GitHub

<https://github.com/nd7141/influence-maximization/blob/master/IC/ArbitraryP/PMIA.py>

PUDN代码集锦：

http://en.pudn.com/Download/item/id/3393425.html

''' Implementation of PMIA algorithm [1].

[1] -- Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks.

'''

from \_\_future\_\_ import division

import networkx as nx

import math, time

from copy import deepcopy

from runIAC import avgIAC

import multiprocessing, json

from runIAC import avgIAC, runIAC

def updateAP(ap, S, PMIIAv, PMIIA\_MIPv, Ep):

''' Assumption: PMIIAv is a directed tree, which is a subgraph of general G.

PMIIA\_MIPv -- dictionary of MIP from nodes in PMIIA

PMIIAv is rooted at v.

'''

# going from leaves to root

sorted\_MIPs = sorted(PMIIA\_MIPv.iteritems(), key = lambda (\_, MIP): len(MIP), reverse = True)

for u, \_ in sorted\_MIPs:

if u in S:

ap[(u, PMIIAv)] = 1

elif not PMIIAv.in\_edges([u]):

ap[(u, PMIIAv)] = 0

else:

in\_edges = PMIIAv.in\_edges([u], data=True)

prod = 1

for w, \_, edata in in\_edges:

# p = (1 - (1 - Ep[(w, u)])\*\*edata["weight"])

p = Ep[(w,u)]

prod \*= 1 - ap[(w, PMIIAv)]\*p

ap[(u, PMIIAv)] = 1 - prod

def updateAlpha(alpha, v, S, PMIIAv, PMIIA\_MIPv, Ep, ap):

# going from root to leaves

sorted\_MIPs = sorted(PMIIA\_MIPv.iteritems(), key = lambda (\_, MIP): len(MIP))

for u, mip in sorted\_MIPs:

if u == v:

alpha[(PMIIAv, u)] = 1

else:

out\_edges = PMIIAv.out\_edges([u])

assert len(out\_edges) == 1, "node u=%s must have exactly one neighbor, got %s instead" %(u, len(out\_edges))

w = out\_edges[0][1]

if w in S:

alpha[(PMIIAv, u)] = 0

else:

in\_edges = PMIIAv.in\_edges([w], data=True)

prod = 1

for up, \_, edata in in\_edges:

if up != u:

# pp\_upw = 1 - (1 - Ep[(up, w)])\*\*edata["weight"]

pp\_upw = Ep[(up, w)]

prod \*= (1 - ap[(up, PMIIAv)]\*pp\_upw)

# alpha[(PMIIAv, u)] = alpha[(PMIIAv, w)]\*(1 - (1 - Ep[(u,w)])\*\*PMIIAv[u][w]["weight"])\*prod

alpha[(PMIIAv, u)] = alpha[(PMIIAv, w)]\*(Ep[(u,w)])\*prod

def computePMIOA(G, u, theta, S, Ep):

'''

Compute PMIOA -- subgraph of G that's rooted at u.

Uses Dijkstra's algorithm until length of path doesn't exceed -log(theta)

or no more nodes can be reached.

'''

# initialize PMIOA

PMIOA = nx.DiGraph()

PMIOA.add\_node(u)

PMIOA\_MIP = {u: [u]} # MIP(u,v) for v in PMIOA

crossing\_edges = set([out\_edge for out\_edge in G.out\_edges([u]) if out\_edge[1] not in S + [u]])

edge\_weights = dict()

dist = {u: 0} # shortest paths from the root u

# grow PMIOA

while crossing\_edges:

# Dijkstra's greedy criteria

min\_dist = float("Inf")

sorted\_crossing\_edges = sorted(crossing\_edges) # to break ties consistently

for edge in sorted\_crossing\_edges:

if edge not in edge\_weights:

# edge\_weights[edge] = -math.log(1 - (1 - Ep[edge])\*\*G[edge[0]][edge[1]]["weight"])

edge\_weights[edge] = -math.log(Ep[edge])

edge\_weight = edge\_weights[edge]

if dist[edge[0]] + edge\_weight < min\_dist:

min\_dist = dist[edge[0]] + edge\_weight

min\_edge = edge

# check stopping criteria

if min\_dist < -math.log(theta):

dist[min\_edge[1]] = min\_dist

# PMIOA.add\_edge(min\_edge[0], min\_edge[1], {"weight": G[min\_edge[0]][min\_edge[1]]["weight"]})

PMIOA.add\_edge(min\_edge[0], min\_edge[1])

PMIOA\_MIP[min\_edge[1]] = PMIOA\_MIP[min\_edge[0]] + [min\_edge[1]]

# update crossing edges

crossing\_edges.difference\_update(G.in\_edges(min\_edge[1]))

crossing\_edges.update([out\_edge for out\_edge in G.out\_edges(min\_edge[1])

if (out\_edge[1] not in PMIOA) and (out\_edge[1] not in S)])

else:

break

return PMIOA, PMIOA\_MIP

def updateIS(IS, S, u, PMIOA, PMIIA):

for v in PMIOA[u]:

for si in S:

# if seed node is effective and it's blocked by u

# then it becomes ineffective

if (si in PMIIA[v]) and (si not in IS[v]) and (u in PMIIA[v][si]):

IS[v].append(si)

def computePMIIA(G, ISv, v, theta, S, Ep):

# initialize PMIIA

PMIIA = nx.DiGraph()

PMIIA.add\_node(v)

PMIIA\_MIP = {v: [v]} # MIP(u,v) for u in PMIIA

crossing\_edges = set([in\_edge for in\_edge in G.in\_edges([v]) if in\_edge[0] not in ISv + [v]])

edge\_weights = dict()

dist = {v: 0} # shortest paths from the root u

# grow PMIIA

while crossing\_edges:

# Dijkstra's greedy criteria

min\_dist = float("Inf")

sorted\_crossing\_edges = sorted(crossing\_edges) # to break ties consistently

for edge in sorted\_crossing\_edges:

if edge not in edge\_weights:

# edge\_weights[edge] = -math.log(1 - (1 - Ep[edge])\*\*G[edge[0]][edge[1]]["weight"])

edge\_weights[edge] = -math.log(Ep[edge])

edge\_weight = edge\_weights[edge]

if dist[edge[1]] + edge\_weight < min\_dist:

min\_dist = dist[edge[1]] + edge\_weight

min\_edge = edge

# check stopping criteria

# print min\_edge, ':', min\_dist, '-->', -math.log(theta)

if min\_dist < -math.log(theta):

dist[min\_edge[0]] = min\_dist

# PMIIA.add\_edge(min\_edge[0], min\_edge[1], {"weight": G[min\_edge[0]][min\_edge[1]]["weight"]})

PMIIA.add\_edge(min\_edge[0], min\_edge[1])

PMIIA\_MIP[min\_edge[0]] = PMIIA\_MIP[min\_edge[1]] + [min\_edge[0]]

# update crossing edges

crossing\_edges.difference\_update(G.out\_edges(min\_edge[0]))

if min\_edge[0] not in S:

crossing\_edges.update([in\_edge for in\_edge in G.in\_edges(min\_edge[0])

if (in\_edge[0] not in PMIIA) and (in\_edge[0] not in ISv)])

else:

break

return PMIIA, PMIIA\_MIP

def PMIA(G, k, theta, Ep):

start = time.time()

# initialization

S = []

IncInf = dict(zip(G.nodes(), [0]\*len(G)))

PMIIA = dict() # node to tree

PMIOA = dict()

PMIIA\_MIP = dict() # node to MIPs (dict)

PMIOA\_MIP = dict()

ap = dict()

alpha = dict()

IS = dict()

for v in G:

IS[v] = []

PMIIA[v], PMIIA\_MIP[v] = computePMIIA(G, IS[v], v, theta, S, Ep)

for u in PMIIA[v]:

ap[(u, PMIIA[v])] = 0 # ap of u node in PMIIA[v]

updateAlpha(alpha, v, S, PMIIA[v], PMIIA\_MIP[v], Ep, ap)

for u in PMIIA[v]:

IncInf[u] += alpha[(PMIIA[v], u)]\*(1 - ap[(u, PMIIA[v])])

print 'Finished initialization'

print time.time() - start

# main loop

for i in range(k):

u, \_ = max(IncInf.iteritems(), key = lambda (dk, dv): dv)

# print i+1, "node:", u, "-->", IncInf[u]

IncInf.pop(u) # exclude node u for next iterations

PMIOA[u], PMIOA\_MIP[u] = computePMIOA(G, u, theta, S, Ep)

for v in PMIOA[u]:

for w in PMIIA[v]:

if w not in S + [u]:

IncInf[w] -= alpha[(PMIIA[v],w)]\*(1 - ap[(w, PMIIA[v])])

updateIS(IS, S, u, PMIOA\_MIP, PMIIA\_MIP)

S.append(u)

for v in PMIOA[u]:

if v != u:

PMIIA[v], PMIIA\_MIP[v] = computePMIIA(G, IS[v], v, theta, S, Ep)

updateAP(ap, S, PMIIA[v], PMIIA\_MIP[v], Ep)

updateAlpha(alpha, v, S, PMIIA[v], PMIIA\_MIP[v], Ep, ap)

# add new incremental influence

for w in PMIIA[v]:

if w not in S:

IncInf[w] += alpha[(PMIIA[v], w)]\*(1 - ap[(w, PMIIA[v])])

return S

def getCoverage((G, S, Ep)):

return len(runIAC(G, S, Ep))

if \_\_name\_\_ == "\_\_main\_\_":

import time

start = time.time()

model = "Categories"

if model == "MultiValency":

ep\_model = "range"

elif model == "Random":

ep\_model = "random"

elif model == "Categories":

ep\_model = "degree"

dataset = "gnu09"

G = nx.read\_gpickle("../../graphs/%s.gpickle" %dataset)

print 'Read graph G'

print time.time() - start

Ep = dict()

with open("Ep\_%s\_%s1.txt" %(dataset, ep\_model)) as f:

for line in f:

data = line.split()

Ep[(int(data[0]), int(data[1]))] = float(data[2])

ALGO\_NAME = "PMIA"

FOLDER = "Data4InfMax"

SEEDS\_FOLDER = "Seeds"

TIME\_FOLDER = "Time"

DROPBOX\_FOLDER = "/home/sergey/Dropbox/Influence Maximization"

seeds\_filename = SEEDS\_FOLDER + "/%s\_%s\_%s\_%s.txt" %(SEEDS\_FOLDER, ALGO\_NAME, dataset, model)

time\_filename = TIME\_FOLDER + "/%s\_%s\_%s\_%s.txt" %(TIME\_FOLDER, ALGO\_NAME, dataset, model)

theta = 1.0/20

pool = None

I = 1000

l2c = [[0, 0]]

# open file for writing output

seeds\_file = open(seeds\_filename, "a+")

time\_file = open(time\_filename, "a+")

dbox\_seeds\_file = open("%/%", DROPBOX\_FOLDER, seeds\_filename, "a+")

dbox\_time\_file = open("%/%", DROPBOX\_FOLDER, time\_filename, "a+")

for length in range(1, 250, 5):

time2length = time.time()

print "Start finding solution for length = %s" %length

time2S = time.time()

S = PMIA(G, length, theta, Ep)

time2complete = time.time() - time2S

print >>time\_file, (time2complete)

print >>dbox\_time\_file, (time2complete)

print 'Finish finding S in %s sec...' %(time2complete)

print 'Writing S to files...'

print >>seeds\_filename, json.dumps(S)

print >>dbox\_seeds\_file, json.dumps(S)

# print "Start calculating coverage..."

# # def map\_AvgIAC (it):

# # return avgIAC(G, S, Ep, I)

# if pool == None:

# pool = multiprocessing.Pool(processes=None)

# avg\_size = 0

# time2avg = time.time()

# T = pool.map(getCoverage, ((G, S, Ep) for i in range(I)))

# avg\_size = sum(T)/len(T)

# print 'Average coverage of %s nodes is %s' %(length, avg\_size)

# print 'Finished averaging seed set size in', time.time() - time2avg

# print >>ftime, "%s %s" %(length, time.time() - time2S)

#

# l2c.append([length, avg\_size])

# with open('plotdata/plot' + FILENAME, 'w+') as fresults:

# json.dump(l2c, fresults)

# with open(DROPBOX + 'plotdata/plot' + FILENAME, 'w+') as fp:

# json.dump(l2c, fp)

print 'Total time for length = %s: %s sec' %(length, time.time() - time2length)

print '----------------------------------------------'

seeds\_file.close()

dbox\_seeds\_file.close()

time\_file.close()

dbox\_time\_file.close()

print 'Total time: %s' %(time.time() - start)