self.label) ]
        if self.pair != (None,None) :
            li.append ("    (%s,%s)" % (str(self.pair[0]), str (self.pair[1])) )
        return "\n".join(li)

    def isVertex (self) : return False
    def isEdge (self) : return True
    def Label(self) : return self.label
    def __str__(self) : return self.Label()



class Graph :

    # graph is a dictionary
    def GetListOfVertices (graph) :
        edges = [ edge[:2] for edge in graph ]
        unique = { }
        for i,j in edges : unique [i] = unique [j] = 1
        vertices = list(unique.keys())
        vertices.sort()
        return vertices
    GetListOfVertices = staticmethod(GetListOfVertices)

    def __init__ (self, edgeList, vertexLabel = { },
                        addLoop = False,
                        weightVertex = 1.,
                        weightEdge = 1.) :
        if type(edgeList) is str :
            self.load_from_file(edgeList, addLoop)
        else :
            self.vertices = { }
            self.edges    = { }
            self.labelBegin = "00"
            self.labelEnd   = "11"
            vid = Graph.GetListOfVertices (edgeList)
            for u in vid :
                self.vertices [ u ] = Vertex (u, vertexLabel.get(u, str(u)), weightVertex)
            for e in edgeList :
                i,j = e[:2]
                l   = "" if len(e) < 3 else e[2]
                self.edges [ i,j ] = Edge (i,j, str(l), weightEdge)
            self.private__init__(addLoop, weightVertex, weightEdge)

    def __getitem__ (self, index) :
        if isinstance (index, str) :
            return self.vertices[index]
        elif isinstance (index, tuple) :
            return self.edges[index]
        else :
            raise Exception("unable to get element " + str(index))

    def load_from_file (self, filename, addLoop) :
        lines = open(filename, "r").readlines ()
        regV = re.compile("\\\"?([a-z0-9_]+)\\\"? *[[]label=\\\"(.*)\\\"[]]")
        regE = re.compile("\\\"?([a-z0-9_]+)\\\"? *-> *\\\"?" + \
                                "([a-z0-9_]+)\\\"? *[[]label=\\\"(.*)\\\"[]]")
        edgeList = []
        vertexLabel = { }
        for line in lines :
            line = line.strip("\r\n ;")
            ed = regE.search(line)
            ve = regV.search(line)
            if ed :
                g = ed.groups()
                edgeList.append ( (g[0], g[1], g[2] ) )
            elif ve :
                g = ve.groups()
                vertexLabel [g[0]] = g[1]
        if len(vertexLabel) == 0 or len(edgeList) == 0 :
            raise Exception("unable to parse file " + filename)
        self.__init__(edgeList, vertexLabel, addLoop)

    def private__init__(self, addLoop, weightVertex, weightEdge) :
        if addLoop :
            for k,v in self.vertices.items () :
                if k != self.labelBegin and k != self.labelEnd :
                    self.edges [k,k] = Edge (k,k, "", weightEdge)
        self.ConnectRootAndLeave(weightVertex, weightEdge)
        self.ComputePredecessor()
        self.ComputeSuccessor()

    def ConnectRootAndLeave (self, weightVertex, weightEdge) :
        order = self.GetOrderVertices()
        roots = [ v for v,k in order.items () if k == 0 ]
        vert = { }
        for o in order : vert [o] = 0
        for k,v in self.edges.items () :
            if k[0] != k[1] :
                vert[k[0]] += 1
        for r in roots :
            if self.labelBegin not in self.vertices :
                self.vertices [ self.labelBegin ] = \
                    Vertex ( self.labelBegin, self.labelBegin, weightVertex)
        if r != self.labelBegin :
            self.edges [ self.labelBegin, r ] = \
                    Edge ( self.labelBegin, r, "", weightEdge )

    leaves = [ k for k,v in vert.items () if v == 0 ]
    for r in leaves:
        if self.labelEnd not in self.vertices :
            self.vertices [ self.labelEnd ] = \
                    Vertex ( self.labelEnd, self.labelEnd, weightVertex)
        if r != self.labelEnd :
            self.edges [ r, self.labelEnd ] = \
                    Edge ( r, self.labelEnd, "", weightEdge )

def GetOrderVertices (self) :
    edges = self.edges
    order = { }
    for k,v in edges.items () :
        order[k[0]] = 0
        order[k[1]] = 0

    modif = 1
    while modif > 0 :
        modif = 0
        for k,v in edges.items () :
            i,j = k
            if i != j and order[j] <= order[i] :
                order [j] = order[i]+1
                modif += 1

    return order

def __str__(self) :
    li = []
    for k,v in self.vertices.items() :
        li.append ( str (v) )
    for k,e in self.edges.items() :
        li.append ( str(e) )
    return "\n".join(li)

def ComputePredecessor (self) :
    pred = { }
    for i,j in self.edges :
        if j not in pred : pred[j] = { }
        pred [j][i,j] = 0
    for k,v in pred.items () :
        for n in v :
            self.vertices[k].predE [n] = self.edges [n]

def ComputeSuccessor (self) :
    succ = { }
    for i,j in self.edges :
        if i not in succ : succ[i] = { }
        succ [i][i,j] = i,j
    for k,v in succ.items () :
        for n in v :
            self.vertices[k].succE[n] = self.edges[n]

def GetMatchingFunctions(self, functionMatchVertices, fonctionMatchEdges,
                            cost = False) :
    if cost :

        if functionMatchVertices == None :
            def tempF1 (v1,v2,g1,g2,w1,w2) :
                if v1 != None and not v1.isVertex() : raise Exception("should be a vertex")
                if v2 != None and not v2.isVertex() : raise Exception("should be a vertex")
                if v1 == None and v2 == None : return 0
                elif v1 == None or v2 == None :
                    return v2.weight*w2 if v1 == None else v1.weight*w1
                else :
                    return 0 if v1.label == v2.label else 0.5*(v1.weight*w1 + v2.weight*w2)
            functionMatchVertices = tempF1

        if fonctionMatchEdges == None :
            def tempF2 (e1,e2,g1,g2,w1,w2) :
                if e1 != None and not e1.isEdge() : raise Exception("should be an edge")
                if e2 != None and not e2.isEdge() : raise Exception("should be an edge")
                if e1 == None and e2 == None : return 0
                elif e1 == None or e2 == None :
                    return e2.weight*w2 if e1 == None else e1.weight*w1
                elif e1.label != e2.label : return 0.5*(e1.weight*w1 + e2.weight*w2)
                else :
                    lab1 = g1.vertices [e1.from_].label == g2.vertices [e2.from_].label
                    lab2 = g1.vertices [e1.to].label == g2.vertices [e2.to].label
                    if lab1 and lab2 : return 0
                    else :  return e1.weight*w1 + e2.weight*w2

            fonctionMatchEdges = tempF2
    else :
        if functionMatchVertices == None :
            functionMatchVertices = lambda v1,v2,g1,g2,w1,w2 : v1.label == v2.label
        if fonctionMatchEdges == None :
            fonctionMatchEdges = lambda e1,e2,g1,g2,w1,w2 : e1.label == e2.label and \
                        (e1.from_ != e1.to or e2.from_ != e2.to) and \
                        (e1.from_ != self.labelBegin or e1.to != self.labelBegin) and \
                        (e1.from_ != self.labelEnd or e1.to != self.labelEnd)
    return functionMatchVertices, fonctionMatchEdges

def CommonPaths (self, graph2,
                 functionMatchVertices = None,
                 fonctionMatchEdges = None,
                 noClean = False) :
    functionMatchVertices, fonctionMatchEdges = \
                    self.GetMatchingFunctions(functionMatchVertices, fonctionMatchEdges)
    g = Graph ( [] )
    vfirst = Vertex ( self.labelBegin, "%s-%s" % (self.labelBegin, self.labelBegin),
                    (self.vertices[self.labelBegin].weight +
                     graph2.vertices[self.labelBegin].weight)/2)
    g.vertices [ self.labelBegin ] = vfirst
    vfirst.pair = self.vertices [ self.labelBegin ], graph2.vertices [ self.labelBegin ]

    modif = 1
    while modif > 0 :
        modif = 0
        add = { }
        for k,v in g.vertices.items () :

            v1,v2 = v.pair
            if len(v.succE) == 0 :
                for e1 in v1.succE :
                    for e2 in v2.succE :
                        oe1 = self.edges[e1]
                        oe2 = graph2.edges[e2]
                        if fonctionMatchEdges(oe1,oe2,self,graph2,1.,1.) :
                            tv1 = self.vertices [oe1.to]
                            tv2 = graph2.vertices [oe2.to]
                            if functionMatchVertices (tv1, tv2, self, graph2, 1., 1.) :
                                # we have a match
                                ii = "%s-%s" % (tv1.nb, tv2.nb)
                                if tv1.nb == self.labelEnd and tv2.nb == self.labelEnd :
                                    ii = self.labelEnd
                                lab = "%s-%s" % (tv1.label, tv2.label) \
                                          if tv1.label != tv2.label else tv1.label
                                tv = Vertex (ii, lab, (tv1.weight+tv2.weight)/2)
                                lab = "%s-%s" % (oe1.label, oe2.label) \
                                          if oe1.label != oe2.label else oe1.label
                                ne = Edge (v.nb, tv.nb,  lab, (oe1.weight + oe2.weight)/2)
                                add [ tv.nb ] = tv
                                g.edges [ ne.from_, ne.to ] = ne
                                ne.pair = oe1,oe2
                                tv.pair = tv1,tv2
                                v.succE [ ne.from_, ne.to ] = ne
                                modif += 1
        for k,v in add.items () :
            g.vertices[k] = v

    if not noClean :
        #g.ConnectRootAndLeave()
        g.ComputePredecessor()
        g.CleanDeadEnds()
    return g

def CleanDeadEnds (self) :
    edgesToKeep = { }
    verticesToKeep = { }
    if self.labelEnd in self.vertices :
        v = self.vertices  [ self.labelEnd ]
        verticesToKeep [v.nb] = False

        modif = 1
        while modif > 0 :
            modif = 0
            add = { }
            for k,v in verticesToKeep.items() :
                if v : continue
                modif += 1
                verticesToKeep [k] = True
                for pred,vv in self.vertices[k].predE.items() :
                    edgesToKeep [ pred ] = True
                    add [ vv.from_ ] = verticesToKeep.get ( vv.from_, False )
            for k,v in add.items () :
                verticesToKeep [k] = v

        remove = {}
        for k in self.vertices :
            if k not in verticesToKeep : remove[k] = True
        for k in remove : del self.vertices[k]
```

```
            remove = {}
            for k in self.edges:
                if k not in edgesToKeep : remove[k] = True
            for k in remove : del self.edges[k]
        else :
            self.vertices = {}
            self.edges = { }

    def EnumerateAllPaths (self, edgesAndVertices, begin = []) :
        if len(self.vertices) > 0 and len(self.edges) > 0 :
            if edgesAndVertices :
                last = begin[-1] if len(begin) > 0 \
                                 else self.vertices [self.labelBegin]
            else :
                last = self.vertices[begin[-1].to] if len(begin) > 0 \
                                       else self.vertices [self.labelBegin]

            if edgesAndVertices and len(begin) == 0 :
                begin = [ last ]

            for ef in last.succE :
                e = self.edges[ef]
                path = copy.copy(begin)
                v = self.vertices[e.to]
                if e.to == e.from_ :
                    # cycle
                    continue
                else :
                    path.append (e)
                    if edgesAndVertices : path.append (v)
                    if v.label == self.labelEnd :
                        yield path
                    else :
                        for p in self.EnumerateAllPaths(edgesAndVertices, path) :
                            yield p

    def EditDistancePath (self, p1, p2, g1, g2,
                          functionMatchVertices = None,
                          fonctionMatchEdges    = None,
                          useMin                = False,
                          debug                 = False) :
        functionMatchVertices, fonctionMatchEdges = \
                 self.GetMatchingFunctions(functionMatchVertices, fonctionMatchEdges, True)
        dist = { (-1,-1) : (0,None,None) }

        w1 = 1.0 / len(p1) if useMin else 1.
        w2 = 1.0 / len(p2) if useMin else 1.

        for i1,eorv1 in enumerate (p1) :
            for i2,eorv2 in enumerate (p2) :
                np = i1,i2
                posit = [ ((i1-1,i2),  (eorv1, None)),
                          ((i1,i2-1),  (None, eorv2)),
                          ((i1-1,i2-1),(eorv1,eorv2)), ]

                if eorv1.isEdge() and eorv2.isEdge() :
                    func = fonctionMatchEdges
                elif eorv1.isVertex() and eorv2.isVertex() :
```

```
                func = functionMatchVertices
            else :
                func = lambda x,y,g1,g2,w1,w2 : \
                            0.5*(x.weight*w1 + y.weight*w2) if x != None and y != None \
                            else (x.weight*w1 if y == None else y.weight*w2)

            for p,co in posit :
                if p in dist :
                    c0 = dist [p][0]
                    c1 = func (co[0], co[1], g1, g2, w1, w2)
                    c  = c0 + c1
                    if np not in dist :
                        dist[np] = (c, p, co, (c0, c1))
                    elif c < dist[np][0] :
                        dist[np] = (c, p, co, (c0, c1))

    last = dist [len(p1)-1, len(p2)-1]
    path = [ ]
    while last[1] != None :
        path.append (last)
        last = dist [ last[1] ]

    path.reverse()

    d = dist [len(p1)-1, len(p2)-1][0]
    if useMin :
        n = min (len(p1), len(p2))
        d = d*1.0 / n if n > 0 else 0
    return d, path

def privateCountLeftRight (self, valuesInList) :
    countLeft = { }
    countRight = { }
    for k,v in valuesInList :
        i,j = v
        if i not in countRight : countRight [i] = { }
        countRight [i][j] = countRight [i].get (j, 0) + 1
        if j not in countLeft : countLeft [j] = { }
        countLeft [j][i] = countLeft [j].get (i, 0) + 1
    return countLeft, countRight

def privateKruskalMatrix (self, matrix, reverse) :
    countLeft, countRight = self.privateCountLeftRight(matrix)
    cleft, cright = len(countLeft), len(countRight)
    cold = len(matrix)
    matrix.sort (reverse = reverse)
    count = max ( max ( [ sum (_.values()) for _ in countRight.values() ] ),
                  max ( [ sum (_.values()) for _ in countLeft.values() ] ) )
    oldcount = count
    while count > 1 :
        k,v = matrix.pop()
        i,j = v
        countRight[i][j] -= 1
        countLeft [j][i] -= 1
        count = max ( max ( [ max (_.values()) for _ in countRight.values() ] ),
                      max ( [ max (_.values()) for _ in countLeft.values() ] ) )

    mini = min(cleft, cright)
    if len(matrix) < mini :
        raise Exception("impossible: the smallest set should get all" + \
              "its element associated to at least one coming from the other set")

def privateStringPathMatching (self, path, skipEdge = False) :
    temp = []
    for p in path :
        u,v = p[2]
        if skipEdge and ( (u != None and u.isEdge()) \
                          or (v != None and v.isEdge()))  :
            continue
        su = "-" if u == None else str(u.nb)
        sv = "-" if v == None else str(v.nb)
        s  = "(%s,%s)" % (su,sv)
        temp.append (s)
    return " ".join(temp)

def DistanceMatchingGraphsPaths (self, graph2,
                 functionMatchVertices = None,
                 fonctionMatchEdges = None,
                 noClean = False,
                 store = None,
                 useMin = True,
                 weightVertex = 1.,
                 weightEdge = 1.) :

    functionMatchVertices, fonctionMatchEdges = \
                    self.GetMatchingFunctions(functionMatchVertices, fonctionMatchEdges, True)

    paths1 = list(self.EnumerateAllPaths (True))
    paths2 = list(graph2.EnumerateAllPaths (True))

    if store != None :
        store ["nbpath1"] = len (paths1)
        store ["nbpath2"] = len (paths2)

    matrix_distance = { }
    for i1,p1 in enumerate(paths1) :
        for i2,p2 in enumerate(paths2) :
            matrix_distance [i1,i2] = self.EditDistancePath (p1,p2, self, graph2,
                        functionMatchVertices, fonctionMatchEdges, useMin = useMin)

    if store != None : store ["matrix_distance"] = matrix_distance
    reduction = [ (v[0], k) for k,v in matrix_distance.items() ]
    if store != None : store ["path_mat1"] = copy.deepcopy(reduction)
    self.privateKruskalMatrix(reduction, False)
    if store != None : store ["path_mat2"] = copy.deepcopy(reduction)

    pairCountEdge = { }
    pairCountVertex = { }
    for k,v in reduction :
        res,path = matrix_distance [v]
        for el in path :
            n1,n2 = el[2]
            if n1 != None and n2 != None :
                if n1.isEdge () and n2.isEdge() :
                    add = n1.nb, n2.nb
                    pairCountEdge[add] = pairCountEdge.get(add,0) + 1
                elif n1.isVertex () and n2.isVertex() :
                    add = n1.nb, n2.nb
                pairCountVertex[add] = pairCountVertex.get(add,0) + 1

    if store != None :
        store["pairCountVertex"] = pairCountVertex
        store["pairCountEdge"] = pairCountEdge

    reductionEdge = [ (v, k) for k,v in pairCountEdge.items() ]
    if store != None : store ["edge_mat1"] = copy.copy (reductionEdge)
    self.privateKruskalMatrix(reductionEdge, True)
    if store != None : store ["edge_mat2"] = copy.copy (reductionEdge)

    reductionVertex = [ (v, k) for k,v in pairCountVertex.items() ]
    if store != None : store ["vertex_mat1"] = copy.copy (reductionVertex)
    self.privateKruskalMatrix(reductionVertex, True)
    if store != None : store ["vertex_mat2"] = copy.copy (reductionVertex)

    countEdgeLeft,   countEdgeRight   = self.privateCountLeftRight(reductionEdge)
    countVertexLeft, countVertexRight = self.privateCountLeftRight(reductionVertex)

    resGraph   = Graph ([])
    doneVertex = { }
    doneEdge   = { }

    for k,v in self.vertices.items() :
        newv = Vertex (v.nb, v.label, weightVertex)
        resGraph.vertices [k] = newv
        if v.nb in countVertexRight :
            ind = list(countVertexRight[v.nb].keys())[0]
            newv.pair = ( v, graph2.vertices[ ind ] )
            doneVertex[ ind ] = newv
            if newv.pair[0].label != newv.pair[1].label :
                newv.label = "%s|%s" % (newv.pair[0].label, newv.pair[1].label)
        else :
            newv.pair = ( v, None )

    for k,v in graph2.vertices.items() :
        if k in doneVertex : continue
        newv = Vertex ("2a.%s" % v.nb, v.label, weightVertex)
        resGraph.vertices [newv.nb] = newv
        newv.pair = ( None, v )

    for k,e in self.edges.items() :
        newe = Edge (e.from_, e.to, e.label, weightEdge)
        resGraph.edges [k] = newe
        if e.nb in countEdgeRight :
            ind = list(countEdgeRight[e.nb].keys())[0]
            newe.pair = ( e, graph2.edges[ ind ] )
            doneEdge[ ind ] = newe
        else :
            newe.pair = ( e, None )

    for k,e in graph2.edges.items() :
        if k in doneEdge : continue
        from_ = list(countVertexLeft[e.from_].keys())[0] if e.from_ in countVertexLeft \
                    else "2a.%s" % e.from_
        to    = list(countVertexLeft[e.to].keys())[0] if e.to in countVertexLeft \
                    else "2a.%s" % e.to
        if from_ not in resGraph.vertices : raise Exception("should not happen " + from_)
        if to not in resGraph.vertices : raise Exception("should not happen " + to)
```

```
            newe = Edge (from_, to, e.label, weightEdge)
            resGraph.edges [newe.nb] = newe
            newe.pair = ( None, e )

        resGraph.ComputePredecessor()
        resGraph.ComputeSuccessor()

        allPaths = list(resGraph.EnumerateAllPaths(True))

        temp = [ sum ( [ 0 if None in _.pair else 1 for _ in p ] ) * 1.0 / len(p) \
                         for p in allPaths ]
        distance = 1.0 - 1.0 * sum(temp) / len(allPaths)

        return distance,resGraph

    def DrawVerticesEdges(self) :
        vertices = []
        edges    = []
        for k,v in self.vertices.items() :
            if v.pair == (None, None) or (v.pair[0] != None and v.pair[1] != None) :
                vertices.append ( (k, v.label) )
            elif v.pair[1] == None :
                vertices.append ( (k, "-" + v.label, "red") )
            elif v.pair[0] == None :
                vertices.append ( (k, "+" + v.label, "green") )
            else :
                raise Exception("?")

        for k,v in self.edges.items() :
            if v.pair == (None, None) or (v.pair[0] != None and v.pair[1] != None) :
                edges.append ( (v.from_, v.to, v.label) )
            elif v.pair[1] == None :
                edges.append ( (v.from_, v.to, "-" + v.label, "red") )
            elif v.pair[0] == None :
                edges.append ( (v.from_, v.to, "+" + v.label, "green") )
            else :
                raise Exception("?")

        return vertices,edges


    def unittest_GraphDistance2 (self, debug = False) :
        graph1 = [ ("a","b"), ("b","c"), ("b","X"), ("X","c"), \
                   ("c","d"), ("d","e"), ("0","b") ]
        graph2 = [ ("a","b"), ("b","c"), ("b","X"), ("X","c"), \
                   ("c","t"), ("t","d"), ("d","e"), ("d","g") ]
        graph1 = Graph(graph1)
        graph2 = Graph(graph2)
        store = { }
        distance,graph = graph1.DistanceMatchingGraphsPaths(graph2,
                            useMin = False, store = store)
        if distance == None : raise Exception("expecting something different from None")
        allPaths = list(graph.EnumerateAllPaths(True))
        if len(allPaths) == 0 : raise Exception("the number of paths should not be null")
        if distance == 0 : raise Exception("expecting a distance > 0")
        vertices,edges = graph.DrawVerticesEdges()
        #GV.drawGraphEdgesVertices (vertices,edges, "unittest_GraphDistance2.png")
        node = graph.vertices["X"]
        if None in node.pair :
            raise Exception("unexpected, this node should be part of the common set")

        if False :
            print ("distance",distance)
            for k,v in store.items() :
                print (k, len(v))
                for _ in v :
                    print ("  ",_)

        vertices,edges = graph1.DrawVerticesEdges()
        #GV.drawGraphEdgesVertices (vertices,edges, "unittest_GraphDistance2_sub1.png")
        vertices,edges = graph2.DrawVerticesEdges()
        #GV.drawGraphEdgesVertices (vertices,edges, "unittest_GraphDistance2_sub2.png")

if __name__ == "__main__" :

    fold = os.path.split(__file__) [0]
    fold = os.path.join(fold, "temp_gaph")
    if not os.path.exists (fold) : os.mkdir (fold)

    GV = importme.import_module ("use_graphviz", whereTo = fold)

    outfile1 = os.path.join(fold, "unittest_GraphDistance4_sub1.png")
    outfile2 = os.path.join(fold, "unittest_GraphDistance4_sub2.png")
    outfilef = os.path.join(fold, "unittest_GraphDistance4_subf.png")
    for f in [ outfile1, outfile2, outfilef ] :
        if os.path.exists (f) : os.remove (f)

    graph1 = [ ("a","b"), ("b","c"), ("b","d"), ("d","e"), \
               ("e","f"), ("b","f"), ("b","g"), ("f", "g"),
               ("a","g"), ("a","g"), ("c","d"), ("d", "g"),
               ("d","h"), ("aa","h"), ("aa","c"), ("f", "h"), ]
    graph2 = copy.deepcopy(graph1) + \
             [ ("h", "m"), ("m", "l"), ("l", "C"), ("C", "r"),
               ("a", "k"), ("k", "l"), ("k", "C"),
               ]

    graph1 = Graph(graph1)
    graph2 = Graph(graph2)

    graph2["C"].label = "c"
    store = { }
    if len(list(graph1.EnumerateAllPaths(True))) != 17 : raise Exception("expecting 17 here")
    if len(list(graph2.EnumerateAllPaths(True))) != 19 : raise Exception("expecting 19 here")

    distance,graph = graph1.DistanceMatchingGraphsPaths(graph2,
                    useMin = False, store = store)

    if graph["h"].Label() != "h":
        raise Exception("we expect this node to be merged in the process")

    if distance == None : raise Exception("expecting something different from None")
    vertices,edges = graph1.DrawVerticesEdges()
    GV.drawGraphEdgesVertices (vertices,edges, outfile1, tempFolder = fold, moduleImport = importme)
    assert os.path.exists (outfile1)

    vertices,edges = graph2.DrawVerticesEdges()
    GV.drawGraphEdgesVertices (vertices,edges, outfile2, tempFolder = fold, moduleImport = importme)
    assert os.path.exists (outfile2)
    vertices,edges = graph.DrawVerticesEdges()
    GV.drawGraphEdgesVertices (vertices,edges, outfilef, tempFolder = fold, moduleImport = importme)
    assert os.path.exists (outfilef)
```