15-826 Project Phase2

Silun Wang
silunw@andrew.cmu.edu
Yuwei Zhang
yuweiz1@andrew.cmu.edu

```
import argparse

from gm_params import *
from gm_sql import *
from math import sqrt
import os
import time

db_conn = None;
# 1 for non-clustering src, 2 for clustering src, 3 for composite index
index_type = 3


# Convert directed to undirected + remove multiple edges
def gm_to_undirected(rm_multiple = True):
    cur = db_conn.cursor()
    gm_sql_table_drop_create(db_conn, GM_TABLE_UNDIRECT, "src_id integer, dst_id
        integer, weight real")

    if rm_multiple:
        stmt = "INSERT INTO %s " % (GM_TABLE_UNDIRECT) + \
                   " SELECT src_id, dst_id, AVG(weight) FROM " + \
                   " (SELECT src_id, dst_id, weight FROM %s " % (GM_TABLE) + \
                   " UNION ALL" + \
                   " SELECT dst_id \"src_id\", src_id \"dst_id\", weight FROM %s)
                       \"TAB\"" % (GM_TABLE) + \
                   " GROUP BY src_id, dst_id"
    else:
        stmt = "INSERT INTO %s " % (GM_TABLE_UNDIRECT) + \
                   " (SELECT src_id, dst_id, weight FROM %s " % (GM_TABLE) + \
                   " UNION ALL" + \
                   " SELECT dst_id \"src_id\", src_id \"dst_id\", weight FROM %s) " %
                       (GM_TABLE)


    cur.execute(stmt)
    db_conn.commit()

    cur.close()

def gm_create_node_table ():
    cur = db_conn.cursor()

    gm_sql_table_drop_create(db_conn, GM_NODES, "node_id integer")

    cur.execute ("INSERT INTO %s" % GM_NODES +
                             " SELECT DISTINCT(src_id) FROM %s" % GM_TABLE_UNDIRECT)
```

```
    db_conn.commit()

    cur.close()

def gm_save_tables (dest_dir, belief):
    print "Saving tables..."
    # gm_sql_save_table_to_file(db_conn, GM_KCORE, "node_id, kvalue", \
                            # os.path.join(dest_dir,"kcore.csv"), ",");
    gm_sql_save_table_to_file(db_conn, "GM_CORE_VALUES", "node_id, kvalue", \
                            os.path.join(dest_dir,"kcore.csv"), ",");

    gm_sql_save_table_to_file(db_conn, GM_DEGREE_DISTRIBUTION, "degree, count", \
                            os.path.join(dest_dir,"degreedist.csv"), ",");
    gm_sql_save_table_to_file(db_conn, GM_INDEGREE_DISTRIBUTION, "degree, count", \
                            os.path.join(dest_dir,"indegreedist.csv"), ",");
    gm_sql_save_table_to_file(db_conn, GM_OUTDEGREE_DISTRIBUTION, "degree, count", \
                            os.path.join(dest_dir,"outdegreedist.csv"), ",");

    gm_sql_save_table_to_file(db_conn, GM_NODE_DEGREES, "node_id, in_degree,
        out_degree", \
                            os.path.join(dest_dir,"degree.csv"), ",");

    gm_sql_save_table_to_file(db_conn, GM_PAGERANK, "node_id, page_rank", \
                            os.path.join(dest_dir,"pagerank.csv"), ",");


    gm_sql_save_table_to_file(db_conn, GM_CON_COMP, "node_id, component_id", \
                            os.path.join(dest_dir,"conncomp.csv"), ",");

    gm_sql_save_table_to_file(db_conn, GM_RADIUS, "node_id, radius", \
                            os.path.join(dest_dir,"radius.csv"), ",");

    if (belief):
        gm_sql_save_table_to_file(db_conn, GM_BELIEF, "node_id, belief", \
                            os.path.join(dest_dir,"belief.csv"), ",");

    gm_sql_save_table_to_file(db_conn, GM_EIG_VALUES, "id, value", \
                            os.path.join(dest_dir,"eigval.csv"), ",");

    gm_sql_save_table_to_file(db_conn, GM_EIG_VECTORS, "row_id, col_id, value", \
                            os.path.join(dest_dir,"eigvec.csv"), ",");


#Project Tasks

def gm_kcore ():
    global index_type
    cur = db_conn.cursor()

    # Create the tables
    gm_sql_table_drop_create(db_conn, "GM_NODE", "node_id integer, inout_degree integer")
    gm_sql_table_drop_create(db_conn, "GM_EDGES", "src_id integer, dst_id integer")
    gm_sql_table_drop_create(db_conn, "GM_CORE_VALUES", "node_id integer, kvalue
        integer")
```

```
    # Create the tables & indices
    cur.execute ("INSERT INTO GM_EDGES select src_id, dst_id from %s" %GM_TABLE )
    cur.execute ("INSERT INTO GM_NODE select src_id, count(*) from GM_EDGES group by
        src_id")
    cur.execute ("CREATE index node_index on GM_NODE (node_id)")
    cur.execute ("CREATE index degree_index on GM_NODE (inout_degree)")
    if index_type == 1:
        # non-clustering
        cur.execute("create index kcore_src on GM_EDGES (src_id)")
    elif index_type == 2:
        # clustering
        cur.execute("create index kcore_src on GM_EDGES (src_id)")
        cur.execute("CLUSTER GM_EDGES USING kcore_src")
    elif index_type == 3:
        # composite
        cur.execute("create index kcore_compo on GM_EDGES (src_id, dst_id)")
        cur.execute ("cluster GM_EDGES using kcore_compo")

    cur.execute ("select count(*) from GM_NODE")

    num = cur.fetchone()[0]
    print num

    # shaving
    k = 0
    for i in xrange(0, num):
        cur.execute ("select node_id, inout_degree from GM_NODE where inout_degree =
            (select min(inout_degree) from GM_NODE) limit 1")

        node, degree = cur.fetchone()
        k = max(k, degree)

        cur.execute ("update GM_NODE set inout_degree = inout_degree-1 where node_id in
            (select dst_id from GM_EDGES where src_id = %d )" % node)
        cur.execute ("INSERT INTO GM_CORE_VALUES values (%d, %d)" %(node, k))
        cur.execute ("delete from GM_NODE where node_id = %d" %(node))


    print "The degeneracy value of the graph is: " + str(k)

    cur.execute ("select * from GM_CORE_VALUES")
    output_file = file('core.txt', 'w')
    res = cur.fetchall()
    for line in res:
        output_file.write(str(line[0]) + ',' + str(line[1]) + '\n')

    db_conn.commit()
    cur.close()

#Task 0: kcore
def gm_kcore_my ():
    # compute coreness of each node
    print "Computing kcore..."

    cur = db_conn.cursor()
```

silunw@andrew.cmu.edu
yuweiz1@andrew.cmu.edu

```
GM_TABLE_DUP = "GM_TABLE_DUP"
GM_KCORE_TMP = "GM_KCORE_TMP"
gm_sql_table_drop_create(db_conn, GM_KCORE, "node_id integer, \
                            coreness integer")
gm_sql_table_drop_create(db_conn, GM_TABLE_DUP, "src_id integer, dst_id integer")

cur.execute("insert into %s" % GM_TABLE_DUP + " select src_id, dst_id from %s"
    %GM_TABLE)
db_conn.commit()

cur.execute("create index on %s (src_id)" % GM_TABLE_DUP)
db_conn.commit()

k = 1
while True:
    # each time we pick out elements with less than k neighbors and output them with
        coreness k
    # delete from the original table whose neighbor are those elements
    # if we get no such elements, we increase k
    # until we get no elements left in the orginal table

    gm_sql_table_drop_create(db_conn, GM_KCORE_TMP, "src_id integer, \
                            neighbor integer")
    cur.execute ("insert into %s" % GM_KCORE_TMP +
                " select src_id, count(*) as neighbor from %s" % GM_TABLE_DUP +
                " group by src_id having count(*) <= %d" % k)
    cur.execute("create index on %s (src_id)" % GM_KCORE_TMP)
    db_conn.commit()
    cur.execute("select count(*) from %s" % GM_KCORE_TMP)
    val = cur.fetchone()[0]
    if val == 0:
        k += 1
        continue

    cur.execute ("INSERT INTO %s" % GM_KCORE +
                            " SELECT src_id , %d" % k + " as coreness from %s"
                                %GM_KCORE_TMP)
    db_conn.commit()

    cur.execute("delete from %s"%GM_TABLE_DUP + " where src_id in (select src_id from
        %s)"%GM_KCORE_TMP)
    cur.execute("delete from %s"%GM_TABLE_DUP + " where dst_id in (select src_id from
        %s)"%GM_KCORE_TMP)
    db_conn.commit()

    cur.execute("select count(*) from %s"%GM_TABLE_DUP)
    val = cur.fetchone()[0]
    if val == 0:
        break

gm_sql_table_drop(db_conn, GM_TABLE_DUP)
gm_sql_table_drop(db_conn, GM_KCORE_TMP)

cur.close()
```

silunw@andrew.cmu.edu
yuweiz1@andrew.cmu.edu

```python
#Task 1: Degree distribution
#------------------------------------------------------------------------------#
def gm_node_degrees ():
    cur = db_conn.cursor()

    # Create Table to store node degrees
    # If the graph is undirected, all the degree values will be the same
    print "Computing Node degrees..."

    gm_sql_table_drop_create(db_conn, GM_NODE_DEGREES, "node_id integer, \
                             in_degree integer, out_degree integer")

    cur.execute ("INSERT INTO %s" % GM_NODE_DEGREES +
                 " SELECT node_id, SUM(in_degree) \"in_degree\",
                     SUM(out_degree) \"out_degree\" FROM " +
                 " (SELECT dst_id \"node_id\", count(*) \"in_degree\", \
                   0 \"out_degree\" FROM %s" % GM_TABLE +
                 " GROUP BY dst_id" +
                 " UNION ALL" +
                 " SELECT src_id \"node_id\", 0 \"in_degree\", \
                   count(*) \"out_degree\" FROM %s" % GM_TABLE +
                 " GROUP BY src_id) \"TAB\" " +
                 " GROUP BY node_id")
    db_conn.commit()

    cur.close()


# Degree distribution
def gm_degree_distribution (undirected):

    cur = db_conn.cursor()
    print "Computing Degree distribution of the nodes..."

    gm_sql_table_drop_create(db_conn, GM_DEGREE_DISTRIBUTION, "degree integer, count
        integer")
    gm_sql_table_drop_create(db_conn, GM_INDEGREE_DISTRIBUTION, "degree integer, count
        integer")
    gm_sql_table_drop_create(db_conn, GM_OUTDEGREE_DISTRIBUTION, "degree integer, count
        integer")

    cur.execute ("INSERT INTO %s" % GM_INDEGREE_DISTRIBUTION +
                " SELECT in_degree \"degree\", count(*) FROM %s" %
                    (GM_NODE_DEGREES) +
                " GROUP BY in_degree");

    cur.execute ("INSERT INTO %s" % GM_OUTDEGREE_DISTRIBUTION +
                " SELECT out_degree \"degree\", count(*) FROM %s" %
                    (GM_NODE_DEGREES) +
                " GROUP BY out_degree");

    if (undirected):
        # Degree distribution is same as in/out degree distribution for undirected graphs
        cur.execute ("INSERT INTO %s" % GM_DEGREE_DISTRIBUTION +
                " SELECT * FROM %s" % GM_INDEGREE_DISTRIBUTION);
```

```
    else:
        cur.execute ("INSERT INTO %s" % GM_DEGREE_DISTRIBUTION +
                           " SELECT in_degree+out_degree \"degree\", count(*) FROM %s" %
                               (GM_NODE_DEGREES) +
                           " GROUP BY in_degree+out_degree");

    cur.execute("SELECT * FROM %s ORDER BY degree" % GM_DEGREE_DISTRIBUTION)
    output_file = file('degree.txt', 'w')
    res = cur.fetchall()
    for line in res:
        output_file.write(str(line[0]) + ',' + str(line[1]) + '\n')

    db_conn.commit()
    cur.close()

# Task 2: PageRank
# ------------------------------------------------------------------------------ #
def gm_pagerank (num_nodes, max_iterations = gm_param_pr_max_iter, \
                  stop_threshold = gm_param_pr_thres, damping_factor =
                      gm_param_pr_damping):
    global index_type
    offset_table = "GM_PR_OFFSET"
    next_table = "GM_PR_NEXT"
    norm_table = "GM_PR_NORM"

    cur = db_conn.cursor();
    print "Computing PageRanks..."

    gm_sql_table_drop_create(db_conn, norm_table,"src_id integer, dst_id integer, weight
        double precision")
    if index_type == 1:
        # non-clustering
        cur.execute("create index pr_src on %s (src_id)" % norm_table)
    elif index_type == 2:
        # clustering
        cur.execute("create index pr_src on %s (src_id)" % norm_table)
        cur.execute("CLUSTER %s USING pr_src" % norm_table)
    elif index_type == 3:
        # composite
        cur.execute("create index pr_compo on %s (src_id, dst_id)" % norm_table)
        cur.execute("CLUSTER %s USING pr_compo" % norm_table)

    # Create normalized weighted table
    cur.execute("INSERT INTO %s " % norm_table +
           " SELECT src_id, dst_id, weight/weight_sm \"weight\" FROM %s \"TAB1\", " %
               (GM_TABLE) +
           " (SELECT src_id \"node_id\", sum(weight) \"weight_sm\" FROM %s GROUP BY
               src_id) \"TAB2\" " % (GM_TABLE) +
           " WHERE \"TAB1\".src_id = \"TAB2\".node_id")
    db_conn.commit();

    # Create PageRank Table and initialize to 1/n
    gm_sql_create_and_insert(db_conn, GM_PAGERANK, GM_NODES, \
                            "node_id integer, page_rank double precision default %s" %
                                (1.0/num_nodes), \
```

```
                             "node_id", "node_id")

    # Create offset table and initialize to 1-c/n
    gm_sql_create_and_insert(db_conn, offset_table, GM_NODES, \
                             "node_id integer, page_rank double precision default %s" %
                                 ((1.0-damping_factor)/num_nodes), \
                             "node_id", "node_id")
    num_iterations = 0
    while True:
        # Create Table to store the next pageRank
        gm_sql_table_drop_create(db_conn, next_table,"node_id integer, page_rank double
            precision")

        # Compute Next PageRank
        cur.execute ("INSERT INTO %s " % next_table +
                                " SELECT node_id, SUM(page_rank) FROM (" +
                                " SELECT dst_id \"node_id\", SUM(%s*weight*page_rank)
                                    \"page_rank\" " % damping_factor +
                                " FROM %s, %s" % (norm_table, GM_PAGERANK) +
                                " WHERE src_id = node_id GROUP BY dst_id" +
                                " UNION ALL" +
                                " SELECT node_id, page_rank * val \"page_rank\" " +
                                " FROM %s, (SELECT SUM(page_rank) \"val\" FROM %s)
                                    \"PRSUM\" " % (offset_table, GM_PAGERANK) +
                                " ) \"TAB\" GROUP BY node_id" )

        db_conn.commit()

        diff = gm_sql_vect_diff(db_conn, GM_PAGERANK, next_table, \
                                "node_id", "node_id", "page_rank", "page_rank")

        # Copy the new page rank to the page rank table
        gm_sql_create_and_insert(db_conn, GM_PAGERANK, next_table, \
                                    "node_id integer, page_rank double precision", \
                                    "node_id, page_rank", "node_id, page_rank")

        num_iterations = num_iterations + 1
        print "Iteration = %d, Error = %f" % (num_iterations, diff)

        if (diff<=stop_threshold or num_iterations>=max_iterations):
            break

    # Drop temp tables
    gm_sql_table_drop(db_conn, offset_table)
    gm_sql_table_drop(db_conn, next_table)
    gm_sql_table_drop(db_conn, norm_table)

    cur.close()

# Task 3: Weakly Connected Components
#------------------------------------------------------------#
def gm_connected_components (num_nodes):
    temp_table = "GM_CC_TEMP"
    cur = db_conn.cursor()
    print 'Computing Weakly Connected Components...'
```

```
    # Create CC table and initialize component id to node id
    gm_sql_create_and_insert(db_conn, GM_CON_COMP, GM_NODES, \
                            "node_id integer, component_id integer", \
                            "node_id, component_id", "node_id, node_id")

    while True:
        gm_sql_table_drop_create(db_conn, temp_table,"node_id integer, component_id
            integer")

        # Set component id as the min{component ids of neighbours, node's componet id}
        cur.execute("INSERT INTO %s " % temp_table +
                        " SELECT node_id, MIN(component_id) \"component_id\" FROM (" +
                            " SELECT src_id \"node_id\", MIN(component_id)
                                \"component_id\" FROM %s, %s" %
                                (GM_TABLE_UNDIRECT,GM_CON_COMP) +
                            " WHERE dst_id = node_id GROUP BY src_id" +
                            " UNION" +
                            " SELECT * FROM %s" % GM_CON_COMP +
                        " ) \"T\" GROUP BY node_id")
        db_conn.commit()

        diff = gm_sql_vect_diff(db_conn, GM_CON_COMP, temp_table, "node_id", "node_id",
            "component_id", "component_id")

        # Copy the new component ids to the component id table
        gm_sql_create_and_insert(db_conn, GM_CON_COMP, temp_table, \
                                "node_id integer, component_id integer", \
                                "node_id, component_id", "node_id, component_id")

        print "Error = " + str(diff)
        # Check whether the component ids has converged
        if (diff == 0):
            print "Component IDs has converged"
            break

    cur.execute ("SELECT count(distinct component_id) FROM %s" % GM_CON_COMP)
    num_components = cur.fetchone()[0]

    print "Number of Components =", num_components
    cur.close()

    # Drop temp tables
    gm_sql_table_drop(db_conn, temp_table)

# Task 4: Radius of every node
#-------------------------------------------------------------#
def gm_all_radius (num_nodes, max_iter = gm_param_radius_max_iter):

    hop_table = "GM_RD_HOP"
    max_hop_ngh = "GM_RD_MAX_HOP_NGH"

    cur = db_conn.cursor()
    print 'Computing radius of every node...'
```

```
    # initialize hop 0 table's hash
    gm_sql_create_and_insert(db_conn, hop_table+"0", GM_NODES, \
                             "node_id integer, hash integer", \
                             "node_id, hash", "node_id,
                                 (((node_id%%%s+1)#(node_id%%%s))+1)/2" % (num_nodes,
                                 num_nodes))

    for cur_hop in range(1,max_iter+1):
        print "Hop number : " + str(cur_hop)

        # create ith hop table
        cur_hop_table = hop_table+str(cur_hop)
        prev_hop_table = hop_table+str(cur_hop-1)
        gm_sql_table_drop_create(db_conn, cur_hop_table,"node_id integer, hash integer")
        cur.execute("INSERT INTO %s " % cur_hop_table +
                          " SELECT node_id, bit_or(hash) FROM ( " +
                          " SELECT src_id \"node_id\", bit_or(hash) \"hash\" " +
                          " FROM %s,%s" % (GM_TABLE_UNDIRECT, prev_hop_table) +
                          " WHERE dst_id = node_id GROUP BY src_id " +
                          " UNION ALL" +
                          " SELECT * FROM %s ) \"TAB\" GROUP BY node_id" %
                              (prev_hop_table))
        db_conn.commit()

        # Check convergence
        diff = gm_sql_vect_diff(db_conn, cur_hop_table, prev_hop_table, "node_id",
            "node_id", "hash", "hash")

        print "Current Error = " + str(diff)
        if (diff==0):
            print "Convergence acheived"
            break

    nghbourhd_func = "2^(floor(log(2,hash)+1))/0.77351"
    gm_sql_create_and_insert(db_conn, max_hop_ngh, cur_hop_table, \
                             "id integer, value double precision", \
                             "id, value", "node_id, %s" % (nghbourhd_func))



    gm_sql_table_drop_create(db_conn, GM_RADIUS,"node_id integer, radius integer")

    for i in range(0,cur_hop+1):
        print "Getting nodes with eff. radius " + str(i)
        # effective radius is the hop at which neighbour fucntion value exceeds
        # 0.9 * the value at max hop
        cur.execute("INSERT INTO %s" % GM_RADIUS +
                      " SELECT node_id, %s \"radius\" FROM %s, %s " % (i,
                          hop_table+str(i), max_hop_ngh) +
                      " WHERE node_id = id AND %s>=0.9*value " % (nghbourhd_func))

        db_conn.commit()
        cur.execute("DELETE FROM %s WHERE id IN(SELECT node_id FROM %s)" % (max_hop_ngh,
            GM_RADIUS));
        db_conn.commit()
```

```
    cur.execute ("SELECT max(radius) FROM %s" % GM_RADIUS)
    max_radius = cur.fetchone()[0]
    print "Maximum effective radius =", max_radius

    # drop temp tables
    gm_sql_table_drop(db_conn, max_hop_ngh)
    for i in range(0,cur_hop+1):
        gm_sql_table_drop(db_conn, hop_table+str(i))


    cur.close()

# Task 5: Eigen values
# ------------------------------------------------------------------------ #
# The adjacency matrix should be symmetric

def gm_eigen_QR_decompose(T, n, Q, R):
    G = "GM_QR_DECOMPOSE_GIVENS"
    temp_table = "GM_QR_DECOMPOSE_TEMP"
    I = "GM_QR_DECOMPOSE_IDENTITY"

    cur = db_conn.cursor();

    gm_sql_table_drop_create(db_conn, R,"row_id integer, col_id integer, value double
        precision")

    # Initialize R = T
    cur.execute("INSERT INTO %s" % (R) + " SELECT * FROM %s" % (T))
    db_conn.commit()


    for i in range(1,n):
        # Compute the givens matrix
        cur.execute("SELECT value FROM %s " % (R) +
                       "WHERE col_id = %s AND row_id >= %s ORDER BY row_id" % (str(i),
                           str(i)) )

        c = cur.fetchone()[0]
        s = cur.fetchone()[0]
        r = sqrt(c*c + s*s)
        c = c/r
        s = -s/r

        gm_sql_table_drop_create(db_conn, G,"row_id integer, col_id integer, value double
            precision")
        cur.execute("INSERT INTO %s" % (G) + " SELECT * FROM %s" % (I))
        cur.execute('UPDATE %s' % (G) + ' SET value = %s WHERE row_id = %s AND col_id =
            %s' %(str(c),str(i),str(i)))
        cur.execute('UPDATE %s' % (G) + ' SET value = %s WHERE row_id = %s AND col_id =
            %s' %(str(c),str(i+1),str(i+1)))
        cur.execute('INSERT INTO %s' % (G) + ' VALUES (%s,%s,%s)'
            %(str(i),str(i+1),str(-s)))
        cur.execute('INSERT INTO %s' % (G) + ' VALUES (%s,%s,%s)'
```

```
            %(str(i+1),str(i),str(s)))
        db_conn.commit()

        # Compute Q
        if i == 1:
            # insert G*
            gm_sql_table_drop_create(db_conn, Q,"row_id integer, col_id integer, value
                double precision")
            cur.execute("INSERT INTO %s" % (Q) + " SELECT \"col_id\" row_id, \"row_id\"
                col_id, value FROM %s" % (G))
        else:
            gm_sql_table_drop_create(db_conn, temp_table,"row_id integer, col_id integer,
                value double precision")
            gm_sql_mat_mat_multiply (db_conn, Q, G, temp_table, "col_id", "col_id",
                "value", "value",
                                        "value", "row_id", "row_id", "row_id", "row_id")
            gm_sql_table_drop_create(db_conn, Q,"row_id integer, col_id integer, value
                double precision")
            cur.execute("INSERT INTO %s" % (Q) + " SELECT * FROM %s" % (temp_table))

        db_conn.commit()
        # Compute R
        gm_sql_table_drop_create(db_conn, temp_table,"row_id integer, col_id integer,
            value double precision")
        gm_sql_mat_mat_multiply (db_conn, G, R, temp_table, "col_id", "row_id", "value",
            "value",
                                        "value", "row_id", "col_id", "row_id", "col_id")
        gm_sql_table_drop_create(db_conn, R,"row_id integer, col_id integer, value double
            precision")
        cur.execute("INSERT INTO %s" % (R) + " SELECT * FROM %s" % (temp_table))

        db_conn.commit()

    cur.close()
    # Drop temp tables
    gm_sql_table_drop(db_conn, G)

    gm_sql_table_drop(db_conn, temp_table)



def gm_eigen_QR_iterate(T, n, EVal, EVec, steps, err):

    Q = "GM_QR_Q"
    R = "GM_QR_R"
    temp_table = "GM_QR_TEMP"
    I = "GM_QR_DECOMPOSE_IDENTITY"
    print 'Performing QR Algorithm. Max Iters=%s, Stop threshold=%s' % (steps, err)

    cur = db_conn.cursor();

    gm_sql_table_drop_create(db_conn, EVal,"row_id integer, col_id integer, value double
        precision")
    gm_sql_table_drop_create(db_conn, EVec,"row_id integer, col_id integer, value double
        precision")
```

silunw@andrew.cmu.edu
yuweiz1@andrew.cmu.edu

```
gm_sql_table_drop_create(db_conn, I,"row_id integer, col_id integer, value double
    precision")
gm_sql_load_table(db_conn, I, [str(i) + " " + str(i) + " " + str(1) for i in
    range(1,n+1)])

cur.execute("INSERT INTO %s" % (EVal) + " SELECT * FROM %s" % (T))
db_conn.commit()

for i in range(1,steps+1):

    try:
        gm_eigen_QR_decompose(EVal, n, Q, R)
    except psycopg2.DataError:
        db_conn.commit()
        break

    gm_sql_table_drop_create(db_conn, EVal,"row_id integer, col_id integer, value
        double precision")

    # Set EVal as RQ
    gm_sql_mat_mat_multiply (db_conn, R, Q, EVal, "col_id", "row_id", "value",
        "value",
                                "value", "row_id", "col_id", "row_id", "col_id")

    if i==1:
        # Copy Q to EVec
        cur.execute("INSERT INTO %s" % (EVec) + " SELECT * FROM %s" % (Q))
        db_conn.commit()
    else:
        # Set EVec = EVec * Q
        gm_sql_table_drop_create(db_conn, temp_table,"row_id integer, col_id integer,
            value double precision")
        gm_sql_mat_mat_multiply (db_conn, EVec, Q, temp_table, "col_id", "row_id",
            "value", "value",
                                "value", "row_id", "col_id", "row_id", "col_id")

        gm_sql_table_drop_create(db_conn, EVec,"row_id integer, col_id integer, value
            double precision")
        cur.execute("INSERT INTO %s" % (EVec) + " SELECT * FROM %s" % (temp_table))
        db_conn.commit()

        cur.execute("SELECT max(abs(value)) FROM %s" % (EVal) + " WHERE row_id <>
            col_id" )
        cur_err = cur.fetchone()[0]

        print "QR Algorithm Error = %s" % cur_err
        if cur_err <= err:
            break

cur.close()

# Drop temp tables
gm_sql_table_drop(db_conn, Q)
gm_sql_table_drop(db_conn, R)
```

```
    gm_sql_table_drop(db_conn, temp_table)
    gm_sql_table_drop(db_conn, I)



def gm_eigen (steps, num_nodes, err1, err2, adj_table=GM_TABLE_UNDIRECT):

    QR_max_iter = gm_param_qr_max_iter
    QR_stop_threshold = gm_param_qr_thres

    basis_vect_0 = "GM_EG_BASIS_VECT0"
    basis_vect_1 = "GM_EG_BASIS_VECT1"
    next_basis_vect = "GM_EG_BASIS_VECT_NEXT"
    temp_vect = "GM_EG_TEMP_VECT"
    temp_vect2 = "GM_EG_TEMP_VECT2"
    temp_vect3 = "GM_EG_TEMP_VECT3"
    basis = "GM_EG_BASIS"
    tridiag_table = "GM_EG_TRIDIAGONAL"
    diag_table = "GM_EG_DIAG"
    eigvec_table = "GM_EG_VEC"

    cur = db_conn.cursor();
    print "Computing Eigenvalues..."

    # create basis vectors
    gm_sql_vector_random(db_conn, basis_vect_1)
    gm_sql_create_and_insert(db_conn, basis_vect_0, GM_NODES, \
                             "id integer, value double precision", \
                             "id, value", "node_id, 0")

    # Create table to store the basis vectors
    gm_sql_table_drop_create(db_conn, basis,"row_id integer, col_id integer, value
        double precision")

    gm_sql_table_drop_create(db_conn, tridiag_table,"row_id integer, col_id integer,
        value double precision")

    beta_0 = 0
    beta_1 = 0
    alph_1 = 0

    for i in range(1, steps+1):
        print "Iteration No: " + str(i)

        # Get the next basis
        gm_sql_table_drop_create(db_conn, next_basis_vect,"id integer, value double
            precision")
        gm_sql_adj_vect_multiply(db_conn, adj_table, basis_vect_1, next_basis_vect,
            "dst_id",
                                  "id", "id", "value", "value", "src_id")

        alph_1 = gm_sql_vect_dotproduct (db_conn, next_basis_vect, basis_vect_1, "id",
            "id", "value", "value")

        gm_sql_table_drop_create(db_conn, temp_vect,"id integer, value double precision")
```

```
        # Orthogonalize with previous two basis vectors
        cur.execute("INSERT INTO %s " % (temp_vect) +
                    " (SELECT \"VECT_NEW\".id, " +
                    " (\"VECT_NEW\".value - (%s * \"VECT_0\".value) - (%s *
                       \"VECT_1\".value)) \"value\"" %
                                                    (beta_0, alph_1) +
                    " FROM %s \"VECT_NEW\", %s \"VECT_0\", %s \"VECT_1\" " %
                                                    (next_basis_vect,
                                                       basis_vect_0, basis_vect_1)
                                                       +
                    " WHERE \"VECT_NEW\".id = \"VECT_0\".id AND \"VECT_0\".id =
                       \"VECT_1\".id)")

        db_conn.commit()

        # Insert values into the tridiagonal table
        cur.execute("INSERT INTO %s" % (tridiag_table) + " VALUES(%s,%s,%s)" %
            (i,i,alph_1))
        if i>1:
            cur.execute("INSERT INTO %s" % (tridiag_table) + " VALUES(%s,%s,%s)" %
                (i-1,i, beta_0))
            cur.execute("INSERT INTO %s" % (tridiag_table) + " VALUES(%s,%s,%s)" %
                (i,i-1, beta_0))

        db_conn.commit()

        # Save the basis vector
        cur.execute("INSERT INTO %s " % (basis) +
                "SELECT id \"row_id\", %s \"col_id\", value " % (i) +
                "FROM %s" % (basis_vect_1))

        db_conn.commit()

        if i>1:
            gm_eigen_QR_iterate(tridiag_table, i, diag_table, eigvec_table, QR_max_iter,
                QR_stop_threshold)

            for j in range(1,i+1):
                cur.execute("SELECT abs(value) FROM %s" % (eigvec_table) +
                            " WHERE col_id=%s AND row_id=%s" % (j,i))

                thr = cur.fetchone()
                if thr:
                    thr = thr[0]
                else:
                    thr = 0

                if thr <= err1:
                    print "Performing SO with EigenVector " + str(j)
                    # Get corresponding eigenvector
                    gm_sql_table_drop_create(db_conn, temp_vect2,"id integer, value double
                        precision")

                    gm_sql_mat_colvec_multiply (db_conn, basis, eigvec_table, temp_vect2,
                        "col_id", "row_id",
```

```
                        "id", "value", "value", "value", "row_id",
                            "col_id="+str(j))

                # Selectively orthogonalize
                r = gm_sql_vect_dotproduct (db_conn, temp_vect2, temp_vect, "id",
                    "id", "value", "value")

                gm_sql_table_drop_create(db_conn, temp_vect3,"id integer, value double
                    precision")
                cur.execute("INSERT INTO %s " % (temp_vect3) +
                            " (SELECT \"VECT1\".id, " +
                            " (\"VECT1\".value - (%s * \"VECT2\".value)) \"value\"" %
                                (r) +
                            " FROM %s \"VECT1\", %s \"VECT2\" " % (temp_vect,
                                temp_vect2) +
                            " WHERE \"VECT1\".id = \"VECT2\".id)")

                db_conn.commit()

                gm_sql_table_drop_create(db_conn, temp_vect,"id integer, value double
                    precision")
                cur.execute("INSERT INTO %s" % (temp_vect) + " SELECT * FROM %s" %
                    (temp_vect3))

                db_conn.commit()

        beta_1 = gm_sql_normalize_vector (db_conn, temp_vect, "value");


        if abs(beta_1) <= err2:
            break

        # Prepare for next iteration
        gm_sql_table_drop_create(db_conn, basis_vect_0,"id integer, value double
            precision")
        cur.execute("INSERT INTO %s" % (basis_vect_0) + " SELECT * FROM %s" %
            (basis_vect_1))
        db_conn.commit()

        gm_sql_table_drop_create(db_conn, basis_vect_1,"id integer, value double
            precision")
        cur.execute("INSERT INTO %s" % (basis_vect_1) + " SELECT * FROM %s" % (temp_vect))
        db_conn.commit()

        beta_0 = beta_1

    # Get the eigen values and eigen vectors
    gm_eigen_QR_iterate(tridiag_table, i, diag_table, eigvec_table, QR_max_iter,
        QR_stop_threshold)

    gm_sql_table_drop_create(db_conn, GM_EIG_VALUES,"id integer, value double precision")

    print "Getting EigenValues..."
    # Get top eigen values
    cur.execute("INSERT INTO %s" % (GM_EIG_VALUES) +
```

```
                    " SELECT col_id \"id\", value \"value\" FROM %s" % (diag_table) +
                    " WHERE col_id = row_id ORDER BY abs(value) desc")

    db_conn.commit()

    # Get the top k eigenvectors
    print "Getting top %s EigenVectors..." % gm_param_eig_k
    cur2 = db_conn.cursor();
    gm_sql_table_drop_create(db_conn, GM_EIG_VECTORS,"row_id integer, col_id integer,
        value double precision")

    cur.execute("SELECT id FROM %s ORDER BY abs(value) desc LIMIT %s " %
        (GM_EIG_VALUES,gm_param_eig_k))
    for idx in cur:
        i = idx[0]
        print "Getting eigenvector %s" % i
        gm_sql_table_drop_create(db_conn, temp_vect2,"id integer, value double precision")
        gm_sql_mat_colvec_multiply (db_conn, basis, eigvec_table, temp_vect2, "col_id",
            "row_id",
                              "id", "value", "value", "value", "row_id", "col_id="+str(i))

        cur2.execute("INSERT INTO %s SELECT id \"row_id\", %s \"col_id\", value" %
            (GM_EIG_VECTORS,i) +
                              " FROM %s" % (temp_vect2))
        db_conn.commit()

    cur2.close()

#
#    gm_sql_mat_mat_multiply (db_conn, basis, eigvec_table, GM_EIG_VECTORS, "col_id",
    "row_id", "value", "value",
#                                          "value", "row_id", "col_id", "row_id", "col_id")

    print "EigenValues computed: "
    gm_sql_print_table(db_conn, GM_EIG_VALUES)
    #gm_sql_print_table(db_conn, GM_EIG_VECTORS)

    cur.close()
    # Drop temp tables
    gm_sql_table_drop(db_conn, basis_vect_0)
    gm_sql_table_drop(db_conn, basis_vect_1)
    gm_sql_table_drop(db_conn, next_basis_vect)
    gm_sql_table_drop(db_conn, temp_vect)
    gm_sql_table_drop(db_conn, temp_vect2)
    gm_sql_table_drop(db_conn, temp_vect3)
    gm_sql_table_drop(db_conn, basis)
    gm_sql_table_drop(db_conn, tridiag_table)
    gm_sql_table_drop(db_conn, diag_table)
    gm_sql_table_drop(db_conn, eigvec_table)


# Task 6: Fast Belief Propagation
# -------------------------------------------------------------------------- #
def gm_belief_propagation(belief_file, delim, undirected, \
              max_iterations = gm_param_bp_max_iter, stop_threshold = gm_param_bp_thres):
```

```
next_table = "GM_BP_NEXT"
print "Computing belief propagation..."

# BP require that the graph is undirected.
if (undirected):
    degree_col = "out_degree"
else:
    degree_col = "out_degree+in_degree"

cur = db_conn.cursor()
cur.execute ("SELECT MAX(%s), SUM(%s), SUM((%s)*(%s))" % (degree_col, degree_col,
    degree_col, degree_col) +
            "FROM %s" % GM_NODE_DEGREES)
max_deg, sum_deg, sum_deg2 = cur.fetchone()

c1 = 2+sum_deg
c2 = sum_deg2 -1

h = max(1 / (float)(2 + 2 * max_deg), sqrt((-c1 + sqrt(c1*c1 + 4*c2))/(8*c2)))
print "Homophily factor = " + str(h)

a = (4*h*h)/(1-4*h*h)
c = (2*h)/(1-4*h*h)

print "Getting the priors..."
# Get the belief priors.
gm_sql_table_drop_create(db_conn, GM_BELIEF_PRIOR, "node_id integer, belief double
    precision")
gm_sql_load_table_from_file(db_conn, GM_BELIEF_PRIOR, "node_id, belief",
    belief_file, delim)

# Initialize belief table as belief priors
gm_sql_create_and_insert(db_conn, GM_BELIEF, GM_BELIEF_PRIOR, \
                         "node_id integer, belief double precision", \
                         "node_id, belief", "node_id, belief")

num_iterations = 0
while True:
    # Create Table to store the next belief
    gm_sql_table_drop_create(db_conn, next_table,"node_id integer, belief double
        precision")

    # Compute next belief
    cur.execute ("INSERT INTO %s " % next_table +
                         " SELECT node_id, SUM(belief) FROM (" +
                         " SELECT src_id \"node_id\", %s * SUM(belief) \"belief\" "
                            % c +
                         " FROM %s, %s" % (GM_TABLE_UNDIRECT, GM_BELIEF) +
                         " WHERE dst_id = node_id GROUP BY src_id" +
                         " UNION ALL" +
                         " SELECT \"D\".node_id \"node_id\", %s*(%s)*belief
                            \"belief\"" % (-a, degree_col) +
                         " FROM %s \"D\", %s \"B\" " % (GM_NODE_DEGREES, GM_BELIEF) +
                         " WHERE \"D\".node_id = \"B\".node_id" +
```

```
                                " UNION ALL" +
                                " SELECT * FROM %s " % GM_BELIEF_PRIOR +
                                " ) \"TAB\" GROUP BY node_id" )

        db_conn.commit()

        diff = gm_sql_vect_diff(db_conn, GM_BELIEF, next_table, "node_id", "node_id",
            "belief", "belief")

        # Recreate Belief table and copy values
        gm_sql_create_and_insert(db_conn, GM_BELIEF, next_table, \
                                "node_id integer, belief double precision", \
                                "node_id, belief", "node_id, belief")

        num_iterations = num_iterations + 1
        print "Iteration = %d, Error = %f" % (num_iterations, diff)

        if (diff<=stop_threshold or num_iterations>max_iterations):
            break


    # Drop temp tables
    gm_sql_table_drop(db_conn, next_table)

    cur.close()



# Task 7: Triangle counting
# -------------------------------------------------------------------------- #
def gm_naive_triangle_count(adj_table=GM_TABLE_UNDIRECT):

    temp_table = "GM_TRIANG_TEMP"
    temp_table2 = "GM_TRIANG_TEMP2"
    temp_table3 = "GM_TRIANG_TEMP3"

    cur = db_conn.cursor()
    gm_sql_table_drop_create(db_conn, temp_table,"row_id integer, col_id integer, value
        double precision")
    gm_sql_table_drop_create(db_conn, temp_table2,"row_id integer, col_id integer, value
        double precision")
    gm_sql_table_drop_create(db_conn, temp_table3,"row_id integer, col_id integer, value
        double precision")

    # Copy the adjacency matrix
    cur.execute("INSERT INTO %s" % (temp_table) + \
                " SELECT src_id \"row_id\", dst_id \"col_id\", 1 \"value\" FROM %s" %
                    (adj_table))

    db_conn.commit()

    # Compute A^2
    gm_sql_mat_mat_multiply (db_conn, temp_table, temp_table, temp_table2, "col_id",
        "row_id", "value", "value",
                                        "value", "row_id", "col_id", "row_id", "col_id")
```

```
    # Compute A^3
    gm_sql_mat_mat_multiply (db_conn, temp_table2, temp_table, temp_table3, "col_id",
        "row_id", "value", "value",
                                    "value", "row_id", "col_id", "row_id", "col_id")


    cnt = gm_sql_mat_trace(db_conn, temp_table3, "row_id", "col_id", "value")

    print "Number of Triangles(naive) = " + (str(cnt/6))

    cur.close()

    # Drop temp tables
    gm_sql_table_drop(db_conn, temp_table)
    gm_sql_table_drop(db_conn, temp_table2)
    gm_sql_table_drop(db_conn, temp_table3)


def gm_eigen_triangle_count():

    cur = db_conn.cursor()
    #gm_eigen(steps, num_nodes, err1, err2, adj_table)
    print "Computing the count of triangles..."

    cur.execute("SELECT sum(value^3) FROM %s" % (GM_EIG_VALUES))
    cnt = cur.fetchone()[0]

    print "Number of Triangles = " + (str(cnt/6))

    cur.close()


# Innovative Task : Anomaly Detection for unidrected graphs
def gm_anomaly_detection():
    cur = db_conn.cursor()
    gm_sql_table_drop_create(db_conn, GM_EGONET,"node_id integer, edge_cnt integer,
        wgt_sum double precision")

    print "Extracting Features from Egonets"

    start_time = time.time()
    cur.execute("INSERT INTO %s " % (GM_EGONET) +
                        " SELECT node_id, sum(edge_cnt) \"edge_cnt\", sum(wgt_sum)
                            \"wgt_sum\" FROM" +
                        " (SELECT \"T2\".dst_id \"node_id\", count(*)/2 \"edge_cnt\",
                            sum(\"T2\".weight)/2 \"wgt_sum\" " +
                        " FROM %s \"T1\", %s \"T2\", %s \"T3\" " % (GM_TABLE_UNDIRECT,
                            GM_TABLE_UNDIRECT, GM_TABLE_UNDIRECT) +
                        " WHERE \"T1\".src_id = \"T2\".src_id AND \"T1\".dst_id =
                            \"T3\".dst_id AND \"T2\".dst_id=\"T3\".src_id" +
                        " GROUP BY \"T2\".dst_id" +
                        " UNION ALL" +
                        " SELECT src_id \"node_id\", count(*) \"edge_cnt\", sum(weight)
                            \"wgt_sum\" " +
                        " FROM %s GROUP BY src_id) \"TAB\" " % (GM_TABLE_UNDIRECT) +
```

```
                    " GROUP BY node_id");

    db_conn.commit();
    print "Time taken = " + str(time.time()-start_time)



def main():
    global db_conn
    global GM_TABLE
    global index_type
     # Command Line processing
    parser = argparse.ArgumentParser(description="Graph Miner Using SQL v1.0")
    parser.add_argument ('--file', dest='input_file', type=str, required=True,
                   help='Full path to the file to load from. For weighted \
                   graphs, the file should have the format (<src_id>, <dst_id>,
                       <weight>) \
                   . If unweighted please run with --unweighted option. To specify a
                       \
                   delimiter other than "," (default), use --delim option. \
                   NOTE: The file should have proper permissions set for \
                   the postgres user.'
                   )
    parser.add_argument ('--delim', dest='delimiter', type=str, default=',',
                   help='Delimiter that separate the columns in the input file.
                       default ","')

    parser.add_argument ('--unweighted', dest='unweighted', action='store_const',
        const=True, default=False,
                   help='For unweighted graphs. The input file should be of the form
                       \
                   (<src_id>, <dst_id>). For algorithms that require weighted
                       graphs, default weight \
                   of 1 will be used')

    parser.add_argument ('--undirected', dest='undirected', action='store_const',
        const=True, default=False,
                   help='Treat the graph as undirected instead of directed
                       (default). If this is set \
                   the input graph is first converted into an undirected version by
                       adding reversed edges \
                   with same weight. NOTE: Graph algorithms like eigen values,
                       triangle counting, \
                   connected components etc require undirected graphs and such
                       algorithms work with \
                   undirected version of the graph irrespective of whether this
                       option is set.')

    parser.add_argument ('--dest_dir', dest='dest_dir', type=str, required=True,
                   help='Full path to the directory where the output tables are
                       saved')

    parser.add_argument ('--belief_file', dest='belief_file', type=str, default='',
                   help='Full path to belief priors file. The file should be in the
                       format \
```

```
                    (<node_id>, <belief>). Specify a different delimiter with --delim
                        option.\
                    The prior beliefs are expected to be centered around 0. i.e.
                        positive \
                    nodes have priors >0, negative nodes <0 and unknown nodes 0. ')

    args = parser.parse_args()

    try:
        # Run the various graph algorithm below
        db_conn = gm_db_initialize()


        gm_sql_table_drop_create(db_conn, GM_TABLE, "src_id integer, dst_id integer,
            weight real default 1")
        if (args.unweighted):
            col_fmt = "src_id, dst_id"
        else:
            col_fmt = "src_id, dst_id, weight"

        gm_sql_load_table_from_file(db_conn, GM_TABLE, col_fmt, args.input_file, ',')
        cur = db_conn.cursor()

        gm_to_undirected(True)

        if (args.undirected):
            GM_TABLE = GM_TABLE_UNDIRECT

        if index_type == 1:
            # non-clustering
            cur.execute("create index src on %s (src_id)" % GM_TABLE)
        elif index_type == 2:
            # clustering
            cur.execute("create index src on %s (src_id)" % GM_TABLE)
            cur.execute("CLUSTER %s USING src" % GM_TABLE)
        elif index_type == 3:
            # composite
            cur.execute("create index compo on %s (src_id, dst_id)" % GM_TABLE)
            cur.execute("CLUSTER %s USING compo" % GM_TABLE)

        db_conn.commit()
        # Create table of node ids
        gm_create_node_table()

        # Get number of nodes

        cur.execute("SELECT count(*) from %s" % GM_NODES)
        num_nodes = cur.fetchone()[0]

        start_time = time.time()
        gm_node_degrees()
        # k core
        gm_kcore()
        # degree distribution
        gm_degree_distribution(args.undirected)           # Degree distribution
```

```
        # pagerank
        gm_pagerank(num_nodes)                                # Pagerank
        # connected components
        gm_connected_components(num_nodes)                    # Connected components
        # eigen values
        gm_eigen(gm_param_eig_max_iter, num_nodes, gm_param_eig_thres1,
            gm_param_eig_thres2)
        # node radius
        gm_all_radius(num_nodes)
        # fast belief prop
        if (args.belief_file):
            gm_belief_propagation(args.belief_file, args.delimiter, args.undirected)
        # triangle count
        gm_eigen_triangle_count()

        #gm_naive_triangle_count()
        print "Overrall time taken = " + str(time.time() - start_time)

        # Save tables to disk
        gm_save_tables(args.dest_dir, args.belief_file)
        #gm_anomaly_detection()

        gm_db_bubye(db_conn)
    except:
        print "Unexpected error:", sys.exc_info()[0]
        if (db_conn):
            gm_db_bubye(db_conn)

        raise

if __name__ == '__main__':
    main()
```