Estimating the fundamental matrix of a random walk transition matrix

Clara De Paolis Kaluza

Math250-Graph Algorithms

Final project

Abstract

The analysis of distances between nodes in networks that contain many low-degree vertices and some high-degree hubs requires more sophisticated distance metrics than, for example, simple shortest-path distances. In the study of one type of these networks, protein-protein interaction networks, a promising distance metric is the diffusion state distance, but calculating requires an expensive matrix inversion. Through reforming the problem as a linear system theoretical performance improvements are possible, although the experimental results shown here show numerical instability issues.

1. Problem Statement and Motivation

1.1. Motivation

Genome sequencing allows for the study of all the proteins expressed by the genome of an organism (the proteome). For most organisms, some of these proteins have known biological functions, but for many proteins their biological function is unknown. A protein-protein interaction (PPI) network relates the structure of the proteins expressed by a genome, representing proteins that physically interact as connected nodes in a graph. These graphs capture physical interactions between proteins, including those with known function and those with unknown functions. Therefore if a proper distance metric can be determined between nodes, the structure of these networks can be used to discover the function of uncharacterized proteins[5].

However, these networks can be very complex, and furthermore, determining an appropriate distance metric is not straightforward. Some nodes in the network are considered “hubs,” connecting many proteins that are not functionally similar. Therefore, a distance metric such as a simple shortest path is not useful in identifying functionally similar proteins. The work in [2] shows a promising distance metric, the diffusion state distance (DSD), but calculating it exactly requires the inversion of a matrix corresponding to a large network even for simple organisms. A more efficient method to calculate this metric is needed, especially in order to analyze the much larger PPI networks of more complex organisms and to apply this metric to other, larger networks. The main objective of this project is to implement an efficient way to estimate the DSD while avoiding an exact calculation of the fundamental matrix, an inverse of an $n \times n$ matrix, which is approximately an $O(n^3)$ operation.
2. Background and Assumptions

2.1. Problem Setup

To more precisely define the problem, consider a connected graph $G = (V, E)$ where nodes $v \in V$ represent proteins expressed by a genome and edges $e \in E$ represent physical interaction between those proteins. The number of edges $e$ connected to any node $i$ is the degree of that node, represented by $d(i)$. If we define the probability of a transition from node $i$ to any of its neighboring nodes $j$ as uniform then, the state transition matrix $P$ for this network is given by

$$
(P)_{ij} = \begin{cases} 
\frac{1}{d(i)} & e(i, j) \in E \\
0 & \text{otherwise} 
\end{cases}
$$

(1)

For the Markov chain this network represents, the matrix $W = \lim_{n \to \infty} P^n$ is such that each row of $W$ is equal to $\pi^T$, the vector describing the steady state distribution for each $v \in V$. Finally, the fundamental matrix of $P$ is given by $Z = (I - P + W)^{-1}$.

The distance metric proposed in [2] between two nodes (proteins) $u$ and $v$ is then defined as:

$$
\text{DSD}(u, v) = \| (b^T_u - b^T_v)(I - P + W)^{-1} \|_1
$$

(2)

where $b_i$ is a basis vector, where all entries are zero except entry in the $i^{th}$ position is one.

2.2. Defining The Graph Problem

For compactness, define the following two vectors $b^T_{uv} := (b^T_u - b^T_v)$ and $x^T_{uv} := b^T_{uv}(I - P + W)^{-1}$ so that $\text{DSD}(u, v) = \| x^T_{uv} \|_1 = \| x_{uv} \|_1$. By rearranging the terms, we can state this problem in the familiar form of solving a linear system:

$$
x^T_{uv} = b^T_{uv}(I - P + W)^{-1}
$$

$$
((I - P + W)^T)^{-1}b^T_{uv} = x_{uv}
$$

$$
(I - P + W)^Tx_{uv} = b_{uv}
$$

$$
(I - P^T + W^T)x_{uv} = b_{uv}
$$

From the definition of the state transition matrix in [1], $P$ can be represented in terms of the matrices that describe a graph, namely, the degree matrix $D$, the adjacency matrix $A$, and the graph Laplacian $L$.

$$
P = D^{-1}A
$$

$$
\Rightarrow P^T = AD^{-1}
$$

$$
\Rightarrow (I - P^T) = I - AD^{-1} = (D - A)D^{-1} = LD^{-1}
$$

So the linear system to be solved becomes

$$
(I - P^T + W^T)x_{uv} = (LD^{-1} + W^T)x_{uv} = b_{uv}
$$

(3)
To further simplify, consider the Sherman-Morrison formula for inverting the sum of a square invertible matrix $A$ and the outer product of two vectors $u$ and $v$:

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

with the restriction that $1 + v^TA^{-1}u \neq 0$.

For this problem, the first matrix is $(I - P^T)$, and the two column vectors are the vector of all ones $e$ and $\pi$, the stationary distribution of the Markov chain described by $P$ so that $W = e\pi^T$.

So substituting into the above equation, we get a solution to finding the fundamental matrix $Z$:

$$Z^T = (I - P^T + W^T)^{-1} = (I - P^T + (e\pi^T)^{-1}) = (I - P^T + \pi e^T)^{-1}$$

$$= (I - P^T)^{-1} - \frac{(I - P^T)^{-1}\pi e^T(I - P^T)^{-1}}{1 + e^T(I - P^T)^{-1}\pi}$$

$$= (LD^{-1})^{-1} - \frac{(LD^{-1})^{-1}\pi e^T(LD^{-1})^{-1}}{1 + e^T(LD^{-1})^{-1}\pi}$$

$$= DL^{-1} - \frac{DL^{-1}\pi e^TDL^{-1}}{1 + e^TDL^{-1}\pi}$$

Substituting into (3), the linear system becomes

$$x_{uv} = Z^Tb_{uv}$$

$$= (DL^{-1} - \frac{DL^{-1}\pi e^TDL^{-1}}{1 + e^TDL^{-1}\pi})b_{uv}$$

$$= DL^{-1}b_{uv} - \frac{DL^{-1}\pi e^TDL^{-1}b_{uv}}{1 + e^TDL^{-1}\pi}$$

$$= Dy_{uv} - \frac{Dz e^T Dy_{uv}}{1 + e^T Dz}$$

$$= Dy_{uv} - \left(\frac{e^T Dy_{uv}}{1 + e^T Dz}\right) Dz$$

where $y_{uv} = L^{-1}b_{uv}$ and $z = L^{-1}\pi$. Therefore to solve this system, the following two linear systems of graph Laplacian must be solved:

$$Ly_{uv} = b_{uv} \text{ and } Lz = \pi$$

3. Methods

3.1. Approach

Using the solution to the systems in (5), the original problem can be solved by following the approach outlined in Algorithm 1. Here, we use an aggregation-based algebraic multigrid approach to solve the two linear systems in steps 1 and 2, however this method can be substituted by an alternative linear system solving method.
Algorithm 1 Compute DSD(u, v) given $b_{uv}$, graph Laplacian $L = D - A$, and $\pi$

1. Solve $Ly_{uv} = b_{uv}$
2. Solve $Lz = \pi$
3. Compute $f = Dy_{uv}$ and $g = Dz$
4. Compute $x_{uv} = f - \left( \frac{e^T f}{1 + e^T g} \right) g$
5. Compute $DSD(u, v) = \|x_{uv}^T\|_1 = \|x_{uv}\|_1$

3.2. Solving the Linear Systems

The two linear systems could be solved using several possible methods to achieve a performance gain over directly computing the fundamental matrix (if it can even be computed), but the method implemented in this work is an aggregation-based algebraic multigrid (AMG) method. AMG methods can be used to solve graph Laplacian systems and take the general form of forming aggregates to form $P_l$ then constructing the graph Laplacian for each coarser level using $L_{l+1} = P_l^T L_l P_l$, then recursively calling the AMG cycle algorithm (Algorithm 2)

Algorithm 2 Algebraic Multigrid with Aggregations, AMGCycle($x_l, L_l, b_l, l$)

- if at coarsest level:
  - solve $x = L_l^{-1} b_l$ directly
- else:
  - Pre-smoothing (update to $x_l$)
  - Compute residual $r_l \leftarrow b_l - L_l x_l$
  - Restriction $r_c \leftarrow P_l^T r_l$
  - Coarse-grid correction (recursive call to AMG Cycle)
  - Prolongation $x_l \leftarrow x_l + P_l e_{l+1}$
  - Post-smoothing (update to $x_l$)

4. Results

4.1. Solving $Ly_{uv} = b_{uv}$

Table 1 shows the performance of AMG in solving $Ly_{uv} = b_{uv}$. Since the DSD between each pair of nodes in the PPI is needed, the calculation must be performed for each pair of unique nodes (since DSD(u, u) = 0 and DSD(u, v) = DSD(v, u), as shown in [2]).
| Species | $|V|$ | $|E|$ | $d_{\text{max}}$ | $d_{\text{avg}}$ | method                  | setup time(s) | num iters | solve time(s) |
|---------|------|------|------------|------------|------------------------|--------------|-----------|--------------|
| worm    | 5,281| 13,829| 225        | 5.237      | two-level AMG          | 2.327        | 23        | 3.641        |
|         |      |       |            |            | V-cycle AMG            | 2.358        | 24        | 3.138        |
|         |      |       |            |            | W-cycle AMG            | 2.389        | 23        | 5.193        |
| mouse   | 6,596| 18,697| 714        | 5.669      | two-level AMG          | 2.100        | 29        | 10.959       |
|         |      |       |            |            | V-cycle AMG            | 3.274        | 30        | 8.824        |
|         |      |       |            |            | W-cycle AMG            | 3.133        | 29        | 10.97        |
| yeast   | 6,096| 216,531| 3,472     | 71.040     | two-level AMG          | 2.434        | 10        | 269.132      |
|         |      |       |            |            | V-cycle AMG            | 3.209        | 10        | 277.139      |
|         |      |       |            |            | W-cycle AMG            | 4.059        | 10        | 270.089      |
| human   | 15,129| 155,866| 9,388     | 20.605     | two-level AMG          | 5.556        | 14        | 1543.422     |
|         |      |       |            |            | V-cycle AMG            | 9.300        | *         | *            |
|         |      |       |            |            | W-cycle AMG            | 9.644        | *         | *            |

Table 1: Results of solving the linear system $L_y u_v = b_{uv}$ with a tolerance for the relative residual of for the largest connected component of the PPI network for several species.

The number of proteins (vertices in $G$) is indicated by $|V|$, the number of edges by $|E|$, the maximal degree by $d_{\text{max}}$, and the average degree by $d_{\text{avg}}$.

*For the human PPI network, the V-cycle AMG method reached a relative residual of $2.021 \times 10^{-8}$ at 12 iterations before starting to increase again. For the W-cycle, the relative residual reached $8.647 \times 10^{-6}$ at 5 iterations before increasing.*

4.2. Solving $Lz = \pi$

Using the same method as above was used to solve $L_y u_v = b_{uv}$, the AMG method for solving $Lz = \pi$ does not converge, with the rate oscillating around 1, therefore the residual never converging towards the tolerance set by the algorithm. To investigate this behavior, the system $Lz = \pi$ was solved using two different solvers available through Python’s SciPy and NumPy packages. SciPy provides a sparse matrix solver `scipy.sparse.linalg.spsolve` and Numpy provides a solver `numpy.linalg.solve`. When used on this linear system, the former found $z = (4.876 \times 10^{11})e$ and the latter found $z = (6.546 \times 10^{12})e$ for the worm PPI network, where $e$ denotes the vector of all ones. Both solutions are clearly very far from both the starting guesses for $z (z = \text{ones}(n,1))$ and $z = \text{zeros}(n,1))$ which were attempted with the AMG method. However, changing the initial guess for $z$ to $z = 10^{11}\text{ones}(n,1)$ did not lead to converging behavior either, suggesting a different cause of the numerical issue.

5. Conclusion

The theory presented in this work shows that the calculation of the fundamental matrix corresponding to a Markov chain can be replaced by solving a set of linear systems and some matrix-vector and vector-vector multiplications. Although the experimental results show numerical issues
with one of the linear systems, changes to the implemented method or substituting for another solving method may yield positive results.

5.1. Future Work and Applications

The goal of this work is to develop a method to calculate the DSD distance metric when finding the fundamental matrix is unfeasible. The PPI networks presented here are relatively small, containing thousands or tens of thousands of vertices. However, this same distance metric can prove useful in identifying similar entities in other much larger graphs which exhibit similar properties as the PPI networks, namely networks that contain hubs which link otherwise unrelated entities to each other over potentially short paths. Examples of such networks include large social networks or a network representing linked article on Wikipedia or linked webpages on the Internet. In 2011, the largest connected component of the Facebook social network was found to contain \(99.91\%\) of Facebook’s 721 million active(signed in within 28 days of the analysis) users, with the number of edges \(|E|\) in the entire graph equal to 68.7 billion, although the edges in the largest component were not specified \[4\]. A more recent statistic showed 1.65 billion active in March 2016 \[1\]. When undertaking analyses of such networks, a matrix inversion at \(O(n^3)\) is prohibitive and an efficient algorithm such as that proposed by the theory in the project would prove even more useful.

References


A. Appendix: Code

To run code:
need python 3 and the packages networkx, numpy, scipy, argparse, and timeit
to execute:

> python dsdAMG.py -f [ppi file name without the .ppi extension]

for example

> python dsdAMG.py -f worm

```python
import networkx as nx
import argparse
import numpy as np
import scipy.sparse
import dsd_solveAMG

# Precondition: adj is a NetworkX adjacency matrix of a connected undirected graph
# Postcondition: Returns a NetworkX matrix of transition probabilities for
# a random walk in the graph represented by adjacency.

def createTransitionMatrix(adj):
    n = np.size(adj[0])
    p = np.zeros((n, n))
    degree = np.zeros((n, 1))
    for j in range(n):
        degree[j] = sum(adj[j])
        if degree[j] != 0:
            p[j] = adj[j]/degree[j]
    return p, degree

# Takes in a networkx graph, 'graph', and a list of nodes in the graph
# and returns the adjacency matrix of the graph with the ordering
# in 'nodelist' as a numpy array

def createAdjacencyMatrix(graph, nodelist):
    return np.array(nx.adjacency_matrix(graph, nodelist).todense())

# Return the canonical node ordering, which is the nodes of the graph
# in sorted order

def getNodeOrdering(graph):
    return sorted(graph.nodes())

# b_uv = scipy.sparse.coo_matrix([[1,-1], (u, v)), shape=(n, 1))

b_uv = np.zeros((n, 1))
```
b_uv[u,0] = 1
b_uv[v,0] = -1
return b_uv

def createMatrix(G):
    nodeList = getNodeOrdering(G)
    adj = createAdjacencyMatrix(G, nodeList)

    # number of nodes
    n = np.size(adj[0])
    _, degree = createTransitionMatrix(adj)

    # create w using the fact that the
    # steady state of an undirected random walk
    # is proportional to node degree
    pi = (degree)/sum(degree)

    # create degree matrix with correct ordering
    D = scipy.sparse.dia_matrix((degree.T, [0]), shape=(n,n))

    L = D - scipy.sparse.csc_matrix(adj)
    return L, pi, D

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("-f", required=True, help="PPI file")
    args = parser.parse_args()
    ppi = args.f
    ppi_file = '../../data/' + ppi + '.ppi'

    # Create graph and needed matrices and vectors
    G = nx.read_edgelist(ppi_file, nodetype=str)
    L, pi, D = createMatrix(G)
    n = len(pi)
    (u,v) = (0,1)
    b_uv = basis_uv(u,v,n)

    print('AMG Solve for ', ppi)
    for cycle in ['W']:
        print('AMG Cycle type', cycle)
        # Set up AMG levels and parameters
        # only needs to be done once for solving for linear systems,
        # unless different cycle_types are necessary
        amgData, amgParam = dsd_solveAMG.setup_AMG(L,cycle_type=cycle)

        # Solve the two linear systems Ly_{uv}=b_{uv} and Lz=pi
        y_uv = dsd_solveAMG.solve_AMG(amgData, b_uv, amgParam)
        z = dsd_solveAMG.solve_AMG(amgData, pi, amgParam)

        # Compute the vertices for finding fundamental matrix (I - P + W)
        f = D.dot(y_uv)
        g = D.dot(z)
        x_uv = f - ((np.ones(n).dot(f))/(1 + np.ones(n).dot(g))).dot(g)
import numpy as np
from AMG_Setup import *
from AMG_Solve import *

# solve graph Laplacian using AMG
#
# adapted from matlab code from @ Xiaozhe Hu, Tufts University

def setup_AMG(L, cycle_type='V'):
    """
    Sets up levels and parameters for solving Lx=b using cycle-type specified
    Parameters
    ----------
    L: Graph Laplacian
    cycle_type: AMG cycle type, either 'TL' for two-level, 'V' for V-cycle (default), or 'W' for W-cycle

    Returns
    ------
    amgData
    amgParams
    """

    #---------------------
    # AMG parameters
    #---------------------
    amgParam = {}
    amgParam.update({'print_level': 1})  # how much information to print when using AMG solve only
    # 0: print nothing | positive number print information
    
    # setup phase parameters
    amgParam.update({'max_level': 20})  # maximal number of level in AMG
    amgParam.update({'coarsest_size': 100})  # size of the coarsest level

    # solve phase parameters
    amgParam.update({'cycle_type':cycle_type})  # 'TL: Two-level | 'V': V-cycle | 'W': W-cycle
    amgParam.update({'n_presmooth': 1})  # number of presmoothing
    amgParam.update({'n_postsmooth': 1})  # number of postsmoothing

    amgParam.update({'max_it': 100})  # when AMG is used as standalone solver, maximal number of iterations that is
    amgParam.update({'tol': 1e-8})  # when AMG is used as standalone solver, tolerance for the relative residual

    #---------------------
    # setup phase
    #---------------------
    amgData = AMG_Setup(L, amgParam)
    return amgData, amgParam
def solve_AMG(amgData, b, amgParam):
    """
    Solve Lx=b through AMG
    Parameters
    -----------
    amgData: AMG data produced through AMG_Setup
    b: Right-hand side
    amgParam: AMG parameters produced through AMG_Setup
    cycle_type: AMG cycle type, either 'TL' for two-level, 'V' for V-cycle (default), or 'W' for W-cycle
    Returns
    -------
    x: solution to Lx=b
    """
    n = b.shape[0]
    x = np.zeros((n,1)) #initial guess
    (x, k, err) = AMG_Solve(amgData, b, x, amgParam)
    return x

import numpy as np
import scipy.sparse
from timeit import default_timer as timer
import support_scripts, form_aggregates

def AMG_Setup(Lf, amgParam):
    # Setup phase for AMG method
    # adapted from Matlab code by Xiaozhe Hu, Tufts University
    #----------------
    # local variable
    #----------------
    print_level = amgParam['print_level']
    max_level = amgParam['max_level']
    if amgParam['cycle_type']=='TL':
        max_level = 2
    coarsest_size = amgParam['coarsest_size']
    level = 0
    #----------------
    # AMG information
    #----------------
    AMG_Data={level:{} for level in range(max_level)}
    #----------------
    # finest level
    #----------------
    AMG_Data[0].update({'L':Lf,
'N': Lf.shape[0],
'DL': scipy.sparse.tril(Lf, format='csr'),
'DU': scipy.sparse.triu(Lf, format='csr'),
'D': Lf.diagonal(),
'max_level': 0})

#----------------
# main loop
#----------------
print('----------------------------------------------------')
print(' Calling AMG setup ')
print('----------------------------------------------------')

setup_start = timer()

while (level < max_level-1) and (AMG_Data[level]['N'] > coarsest_size):
    #----------------
    # form aggregation
    #----------------
    # implement your own aggregation algorithm
    # input: L{level} -- graph Laplacian on current level
    # output: aggregation -- information about aggregates
    #         (aggregation(i) = j mean the i-th vertex belong to aggregates j)
    #         num_agg -- number of aggregations
    # (aggregation, num_agg) = form_aggregates.form_aggregates(AMG_Data[level]['L'])

    #----------------
    # generate prolongation
    #----------------
    AMG_Data[level]['P'] = support_scripts.generate_unsmoothed_P(aggregation, num_agg)

    #----------------
    # generate restriction
    #----------------
    AMG_Data[level]['R'] = AMG_Data[level]['P'].transpose()

    #----------------
    # compute coarse grid matrix
    #----------------
    AMG_Data[level+1]['L'] = AMG_Data[level]['R'].dot(AMG_Data[level]['L']).dot(AMG_Data[level]['P'])
    AMG_Data[level+1]['N'] = AMG_Data[level+1]['L'].shape[0]

    #----------------
    # extra information for smoothers
    #----------------
    AMG_Data[level+1]['DL'] = scipy.sparse.tril(AMG_Data[level+1]['L'], format='csr')
    AMG_Data[level+1]['DU'] = scipy.sparse.triu(AMG_Data[level+1]['L'], format='csr')
    AMG_Data[level+1]['D'] = AMG_Data[level+1]['L'].diagonal()

    #----------------
    # update
    #----------------
    level += 1
setup_duration = timer() - setup_start

# construct the data structure
for l in range(level+1):
    AMG_Data[l]['max_level'] = level

# print information
if print_level > 0:
    total_N = 0
    total_NNZ = 0

    print('-----------------------------------------------------
    print('# Level	| # Row	| # Nonzero	| Avg. NNZ/Row#
    print('-----------------------------------------------------

    for i in range(level+1):
        nonzero_i = len(AMG_Data[i][L].nonzero()[0])
        N_i = AMG_Data[i][N]
        total_N += N_i
        total_NNZ += nonzero_i

        print('%2d	|%9d	| %10d	| %7.3f	| % (i, N_i, nonzero_i, nonzero_i/N_i)

    print('-----------------------------------------------------
    print(# Grid complexity: %0.3f | Operator complexity: %0.3f % (total_N/AMG_Data[0][N], total_NNZ/len(AMG_Data[0][L].nonzero()[0]))
    print('-----------------------------------------------------

    # print cputime
    print('-----------------------------------------------------
    print('AMG setup costs', setup_duration, 'seconds')
    print('-----------------------------------------------------

    return AMG_Data

import numpy as np
import timeit import default_timer as timer
import support_scripts
from AMG_Cycle import *

def AMG_Solve(AMG_Data, b, x, amgParam):
    # Solve phase for AMG method
    # adapted from Matlab code by Xiaozhe Hu, Tufts University

    # parameters
    print_level = amgParam['print_level']
    max_it = amgParam['max_it']
tol = amgParam['tol']

# prepare solve
level = 0
err = np.zeros((max_it+1,1))

r = b - amgData[0]['L'].dot(x)
err[0] = np.linalg.norm(r)

# print
print('----------------------------------------------------')
print(' Calling AMG solver  
print('----------------------------------------------------')

if print_level > 0:
    print('----------------------------------------------------')
    print(' # It | ||r||/||r0|| | ||r|| | Rate. |')
    print('----------------------------------------------------')
    print(' %4d | %e | %e | %f |' % (0, 1.0, err[0], 0.0))

# main loop
solve_start = timer()

for k in range(max_it):

    # call multigrid
    x = AMG_Cycle(amgData, b, x, level, amgParam)

    # compute residual
    r = b - amgData[level]['L'].dot(x)

    # compute error
    err[k+1] = np.linalg.norm(r)

    # display
    if print_level > 0:
        print(' %4d | %e | %e | %f |' % (k+1, err[k+1]/err[0], err[k+1], err[k+1]/err[k]))
    if (err[k+1]/err[0]) < tol:
        break

solve_duration = timer() - solve_start

# cut err
err = err[:k+1]

# print
print('----------------------------------------------------')
if k == max_it:
    print(' AMG reached maximal number of iterations ')
else:
    print(' AMG converged or reached max iterations ')
import numpy as np
import scipy.sparse, scipy.sparse.linalg
import support_scripts

def AMG_Cycle(amgData, b, x, level, amgParam):
    # Multigrid cycle
    # adapted from Matlab code of Xiaozhe Hu, Tufts University
    # Clara De Paolis
    
    # parameters
    max_level = amgData[0]['max_level']
    n_presmooth = amgParam['n_presmooth']
    n_postsmooth = amgParam['n_postsmooth']
    cycle_type = amgParam['cycle_type']

    # coarsest level
    if level == max_level:
        x = scipy.sparse.linalg.spsolve((amgData[level]['L'] + 1.0e-12* scipy.sparse.eye(len(b), len(b))), b).reshape((len(x),1))
    else:
        # presmoothing
        x = support_scripts.forward_gs(amgData[level]['L'], b, x, amgData[level]['DL'], n_presmooth)

        # compute residual
        r = b - amgData[level]['L'].dot(x)

        # restriction
        r_c = amgData[level]['L'].dot(r)

        # coarse grid correction
        e_c = np.zeros((amgData[level+1]['L'].shape[0],1))

        if cycle_type=='TL':
            # coarse grid correction for two-level method here
            e_c = AMG_Cycle(amgData, r_c, e_c, level+1, amgParam)
        elif cycle_type=='V':
            # coarse grid correction for V-cycle here
            e_c = AMG_Cycle(amgData, r_c, e_c, level+1, amgParam)
        elif cycle_type=='W':
            # coarse grid correction for W-cycle here

    return x, k, err
for k in range(2):
    e_c = AMG_Cycle(amlData, r_c, e_c, level+1, amgParam)

    # prolongation
    x = x + amlData[level]["P"].dot(e_c)

    # postsmoothing
    x = support_scripts.backward_gs(amlData[level]["L"], b, x, amlData[level]["DU"], n_postsmooth)

return x

import numpy as np
import scipy.sparse, scipy.sparse.linalg

def assembleGraphLaplace(N):
    # Adapted from matlab code
    # Copyright (C) Xiaozhe Hu.

    e = np.ones(N)
    NN = N**2

    L1d = scipy.sparse.spdiags([-1*e, 2*e, -1*e], [-1,0,1], N, N)
    I = scipy.sparse.eye(N,N)

    L = scipy.sparse.kron(L1d, I) + scipy.sparse.kron(I, L1d)
    L = L - scipy.sparse.spdiags(L.diagonal(), 0, NN, NN)
    L = L + scipy.sparse.spdiags(-L.sum(axis=1).T, 0, NN, NN) #row sum

    return L

def backward_gs(A, b, x, DU, nsmooth):
    # Backward Gauss-Seidel smoother
    # Adapted from matlab code from @ Xiaozhe Hu, Tufts University

    #-----------------------
    # Step 1: Main loop
    #-----------------------
    for i in range(nsmooth):
        # GS iteration
        x += scipy.sparse.linalg.spsolve(DU, (b - A.dot(x))).reshape((len(x),1))

    return x

def forward_gs(A, b, x, DL, nsmooth):
    # Forward Gauss-Seidel smoother
    # Adapted from matlab code from @ Xiaozhe Hu, Tufts University

    #-----------------------
    # Step 1: Main loop
    #-----------------------
    for i in range(nsmooth):
        # GS iteration
        x += scipy.sparse.linalg.spsolve(DU, (b - A.dot(x))).reshape((len(x),1))

    return x
x += scipy.sparse.linalg.spsolve(DL, (b - A.dot(x))).reshape((len(x),1))
return x

def generate_unsmoothed_P(aggregation, num_agg):
    # Construct unsmoothed prolongation P
    # Adapted from matlab code from
    # @ Xiaozhe Hu, Tufts University
    n = len(aggregation)
p = scipy.sparse.csr_matrix((np.ones(n), (np.array(range(n)), aggregation)), shape=(n, num_agg))
return p

import numpy as np
import scipy.sparse

def form_aggregates(L):
    # Heavy edge Coarsening
    n = L.shape[0]
count = -1
aggregates = np.zeros(n)
for i in range(n):
    if aggregates[i]==0:
        (_, js, w) = scipy.sparse.find(-L[i]) #find edges and weights
e = np.where(w == max(w))[0] # e lists the indices that match the max weight
j = js[e[-1]]

    if aggregates[j] == 0:
        count += 1
        aggregates[i] = int(count)
        aggregates[j] = int(count)
    else:
        aggregates[i] = aggregates[j]

num_agg = count+1
return aggregates, num_agg