The Application of Algebraic Multigrid Methods to Solving Large Scale HodgeRank Problems

A THESIS SUBMITTED BY

Charles Colley

IN PARTIAL FULFILLMENT OF THE REQUIRMENTS FOR

Master of Science

in

Computer Science

Tufts University

February 2018

Advisors: Shuchin Aeron, Xiaozhe Hu

Abstract

In this thesis we consider unsmoothed aggregation algebraic multigrid preconditioners applied to graph ranking problems arising from the HodgeRank algorithm. We will discuss the HodgeRank algorithm's foundations after a brief discussion of common ranking methods and present an analysis of the UA-AMG method for solving graph Laplacians systems arising from the least squares problems,applying it as a preconditioner for conjugate gradient (CG) to achieve better performance. We also provide experiments comparing the LSRN, LSQR, and CG method (with and without UA-AMG as a preconditoner) on a collection of larger random graphs and a collection of real world network topologies to demonstrate the effectiveness of UA-AMG method for solving least squares problems on graphs.

Contents

List of Figures

List of Tables

List of Algorithms

0.1 Introduction

Social sciences have been interested in voting and rank aggregation problems for centuries, and have led to many interesting theories and theorems on the limitations of these problems[5]. Though the notion of deciding on the winner of an abstract winner may seem straight forward, there is a lot of nuance which can influence the winner simply by changing the method in with the votes are collected. Consider the following example of considering three voters their preference of colors.

Voter	1st preference	2nd preference	3rd preference
Voter 1	red	blue	green
Voter 2	blue	green	red
Voter 3	green	red	blue

Table 1: Three color preferences over three Voters

Each of our voters have voted on every alternative offered and each voter has a distinct preference for each color (no ties in preference). Now to consider a winner we'll consider each color pairwise and count the number of voters who prefer the first color to the second. With this method we find that each color is prefered to the the other colors by two voters, implying that there is no outright winner. This was a smaller example, but consider the following example of votes.

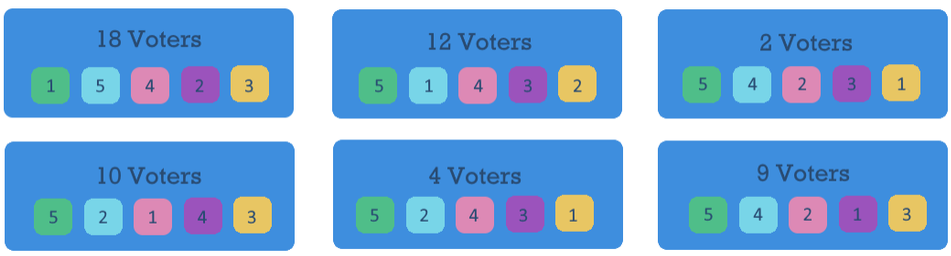

Figure 1: Collection of preferences over 5 colors, Each ballot represents preferences with the number of voters for that preference order.[4]

For this example we will consider a few different common methods of ranking votes; The Plurality method, Two round runoff, Instant runoff, and the Borda count. The most straight forward is the Plurality method, which simply considers the winner to be the the alternative with the most amount of first place votes, which when we consider the data we see that Green is the winner with 18 votes (points).

Alternative	Green	Blue	pink	purple	orange
Votes/Points	**18**	12	10	9	6

Table 2: Plurality Method

Next we look at the Two round run off method, which first looks for the top two candidates being ranked, if either of them has a majority (more than half the votes), then they are declared the winner, else a second election is held pitting them against each

other.

Alternative	Green	Blue	pink	purple	orange
Round 1 Point	**18**	12	10	9	6
Round 2 Point	18	**39**	n/a	n/a	n/a

Table 3: Two round runoff

In this case, instead of having a second vote, we can recount the votes as if there were only those two alternatives on the ballot. In this case, we see that Blue is the winner with 37 points. Next is the instant run off method, which instead of seeking out the alternative with the most votes, it finds the alternative with the least votes and eliminates it from the ranking. The votes are then tallied again without that alternative and this process is repeated until only one alternative remains.

Alternative	Green	Blue	pink	purple	orange
Round 1 Points	18	12	10	9	<u>6</u>
Round 2 Points	18	16	10	<u>9</u>	n/a
Round 3 Points	18	<u>16</u>	21	n/a	n/a
Round 4 Points	18	n/a	**34**	n/a	n/a

Table 4: Instant runoff

After collecting the votes together with this method, we see that pink comes out on top. Finally we consider the Borda count method, which for each vote, awards points to each alternative based off of the number of the number of alternatives it beats. In this example, the alternative that is ranked first on a ballot will receive 4 points, and the alternative ranked last will receive no points. The winner is the alternative with the most points.

Alternative	Green	Blue	pink	purple	orange
Round 1 Points	72	101	107	**136**	134

Table 5: Borda Count

When we tally the result together we find that the winner with this method is purple. Having considered all these different methods, we find that with the same data, which listed all the preferences for all voters the ultimate winner was decided by the voting method. This immediately raises the question of how do we determine the best way to rank votes for a general problem? For a specific problem there may be a preferred method to rank them, but in the general case it is unclear which method should always be used. With this in mind, we turn to the HodgeRank algorithm.

0.2 HodgeRank

Given our votes collected for a set of alternatives, we may ask ourselves what challenges will we encounter when trying to decide which alternative is "best". How do we even know that there will exist a ranking $\vec{r}$ that we can use to order the alternatives, and more importantly how will we assess that the ranking is appropriate? HodgeRank provides a systematic way to

answer those questions based on a graph built based on pairwise ratings. The first question we must ask ourselves is how to build a graph from the voting data that we have accrued.

0.2.1 Building Graphs

To outline our method of building graphs, we will start with how we build a graph for a single voter. For our particular voter $\alpha \in \Lambda$ we call $V_\alpha \subset V$ the set of alternatives that particular voter has given cardinal scores to. We also specify a rank comparison function $f^\alpha(i,j)$ for comparing voter α's preferences between alternative i and j.

In [6], the authors outline four potential rank comparison functions, due to the datasets being used for testing we focus our interest on the following two.

$$\text{Ratings Difference } f^\alpha(i,j) = R(\alpha,j) - R(\alpha,i), \tag{1}$$

$$\text{Ratings Ratio } f^\alpha(i,j) = \log(R(\alpha,j)) - \log(R(\alpha,i)). \tag{2}$$

These functions are used to convert the cardinal scores into pairwise comparisons (edge flows), thus creating a graph $G_\alpha = (V_\alpha, E_\alpha)$ over sets of alternatives. In our experiments for each voter, we form a clique over V_α with all possible $\binom{|V_\alpha|}{2}$ comparisons. This will not be our final graph as we must still accommodate multiple voters. Let Λ_{ij} denote the set of users who voted on alternatives i and j. Now we can define a graph for all the voters. Let $V = \cup_\alpha V_\alpha$ and $E = \cup_\alpha E_\alpha$ and define a weighted graph $G = (V, E, \omega)$ where the edge weight for an edge $e = (i,j) \in E$ is defined as $\omega_e = |\Lambda_{ij}|$. Based on the graph G, we introduce the edge flow f from the pairwise comparison functions f^α as following:

$$f(i,j) = \frac{1}{|\Lambda_{ij}|} \sum_{\alpha \in \Lambda_{ij}} f^\alpha(i,j).$$

This is simply a choice for our experiments, more sophisticated functions can be developed to better account for unrated or ties between alternatives. Ratings Difference (1) produces a network flow with the arithmetic mean of the differences of votes and Ratings Ratio (2) produces a network flow of the logarithmic geometric mean of the ratio of votes. Jiang et al. comment that (2) is more suited for data where the ratings of alternatives may be of different orders of magnitude, such as a currency exchange networks. For applications where the ratings function provides scores for all alternatives in the same order of magnitude, such as a star rating system, the difference function is more appropriate. So these graphs are the result of averaging the graphs induced by each voter, however this still uses Jiang et al's skew symmetric tensor formulation of the HodgeRank problem.

0.2.2 Orientation's Opportunity

From the skew symmetric functions f Hirani et al[7] noted a opportunity to simply the problem by taking advantage of the the symmetry of our f. We decide a default way to orient the edges, i.e., for each edge $(i,j) \in E$, we randomly assign i as the source node and

j as the sink node. Once a choice has been made, we will use a vector $\vec{f} \in \mathbb{R}^{|E|}$ to represent the edge flow constructed from the function f as $\vec{f_e} = f(i,j),\ e = (i,j) \in E$.

Note that, $f(j,i)$ will be ignored once the edge $e = (i,j)$ has assigned the orientation (i,j). There are two important things to keep in mind, first, our choice of orientation is arbitrary, but once we specify it, we must remain consistent as we will use it to define orientation on higher ordered structures such as triangles. We choose our orientation by having lower numbered vertices point towards higher numbered vertices. Second, having an orientation does not imply that the graph induced will be comprised of directed edges, in fact, the graphs will be undirected, with weights corresponding to the number of votes for both alternatives in the edge. By storing the number of voters for a particular pair of alternatives, we do not have to store a multi-graph when condense all the networks together. Instead we can use our vectorized edge flow function $\vec{f}$ which combine all the networks, and has the values of the m edges remaining. Thus $\vec{\omega} \in \mathbb{R}^m$ and it's important that $\vec{\omega}$ and $\vec{f}$ index the same edge.

We will soon discuss the operators over higher order simplexes, rather than just considering edges (1 simplexes). To generalize the notion of an orientation, we think of it as equivalence classes of permutations over the indices of the vertices within the simplexes being analyzed in the network. We maintain our skew symmetry by changing the sign when we move to an odd permutation of the simplex indices.

0.2.3 Origins of Inconsistency

The strongest feature that HodgeRank boasts is its ability to measure the quality of the ranking provided by the algorithm. Though, this begs the question of how we even define quality in the first place. When we consider a ranking problems over decisions which are inherently subjective, Condorcet's paradox[8] shows us that there are conditions where there can't exist a ranking in which everyone will be satiated. For an illustration of the connection between cycles and consistency, we will think of the networks with Jiang et al's framework momentarily. Consider a set of 2 voters $\Lambda = \{\alpha, \gamma\}$ and consider a set of 3 alternatives $V = \{A, B, C\}$. Using our function (1) to induce the edge flow, we will create the graphs though these graphs are directed, we can still create paths through the graph, reorienting

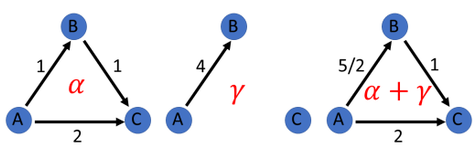

Figure 2: (left to right) Edge flows induced by two voters α and γ, and the combined network $\alpha + \gamma$ over three alternatives

the edges and negating the values as needed. Notice as we traverse the paths ABCA and ABA in the graphs induced by α and γ respectively, both their sums will be 0. In graphs α and γ it stands to reason that we could rank the alternatives from lowest to highest as ABC and CAB respectively (assuming unrated alternatives automatically lose out). But

the network from $\alpha + \gamma$ is less clear as the edge from A to B has a larger weight than the the edge from A to C, but B has an edge oriented towards to C, which would imply that B loses out to C. Though an exhaustive approach is possible for such a small problem set, real world problems will be much more intricate and an exhaustive analysis will be infeasible. We will want a quantity in the network to associate uncertainty in our ranking. When we traverse the path ABCA in the graph induced by $\alpha + \gamma$ we find that the sum will be equal to 1.5. Paths which do not sum to 0 are our indication that there may be inconsistencies in our voting dataset as the amount of scores raised should be equal to the amount of scores lowered in a consistent dataset. When we consider the rank comparison function (1) for a single user, we can see that because we use the differences of the ratings, a single user will have a consistent ranking. The ordering will be the result of sorting the alternatives by their rating. The magnitude of how inconsistent the data is, corresponds to how large the norm of the residual is from the least squares problems in HodgeRank. In the above example, only 2 and 3 dimensional simplexes (edges and triangles) are considered, but in practical problems, these simplexes may be many orders of magnitude larger, and it will become infeasible to search the network to find them.

0.2.4 Simplicial Complexes

Though so far we have only introduced vertices and edges, HodgeRank makes use of the more general framework of abstract simplicial complex (ASC). ASCes are a purely combinatorial method of describing a family of sets of vertices, which naturally generalize the notion of graph. We refer to the elements of the ASC, K, as the faces $\sigma \in K$, and we refer to the elements within a face $v \in \sigma$ as vertices. We define the dimension of a face as $\dim(\sigma) = |\sigma| - 1$ and the $\dim(K)$ is the largest dimension of any face in K.

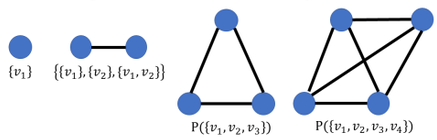

Figure 3: Instances of 0-3 dimensional simplexes. $\mathcal{P}$ denotes the power set.

This makes the simplest 0-dimensional faces the singleton sets of vertices $\{v\} \in K$, and the 1-dimensional faces, pairs of vertices $\{v_1, v_2\}$, or as we have been calling them, edges. In fact, a 1-dimensional ASC is mathematically equivalent to a graph. We can actually generalize all of the higher order structures with a general definition.

Definition 1 (Abstract Simplicial Complex) *K is an abstract simplicial complex if every $\sigma \in K$ then so does every non-empty subset of σ.*

With this framework we can go on to describe higher order structures, such as triangles and tetrahedrons, which are our 2 and 3-dimensional simplexes respectively. The power that comes with this view point are the chain complex we can form over the ASCs. We will form a bounded chain complex where the simplicial complexes are the simplicial complexes of different dimensions, and the homomorphisms between the simplicial complexes are called the boundary operators.

0.2.5 Boundary Operators

Though the ASC may seem to be more general than needed, the bounded chain complex that comes with it allows us to decompose the edge flow over the network to find our ranking. Each ith dimensional ASC is the ith chain in the structure, and the functions that allow us to traverse between chains are called the boundary operators. For our purposes we will think of the i th chain as a vector over the ith simplicial complex, and the boundary operators are expressed as matrices. The co-boundary operators are formed by taking the transpose.

$$\vec{0} \leftarrow C_0 \underset{\partial_1}{\leftarrow} C_1 \underset{\partial_2}{\leftarrow} C_2 \leftarrow \vec{0}.$$

Here $\vec{0}$ denotes the additive identity, which in the case of matrices is just an appropriately dimensioned zero matrix. To traverse backwards, we use vector space duality theorems to construct the closely related Co-chain.

$$\vec{0} \rightarrow C^0 \underset{\partial_1^T}{\rightarrow} C^1 \underset{\partial_2^T}{\rightarrow} C^2 \rightarrow \vec{0}.$$

Another important property of any of the (co-) boundary operators is that the application of consecutive boundary operators is the additive identity, in the general case this is written as $\partial_i \partial_{i+1} = 0,\ \forall i \in \mathbb{Z}$. First let us just consider 1-dimensional ASC. the boundary operator is nothing but the divergence operator (or the signed incidence matrix in the graph setting) defined as $\partial_1 : \mathbb{R}^{|E|} \mapsto \mathbb{R}^{|V|}$ defined as

$$(\partial_1)_{ij} = \begin{cases} -1, & \text{if } v_i \text{ is the source node in } e_j, \\ 1, & \text{if } v_i \text{ is the sink node in } e_j, \\ 0, & \text{else.} \end{cases}$$

In the current form, ∂_1 is actually the negative divergence and ∂_1^T is the gradient.

The second boundary operator is the oriented combinatorial curl operator put forward by Hirani et al [7] to improve upon Jiang et al's [6] original combinatorial curl. Here this operator will capture all cycles of length 3 (triangles), which will be thought of as a local curl of the graph. We can denote the number of cycles of length 3 in the network with $|T|$ and we will call T_j the jth triangle for $j \in [|T|]$. Note that for each one of the triangles in the network, we can apply a similar notion of orientation like we did for the edges, as we can move either clockwise or counter clockwise around the triangle's vertices. The oriented combinatorial curl $\partial_2 : \mathbb{R}^{|T|} \mapsto \mathbb{R}^{|E|}$ is defined as

$$(\partial_2)_{ij} = \begin{cases} 1, & \text{if } e_i \in T_j \text{ with same orientation as } T_j, \\ -1, & \text{if } e_i \in T_j \text{ with different orientation as } T_j, \\ 0, & \text{else.} \end{cases}$$

From these definitions we can show that $\partial_1 \partial_2 = 0$. We can continue to define boundary operators for higher dimension simplicial complexes,extending the chain complex as needed, but we will focus on ∂_1 and ∂_2 in this investigation.

0.2.6 Hodge Decomposition

With the definitions in place, we may now start to answer the hard questions. Most importantly the existence of the ranking we seek. With conditions that come with the chain complex, we are able to apply theorems of Hodge decompositions of the 1-chains (edges), with the help of the boundary operators. Lim [9] offers an excellent explanation of cohomology through a matrix algebra lense. These ideas provide the framework to prove the existence of the decomposition of the edge flow $\vec{f}$.

Definition 2 (Cohomology Group) *Given two matrices $\partial_1 \in \mathbb{R}^{|V|\times|E|}$ and $\partial_2 \in \mathbb{R}^{|E|\times|T|}$ satisfying the property $\partial_1\partial_2 = 0$ which is equivalent to $im(\partial_2) \subseteq ker(\partial_1)$, the cohomology group is the quotient vector space ${ker(\partial_1)}/{im(\partial_2)}$. Elements in ${ker(\partial_1)}/{im(\partial_2)}$ are referred to as the cohomology classes.*

With matrices ∂_1 and ∂_2 where $\partial_1\partial_2 = 0$, the 1-Hodge Laplacian is $L_1 = \partial_1^T\partial_1 + \partial_2\partial_2^T \in \mathbb{R}^{|E|\times|E|}$ and then $ker(\partial_1^T\partial_1 + \partial_2\partial_2^T) = ker(\partial_1) \cap ker(\partial_2^T)$. Furthermore the kernel of the Hodge Laplacian is isomorphic to the cohomology group, i.e., $ker(\partial_1) \cap ker(\partial_2^T) \cong {ker(\partial_1)}/{im(\partial_2)}$. Now, $\mathbb{R}^{|E|}$ has a Hodge decomposition,

$$\mathbb{R}^{|E|} = im(\partial_1^T) \oplus ker(L_1) \oplus im(\partial_2^T).$$

So for any $\vec{x} \in \mathbb{R}^{|E|}$ there exists a $\vec{r} \in \mathbb{R}^{|V|}$ and a $\vec{c} \in \mathbb{R}^{|T|}$, and $\vec{x}_h \in ker(L_1)$ such that we can write

$$\vec{f} = \partial_1^T\vec{r} + \partial_2\vec{c} + \vec{x}_h.$$

This proves for an arbitrary edge flow, that we will always be able to find the ranking vector $\vec{r}$ and the local consistency vector $\vec{c}$.

0.2.7 Least Squares Formulation

In order to find $\vec{r}$ and $\vec{c}$ in the Hodge decomposition of $\vec{f}$, we solve the following two weighted least squares problems,

$$\min_{\vec{r}\in\mathbb{R}^n} \|\vec{f} - \partial_1^T\vec{r}\|_W^2, \tag{3}$$

$$\min_{\vec{c}\in\mathbb{R}^m} \|\vec{f} - \partial_2\vec{c}\|_W^2, \tag{4}$$

where $W = \text{diag}\{\omega_e\} \in \mathbb{R}^{|E|\times|E|}$. The least squares problems are weighted because we apply orientation to the network to simplify and condense the edge flow $\vec{f}$. Note that these weights correspond to the number of users which voted for the two alternatives being compared by the edge. It is possible to write them as unweighted least squares problems, but that requires a multigraph view of the network.

Typically in (4) the second least squares problem is over the residual of the (3), but Jiang et al. [6] noted that because the components of the decompostion come from a direct sum, they're all mutually orthogonal in the standard inner product. This means that we can solve the least squares problems with the normal equations in parallel.

$$\partial_1 W \partial_1^T \vec{r} = \partial_1 W \vec{f}, \tag{5}$$

$$\partial_2^T W \partial_2 \vec{c} = \partial_2^T W \vec{f}. \tag{6}$$

Note that the linear system in (5) is nothing but the wighted graph Laplacian defined as $L = \partial_1 W \partial_1^T$. Though we will only focus our attention on solving (5), we believe that this work can be extended to solve (6) too.

0.2.8 Quality Control

Once we have our solutions, we note that because $\vec{r} \in im(\partial_1^T)$ then it will be curl free, because the curl of the gradient is $\vec{0}$. This implies that our solution $\vec{r}$ is acyclic, and we can use this for our ranking. In fact, we are only interested in an ordering, so we solve $\vec{r}$ up to an additive constant. The solution $\vec{c}$ is used to identify the quality of the ranking on a local scale. Because ∂_2 captures the inconsistency of 3-cycles in the graph (non-zero path weights), we use this to identify when a ranking is locally consistent, even if the residual of the solution to the global ranking may have a large norm. We determine how much inconsistency, and thus how reliable the ranking is by the norm of the residual from each of the least squares problems. The larger the norm, the more inconsistency is present in the dataset. When we look at the values of the triangles represented by $\vec{c}$, a large value of c_i implies that the ith 3-cycle will have a large path weight. In the case where the residual of (4) is quite large, we can define more boundary operators moving through higher order simplexes in order to capture more of the inconsistency.

0.3 Iterative Methods

To solve (5), we employ and compare UA-AMG method to the widely known incumbent conjugate gradient method.

0.3.1 Preconditioned Conjugate Gradient

Preconditioned Conjugate Gradient (PCG) is a well known Kyrlov subspace method for solving symmetric positive definite (SPD) matrix problems. Due to the recognition of this algorithm we will forgo an extensive analysis (which can be found in [10]) and only discuss the major features.

Conjugate Gradient offers excellent computational cost as the dominating cost of the algorithm is the mat-vec required at each iteration of the algorithm. Typically mat-vecs are $O(n^2)$ operations to compute, but for sparse matrix data structures we can reduce this to the number of non-zeros in the matrix, which we denote with nnz. Seeing as our experiments we will be using sparse matrices, we can reduce the computational cost of the algorithm from $O(nnz)$ to $O(n)$. We can also bound the A-norm $\|\cdot\|_A$ of the error at the ith iteration with

$$\|e_i\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \|e_0\|_A,$$

where κ denotes the condition number of the matrix A in the linear systems problem. With this we can see conjugate gradient is an excellent algorithm for a well conditioned matrix, but once the condition number increases then so will the number of iterations needed to reach a desired tolerance. When the condition number of the matrix becomes large, then experiments show that using UA-AMG as a preconditioner provide the best results (see Section 0.6 for details). The algorithm for the PCG is as shown in Algorithm 1.

Algorithm 1: Preconditioned Conjugate Gradient

Input: SPD MATRICES $A, M \in \mathbb{R}^{n\times n}$,
VECTORS $\vec{b}, \vec{x}_0 \in \mathbb{R}^n$, AND TOLERANCE $\epsilon \in \mathbb{R}$
$\vec{r}_0 = \vec{b} - A\vec{x}_0$, $\vec{z}_0 = M^{-1}\vec{r}_0$, $\vec{p}_0 = \vec{z}_0$, $k = 0$;
repeat
  $\alpha_k = \frac{\vec{r}_k^T \vec{z}_k}{\vec{p}_k^T A\vec{p}_k}$;
  $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$;
  $\vec{r}_{k+1} = \vec{r}_k - \alpha_k A\vec{p}_k$;
  if $\frac{\|r_k\|}{\|b\|} < \epsilon$ **then**
    break
  end
  $\vec{z}_{k+1} = M^{-1}\vec{r}_{k+1}$;
  $\beta_k = \frac{\vec{z}_{k+1}^T \vec{r}_{k+1}}{\vec{z}_k^T \vec{r}_k}$;
  $\vec{p}_{k+1} = \vec{z}_{k+1} + \beta_k \vec{p}_k$;
  $k = k + 1$;
until;
Output: $\vec{x}_{k+1}$

To use another matrix algorithm as a preconditioner, we will use them to solve the M matrix problems. i.e. $M\vec{z}_{k+1} = \vec{r}_{k+1}$ which gives $\vec{z}_{k+1} = M^{-1}\vec{r}_{k+1}$. For all of our experiments we use our AMG routines to implicitly apply M^{-1}. Though in the next section will discuss AMG as it stands, in practice it's always used as a preconditioner for a Krylov method. This is because you can prove that the convergence rate of PCG with a general linear systems solver will always be better than just the linear systems solver. Note that conjugate gradient without a preconditioner is obtained by setting $M = I$ where $I \in \mathbb{R}^{n\times n}$ is the identity matrix.

0.4 Algebraic Multigrid

AMG is a successive subspace correction method which creates a hierarchy of matrices to split the original Hilbert space into subspaces. Let us focus on AMG for graph problems, consider a graph $G = (V, E, \omega)$ where we want to solve a linear system of equations of the graph Laplacian which may come from the least squares problem (3). By using fast partitioning algorithms such as matching or maximal independent sets and condensing each partition into one single vertex, we can create a new graph with at least half the number of vertices in the graph. In the AMG community, the partitions are referred as aggregates, so we will use this terminology here on out. Continuing this process l steps, we will reach a point where the the lth graph has so few vertices that the matrix problem $A_l\vec{x}_l = \vec{b}_l$ can be

Algorithm 2: Algebraic Multigrid: Setup Phase

Input: $A \in \mathbb{R}^{n \times n}$,
MAX LEVELS AND MIN GRAPH SIZE: $max\_level, coarse\_size \in \mathbb{Z}^+$
$A_1 \leftarrow A$;
for $l \leftarrow 1$ **to** $max\_level - 1$ **do**
  if $size(A_l) \leq coarse\_size$ **then**
  Aggregates $\{\mathcal{A}\} \leftarrow$ `Form_Aggregates`(A_l);
  $P_l \leftarrow$ `Form_Prolongation_Operator`$(\{\mathcal{A}\})$;
  $A_{l+1} = P_l^T A_l P_l$;
end
Output: A_l and P_l

solved directly without any thought of computational cost. To improve efficiency we split the algorithm into a setup and solve phase (here we use V_cycle as our solver phase). The setup phase is responsible for populating a struct which holds all the multilevel hierarchy. Algorithm 2 shows setup phase for a general AMG method.The V_cycle solve phase is when we actually use the subspace correction methods to find out solutions iterations and is shown in Algorithm 3.

Algorithm 3: $[\vec{x}_l] =$ `V_cycle`$(A_l, \vec{x}_l, \vec{b}_l, l)$

if $l = max\_level$ **then**
  $\vec{x}_l = A_l^{-1}\vec{b}_l$;
else
  $\vec{x}_l \leftarrow$ `pre-smoothing`$(A_l, \vec{x}_l, \vec{b}_l, l)$
  $\vec{r}_{l+1} = P_l^T \vec{r}_l$, $\vec{r}_l = \vec{b}_l - A_l\vec{x}_l$
  $[\vec{e}_{l+1}] =$ `V_cycle`$(A_{l+1}, \vec{0}, \vec{r}_{l+1}, l+1)$
  $\vec{x}_l = \vec{x}_l + P_l\vec{e}_{l+1}$
  $\vec{x}_l \leftarrow$ `post-smoothing`$(A_l, \vec{x}_l, \vec{b}_l, l)$;

The pre- and post-smoothing step in Algorithm 3 is applied before each time we restrict the error to a coarse level and after we prolongate the error back to the fine level. It is used to remove the high oscillatory errors. Variety of iterative methods could be adopted for smoothing, such as Jacobi method, Gauss-Seidel method and Richardson method. For all the experiments in this paper, we use forward Gauss-Seidel as pre-smoother and backward Gauss-Seidel as post-smoother to keep symmetry.

Prolongation Operator

The primary difference between the AMG routines that we run compared to the AMG routines in the experiments of Hirani et al[7] is the use of piecewise constant prolongation as opposed to the smoothed prolongation provided in PyAMG[11]. Focusing on the case that A is just graph Laplacian corresponding to a given graph $G = (V, E)$, we form a non-overlapping partition of the vertices by running a matching or MIS algorithm on the graph. Assuming that we form $n_a \in \mathbb{Z}^+$ aggregates such that $V = \bigcup_{j=1}^{n_a} V_j$ where $V_i \cap V_j = \emptyset$, $\forall\, i, j \in [n_a]$. Then we then build the prolongation P as $P_{ij} = 1$, if v_i is in aggregate j. Otherwise, $P_{ij} = 0$.

This piecewise constant prolongation, which usually is referred as unsmoothed aggregation prolongation, is better for graph problems because it maintains the structure of the

graph Laplacian on all of the coarse levels.

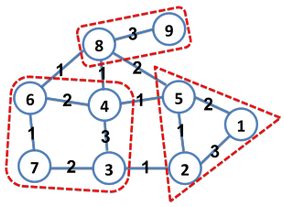

Figure 4: Aggregates: $\{1,2,5\}$, $\{3,4,6,7\}$, and $\{8,9\}$

$$L_f = \begin{pmatrix} 5 & -3 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ -3 & 5 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 6 & -3 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -3 & 7 & -1 & -2 & 0 & -1 & 0 \\ -2 & -1 & 0 & -1 & 6 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & -2 & 0 & 4 & -1 & -1 & 0 \\ 0 & 0 & -2 & 0 & 0 & -1 & 3 & 0 & 0 \\ 0 & 0 & 0 & -1 & -2 & -1 & 0 & 7 & -3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 3 \end{pmatrix}$$

Letting L_f denote the fine graph Laplacian corresponding to the graph shown in Figure 4. And we define the prolongation according to the aggregates $\{1,2,5\}$, $\{3,4,6,7\}$, and $\{8,9\}$ as following,

$$P^T = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$L_c = P^T L P = \begin{pmatrix} 4 & -2 & -2 \\ -2 & 4 & -2 \\ -2 & -2 & 4 \end{pmatrix}.$$

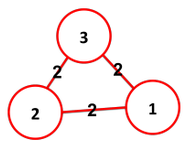

Figure 5: Coarse level graph

Then compute $L_c = P^T L_f P$, which is the graph Laplacian of the coarse graph by condensing each aggregate into one vertex as shown in Figure 5,

We can observe from the aforementioned example that the total connection between aggregations as well as the graph structure are preserved after the transformation. In fact, if L_c is easy to invert, we solve it on the coarse level and prolongate the coarse level correction back to the original fine level. This process is called Two-level UA-AMG method (Algorithm 3 with $\texttt{max\_level} = 2$). using B_{pre} and B_{post} to denote the pre and post smoothing operators we can construct the error iteration matrix $E = (I - B_{post}L)(I - PL_c^{-1}P^T L)(I - B_{pre}L)$. The formulation for this method is derived in the appendix. The following theory concerns the convergence rate of using UA-AMG methods alone as iterative method for solving $A\vec{x} = \vec{b}$, which is independent of the condition number of A.

Theorem 1 ([12]) *The convergence rate of two-level UA-AMG method for solving graph Laplacian corresponding to the graph $G = (V, E, \omega)$, $\|E\|_L = 1 - \frac{1}{\mathcal{K}}$, where*

$$\mathcal{K} \leq C \max_{1 \leq i \leq n_a} \{Diam(G_i)\} \max_{1 \leq i \leq n_a} \{Vol(G_i)\} \left(\min_{\substack{e \in E_i \\ 1 \leq i \leq n_a}} \omega_e \right)^{-1}.$$

Here $G_i = (V_i, E_i, \omega_i)$ is the local graph corresponding to aggregate V_i, $Diam(G_i)$ is the diameter of subgraph G_i and $Vol(G_i)$ is the sum of degrees of all vertices in G_i.

Complexity Analysis

Assuming that on each level of the AMG algorithm, the aggregate forming routine forms aggregates of size as least $c \in \mathbb{Z}^+$ and the algorithm on each level does $O(n^k)$ work then we can use a geometric series to compute the total work needed to complete one V-cycle routine in the multigrid algorithm. If our aggregates are at size at least c then we can estimate the number of levels the V-cycle will take will be at most $log_c n$. So we can compute the work

as $\sum_{i=1}^{\log_c n} ((\frac{1}{c})^{i-1}n)^k = \sum_{i=0}^{\log_c n-1} n^k(\frac{1}{c^k})^i$. Since we must have $c > 1$ for the subspaces to be smaller in size then its predecessor we can see that $\frac{1}{c} < 1 \Rightarrow \frac{1}{c^k} < 1$ which means that we can apply the geometric sum formula.

$$\sum_{i=1}^{log_c n-1} n^k(\frac{1}{c^k})^i = n^k \frac{1-(\frac{1}{c})^{k\log_c n}}{1-\frac{1}{c^k}} = n^k \frac{1-(\frac{1}{c})^{\log_c n^k}}{1-\frac{1}{c^k}}$$
$$= n^k \frac{1-(\frac{1}{n^k})}{1-\frac{1}{c^k}} = \frac{n^k-1}{1-\frac{1}{c^k}} \leq (\frac{1}{1-\frac{1}{c^k}})n^k$$

This analysis shows that the set up phase and the solving phase combined will yield an $O(n^k)$ algorithm.

During the set up phase of AMG, the algorithm constructs the hierarchy of coarser graphs by running vertex partitioning algorithms such as maximal independent set algorithms, or vertex matching. Both of these aggregation algorithms have greedy or randomized algorithms that run in at most $O(n)$ time. For our smoothing routines, the amount of work we do can be expressed as $O(n^k)$. For a sparse graph the complexity of the smoothers will be $k = 1$. For dense graph, we may have $k = 2$, which gives $O(n^2)$ algorithm. This is the best one can expect without using some special preprocessing on the graph such as sparsification.

0.5 Data

0.5.1 Random Graphs

For the experiments conducted on random graphs we will consider the Erdös Réyni[13] and Watts Strogatz[14] random graphs. Each of these graphs offer different emergent phenomena that resembles real world graphs. Heuristically we find that these random graphs typically lead to quite well conditioned adjacency matrices. To emphasize UA-AMG's robustness for least squares problems on graphs, we scale the edge weights uniformly over a range of e^{-k} to e^k for $k \in \mathbb{Z}$. This drastically increases the condition number of the graph, which enables us to see the limitations of conjugate gradient. When testing with the Watts Strogatz model, we set $\beta = 0$ for one of the experiments in order to produce a ring graph, which will have a large condition number. These results may be extended to other lattice graphs which are common graphs from discretizations for partial differential equations.

0.5.2 MovieLens [1]

The MovieLens data set is a collection of movies rated by voters collected by the Grouplens team at the University of Minnesota. The group offers data of different sizes with different metadata associated with each movie. The set used for our experiments was over 20 million ratings for $27,000$ movies by $138,000$ users. In order to understand the scalability of the algorithms for different sized real world problems, we test over subsections of the dataset. We also use Ratings Difference function, f_{rd}, and Ratings Ratio function, f_{rr}, to create

the edge flow in hopes of potentially highlighting differences between the edge flows the functions induce.

0.5.3 Yahoo Webscope [2]

This data set comes from anonymous user data from Yahoo! Messenger. It is a collection of edges representing communication from users to one another over a phone or computer. The entire graph unweighted and undirected with $1,878,736$ users and $4,079,161$ edges. Because we're interested in the topology of real world graphs, we use this as a template and alter the problem to produce a directed weighted graph. For each user, an orientation is imposed from the lower numbered vertex to the higher number vertex, and a weight randomly chosen from a uniform distribution of 0 to $1,000$ represents a hypothetical friendship score, and HodgeRank could be used to rank the users based off of popularity. We use this model as a frame for a real world application and to increase the difficulty of solving the graph problems for our numerical experiments, rather than to investigate the most popular user in the dataset.

0.5.4 DREAM Networks[3]

DREAM Challenges is a nonprofit company which specializes in hosting open source machine learning challenges for biomedical applications. We use networks provided for the 2016 Disease Module Identification DREAM Challenge. The networks in the challenge represent anonymized gene and protein interactions or associations. Though the original purpose of the datasets were for clustering problems, the topology of the underlying biological data is still very useful for understanding how least squares on these types of networks may behave. Because these graphs do not have any inherent rating data, we use a random vector to simulate a potential edge flow over the networks.

0.6 Experiment Results

The following experiments are conducted based on UA-AMG package implemented in MATLAB. Note that number k in column Weights represents that the edge weight is uniformly distributed within $[\exp(-k), \exp(k)]$. And if some entries are denoted as '-', this means that the method does not converge when maximal number of iterations is reached (here we set it to be 3000).

As mentioned above, we compare the performances between plain CG (denoted by "CG") and UA-AMG preconditioned CG (denoted by "CG+AMG") for solving the normal equation (5). The following is a list of components used in our UA-AMG implementation,

- Aggregation method: Heavy Edge Coarsening [15]
- Maximum level: 10
- Pre-smoothing method: forward Gauss-Seidel

- Post-smoothing method: backward Gauss-Seidel

We also compare with LSQR and LSRN for solving the original least squares problem (3) directly on different graphs. We use MATLAB build-in function `lsqr` for testing LSQR. For the implementation of LSRN, we use MATLAB build-in function `svds` for the singular value decomposition (SVD) of sparse matrices and LSQR as the inner iterative solver. For all the iterative methods, we use the stopping criterion $\|\vec{b} - L\vec{x}^k\|/\|\vec{b}\| \leq 10^{-10}$. The CPU time (in seconds) reported in the tables includes both setup phase and solver phase. Moreover, "-" means the method fails to converge within $3,000$ iterations or $1,000$ seconds.

All the experiments are conducted in MATLAB on a Dell workstation with 3.50 GHz Intel(R) Xeon(R) CPU and 256 GB RAM. These experiments we published in the proceedings from the GABB workshop of the 2017 IPDPS conference [16].

0.6.1 Random Graph Experiments

For Erdös-Réyni random graph, we set edge probability equals to $(2\ln|V|)/|V|$ to ensure the connectivity of the graph. For the Watts-Strogatz graph, each vertex has rewiring probability β to connect to its K neighbors. We set $\beta = 0$ to generate all the ring lattice graphs and $\beta = 1$ to generate all the Watts-Strogatz random graphs. The results of the random graphs are reported in Table 6, 8, and 7, respectively. Each set of parameters is repeated 5 times and the results are averaged.

Table 6: Erdös-Reyni random graph with uniformly distributed edges weights

Info.			CG		CG + AMG		LSQR		LSRN	
# Nodes	# Edges	Weights	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)
4,000	67,194	$[e^{-7}, e^{7}]$	97	0.15	10	0.10	81	0.07	66	33.56
4,000	67,194	$[e^{-10}, e^{10}]$	144	0.25	11	0.10	148	0.12	66	33.94
4,000	67,194	$[e^{-15}, e^{15}]$	578	1.78	12	0.10	429	0.28	67	34.35
8,000	143,134	$[e^{-10}, e^{10}]$	177	0.23	11	0.38	195	0.26	67	216.02
8,000	143,134	$[e^{-15}, e^{15}]$	506	0.54	13	0.41	466	0.54	67	175.96
8,000	143,134	$[e^{-20}, e^{20}]$	965	1.15	14	0.46	1228	1.53	68	204.38
16,000	311,086	$[e^{-20}, e^{20}]$	1185	1.21	14	0.56	1350	3.29	-	-
32,000	664,866	$[e^{-20}, e^{20}]$	1562	3.70	14	1.10	1533	9.17	-	-
64,000	1,417,062	$[e^{-20}, e^{20}]$	1965	19.36	16	2.45	-	-	-	-
128,000	3,008,518	$[e^{-20}, e^{20}]$	2195	27.81	16	7.38	-	-	-	-

Table 7: Watts-Strogatz graph with uniformly distributed edge weights

Info.			CG		CG + AMG		LSQR		LSRN	
# Nodes	# Edges	Weights	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)
4,000	20,000	$[e^{-7}, e^{7}]$	122	0.08	11	0.06	123	0.10	67	20.84
4,000	20,000	$[e^{-10}, e^{10}]$	266	0.10	13	0.08	327	0.19	67	22.04
4,000	20,000	$[e^{-15}, e^{15}]$	983	0.25	14	0.09	1085	0.54	67	23.56
8,000	40,000	$[e^{-7}, e^{7}]$	144	0.13	12	0.17	28	0.17	67	143.22
8,000	40,000	$[e^{-10}, e^{10}]$	296	0.17	13	0.18	396	0.39	68	138.74
8,000	40,000	$[e^{-15}, e^{15}]$	1095	0.57	15	0.20	1382	1.25	69	141.48
16,000	80,000	$[e^{-15}, e^{15}]$	1670	1.05	15	0.23	1739	2.67	-	-
32,000	160,000	$[e^{-15}, e^{15}]$	1924	5.61	16	0.89	2241	10.54	-	-
64,000	320,000	$[e^{-15}, e^{15}]$	2564	44.92	16	2.34	2688	63.57	-	-
128,000	640,000	$[e^{-15}, e^{15}]$	-	-	16	3.11	-	-	-	-

Table 8: Watts-Strogatz ring lattice graph (average degree = 10)

Info.		CG		CG + AMG		LSQR		LSRN	
# Nodes	# Edges	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)
2^{10} =1,024	5,210	135	0.03	25	0.04	135	0.03	135	0.05
2^{12} =4,096	20,480	517	0.17	25	0.13	517	0.18	517	0.25
2^{14} =16,384	81,920	2043	1.10	27	0.41	2043	1.47	2041	2.26
2^{16} =65,536	327,680	-	-	27	1.82	-	-	-	-
2^{18} =262,144	1,310,720	-	-	26	6.67	-	-	-	-

0.6.2 Real World Network Experiments

We pre-process the real world network matrices so that they are all undirected, non-negative weighted and we only analyze the largest connected component of each graph. The results are reported in Table 9 and 10, respectively.

Table 9: Subsections of MovieLens Data

Info.			CG		CG + AMG		LSQR	
# Nodes	# Edges	$f^{\alpha}(i,j)$	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)
1,337	646,157	(1)	62	0.13	4	0.09	63	0.76
1,337	660,168	(2)	60	0.12	4	0.09	62	0.78
2,674	2,858,886	(1)	75	0.63	4	0.17	78	5.48
2,674	2,910,696	(2)	73	0.59	7	0.43	77	5.74
4,011	6,575,098	(1)	117	1.89	8	0.92	124	22.24
4,011	6,692,123	(2)	110	2.24	5	1.34	123	29.46
5,348	11,489,665	(1)	130	4.77	11	3.02	136	48.11
5,348	11,705,936	(2)	118	5.68	7	2.14	135	52.40
6,685	17,618,992	(1)	126	10.86	6	3.67	134	76.64
6,685	17,978,210	(2)	129	8.97	6	4.12	134	75.71
8,022	24,958,100	(1)	143	11.53	11	8.23	161	121.25
8,022	25,498,371	(2)	154	13.08	13	6.57	162	131.67

Table 10: DREAM challenge protein networks and Webscope network

Info.		CG		CG + AMG		LSQR	
# Nodes	# Edges	# Iter	Time(s)	# Iter	Time(s)	# Iter	Time(s)
DREAM challenge network							
17,397	2,232,405	794	37.10	9	8.30	727	21.89
12,420	397,309	461	12.39	10	4.60	454	8.67
5,254	21,826	527	2.66	11	1.15	533	3.51
12,588	1,000,000	193	5.60	6	2.50	191	4.95
14,679	1,000,000	943	31.44	10	5.14	871	18.51
10,405	4,223,606	1024	20.04	9	2.50	953	88.35
Webscope network							
1,878,736	8,158,322	1497	473.50	33	46.75	1534	248.49

0.7 Analysis

From the results presented in Section 0.6, we see that UA-AMG preconditioned CG outperforms all the other iterative methods. In particular, for large-scale and ill-conditioned graphs, UA-AMG can achieve speedup around 1.3-17 times. This demonstrates the robustness and effectiveness of the UA-AMG for solving least squares problems on graphs, especially for large-scale real world networks.

We believe this comes from the fact that all of the networks we experimented with were expected to be ill conditioned, or at least not as well conditioned as normal random graphs. Though we discuss the results of UA-AMG preconditioned conjugate gradient, instead of just a pure AMG algorithm, these results indicate the robustness of UA-AMG algorithms. We also do not distinguish the setup up times from the run times of the V_cycles implemented, which for small graphs leads to CG running in less time, but for larger random graphs and real world networks is no longer true. It should be noted that a direct comparison of the timings of experiments between our paper and Hirani et al's [7] should be taken with a grain of salt, as the hardware and software used for the experiments varied. However the results of our scalability experiments still stand regardless of the differences in hardware.

0.7.1 Random Graphs

Table 6 and 7 show the results on the random graphs. We see that UA-AMG converges for all cases. Moreover, as we increase the size of the graphs and/or the scaling of the edge weights, the number of iterations of UA-AMG only increases moderately, as opposed to CG, LSQR shows a large increase in the number of iterations (more than $1,000$ iterations). For LSRN, although the number of iterations only increase slightly, due to the SVD involved, the CPU time is much longer then UA-AMG and, for graphs larger than $10,000$, it could not finish within a reasonable time. In Table 8, we test on the Watts-Strogatz lattice graphs by increasing the number of vertices, and the number of iterations of UA-AMG only increase slightly, unlike CG and LSQR.

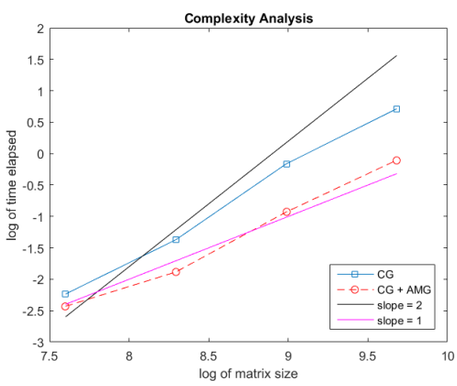

Figure 6: Computational complexity for Watts-Strogatz ring lattice graphs

Figure 6 shows a log-log plot of matrix size versus solving time using CG and CG preconditioned with UA-AMG. We can see that CG preconditioned with UA-AMG gives nearly linear computational complexity while CG is closer to quadratic complexity.

0.7.2 Real World Networks

Though the MovieLens Data in Table 9 is for unweighted graphs, with fewer than the total number of votes in the entire data set, we still see the benefits of UA-AMG over other iterative method (we did not consider LSRN for real work networks due to its unrealistically long running time). Though the runtimes of other iterative methods do not grow as quickly as they did for the ill-conditioned random graphs, we still see by the graph over $8,000$

movies that UA-AMG runs in half the time of CG. We would expect the trend to grow even larger as we weight the edges by the number of votes for a particular pair of alternatives and consider more alternatives for constructing the graphs.

We don't see very much variation between the graphs induced by Ratings Difference function (1) and Ratings Ratio function (2) which is promising as we would hope that a poor choice of f would not lead to significant differences in performance.

The DREAM networks (first 6 rows) and Yahoo Webscope data (last row) in Table 10 showcase typical real world problems. The Yahoo Webscope data is by far the largest graph that we experimented on, which shows that UA-AMG preconditioned CG method is at least 1.3 times faster compared to all other methods. The sixth graph in Table 10, which has the highest average degree, has UA-AMG beating CG by a factor of 8. Again all of these networks do not have any weights on the edges. In general we would expect even better comparison results for UA-AMG preconditioned CG as we scale the edges with weights which may come from meta data on the edges or the vertices.

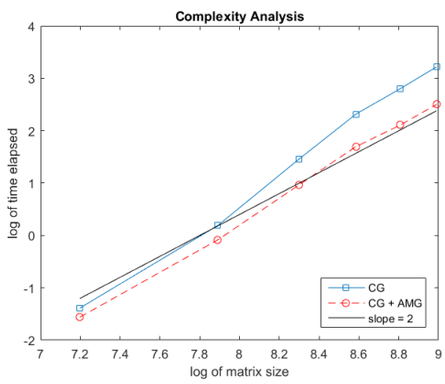

Figure 7: Computational complexity for MovieLens graphs

Again, we analyze the log-log plot of matrix size versus solving time using CG and CG preconditioned with UA-AMG in Figure 7. Since MovieLens data is dense, the best computational complexity we can expect is $O(|V|^2)$, which is what we observe, yet UA-AMG still outperforms CG. These results indicate that even for real world networks, UA-AMG methods should be thought of as a robust alternative to traditional iterative methods for problems which need to take scalability into consideration.

0.8 Conclusion

We have shown that unsmoothed aggregation AMG routines are a class of scalable matrix algorithms for the first least squares problems (3) on graphs arising from HodgeRank. We hope that this work would draw attention to these methods for graph problems as we believe that they should be seen as an important tool. We tested UA-AMG method for a variety of random graphs and real world networks. The numerical results show that the UA-AMG is robust among different networks, even when the corresponding graph Laplacians is highly ill-conditioned. Though the list was far from exhaustive, the variation between rating data, biological, and internet communication graph topologies is encouraging to believe that UA-

AMG will be successful on networks stemming from other applications.

0.9 Appendix: Error Iteration Matrices

0.9.1 Derivation of Gauss Seidel Error Iteration Matrix

to solve $Ax = b$, Gauss Seidel is a splitting method separating the matrix into the upper strictly triangular portion U and lower triangular portion the of the matrix L' such that any matrix $A \in \mathbb{R}^{n\times n} = U + L'$. We can also write L' as the diagonal D and a strictly lower triangular matrix L. Thus we get $A = D + L + U$. Each iteration of Gauss Seidel computes x_{k+1} by

$$x_{k+1} = (D + L)^{-1}(b - Ux_k).$$

Note this is referred to as forward Gauss Seidel because we use forward substitution to solve the $(D + L)$ inverse problem. An iteration of backwards Gauss Seidel is computed by

$$x_{k+1} = (D + U)^{-1}(b - Lx_k).$$

To write out the error iteration matrix we will focus on an iteration of forwards Gauss Seidel as the derivation for backwards Gauss Seidel follows in a similar fashion. Let $e_k = (x - x_k)$ denote the error at the kth step.

$$\begin{aligned}
x_{k+1} &= (D + L)^{-1}(b - Ux_k) \\
x_{k+1} &= (D + L)^{-1}(Ax - Ux_k) \\
x_{k+1} &= (D + L)^{-1}((D + U + L)x - Ux_k) \\
x_{k+1} &= (D + L)^{-1}((D + L)x + Ux - Ux_k) \\
x_{k+1} &= (D + L)^{-1}((D + L)x + U(x - x_k)) \\
x_{k+1} &= \quad x + (D + L)^{-1}Ue_k \\
x_{k+1} - x &= (D + L)^{-1}Ue_k \\
e_{k+1} &= -(D + L)^{-1}Ue_k \\
e_{k+1} &= -(D + L)^{-1}(A - (D + L))e_k \\
e_{k+1} &= (I - (D + L)^{-1}A)e_k
\end{aligned}$$

This yields the error iteration matrix as $E_{FGS} = (I - (D + L)^{-1}A)$ with the smoothing operator $B_{FGS} = (D + L)^{-1}$. Using a similar derivation we would compute an error interation matrix for backwards Gauss Seidel as $E_{BGS} = (I - (D + U)^{-1}A)$ with smoothing operator $B_{BGS} = (D + U)^{-1}$

0.9.2 Two Level UA-AMG Error Iteration Matrix Derivation

In this section we will consider a two level unsmoothed aggregation AMG routine to solve $Ax = b$ using a presmoother B_{pre},a post smoother B_{post}, and a prolongation matrix P. On the coarse level we will compute the error on the coarse grid by inverting the coarse matrix rather than applying a smoother.

Algorithm 4: $\vec{x}_{k+1} = \texttt{Two\_Level\_AMG}(A, \vec{x}_k, \vec{b})$

$\vec{u}_1 \leftarrow \vec{x}_k + B_{pre}(\vec{b} - A\vec{x}_k)$
$\vec{r}_f \leftarrow \vec{b} - A\vec{u}_1$
$\vec{r}_c \leftarrow P^T\vec{r}_f$
$\vec{e}_c \leftarrow A_c^{-1}\vec{r}_c$
$\vec{u}_2 \leftarrow \vec{u}_1 + P\vec{e}_c$
$\vec{x}_{k+1} \leftarrow \vec{u}_2 + B_{post}(\vec{b} - A\vec{u}_2)$

With this algorithm we can construct the full iteration out as

$$
\begin{aligned}
\vec{x}_{k+1} &= \vec{u}_2 + B_{post}(\vec{b} - A\vec{u}_2) = B_{post}\vec{b} + (I - B_{post}A)\vec{u}_2 = B_{post}\vec{b} + (I - B_{post}A)(\vec{u}_1 + P\vec{e}_c) \\
&= B_{post}\vec{b} + (I - B_{post}A)(\vec{u}_1 + P(A_c^{-1}\vec{r}_c)) = B_{post}\vec{b} + (I - B_{post}A)(\vec{u}_1 + P(A_c^{-1}P^T\vec{r}_f)) \\
&= B_{post}\vec{b} + (I - B_{post}A)(\vec{u}_1 + PA_c^{-1}P^T(\vec{b} - A\vec{u}_1)) \\
&= B_{post}\vec{b} + (I - B_{post}A)(PA_c^{-1}P^T\vec{b} + (I - PA_c^{-1}P^TA)\vec{u}_1) \\
&= B_{post}\vec{b} + (I - B_{post}A)(PA_c^{-1}P^T\vec{b} + (I - PA_c^{-1}P^TA)(\vec{x}_k + B_{pre}(\vec{b} - A\vec{x}_k)))) \\
&= B_{post}\vec{b} + (I - B_{post}A)(PA_c^{-1}P^T(A\vec{x}) + (I - PA_c^{-1}P^TA)(\vec{x}_k + B_{pre}((A\vec{x}) - A\vec{x}_k)))) \\
&= B_{post}\vec{b} + (I - B_{post}A)(PA_c^{-1}P^TA\vec{x} - PA_c^{-1}P^TA\vec{x}_k + \vec{x}_k + B_{pre}(A(\vec{e}_k)) - (PA_c^{-1}P^TA)(B_{pre}(A(\vec{e}_k)))) \\
&= B_{post}\vec{b} + (I - B_{post}A)(\vec{x}_k + PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
&= B_{post}(A\vec{x}) - B_{post}A\vec{x}_k + \vec{x}_k + (I - B_{post}A)(PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
\vec{x}_{k+1} &= \vec{x}_k + B_{post}A(\vec{e}_k) + (I - B_{post}A)(PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
\vec{x}_{k+1} - x &= \vec{x}_k - x + B_{post}A(\vec{e}_k) + (I - B_{post}A)(PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
-\vec{e}_{k+1} &= -\vec{e}_k + B_{post}A(\vec{e}_k) + (I - B_{post}A)(PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
\vec{e}_{k+1} &= \vec{e}_k - B_{post}A(\vec{e}_k) - (I - B_{post}A)(PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
&= \vec{e}_k - B_{post}A(\vec{e}_k) - (PA_c^{-1}P^TA\vec{e}_k - PA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{pre}A\vec{e}_k) \\
&\quad + B_{post}APA_c^{-1}P^TA\vec{e}_k - B_{post}APA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{post}AB_{pre}A\vec{e}_k \\
&= \vec{e}_k - B_{post}A(\vec{e}_k) - PA_c^{-1}P^TA\vec{e}_k + PA_c^{-1}P^TAB_{pre}A\vec{e}_k - B_{pre}A\vec{e}_k \\
&\quad + B_{post}APA_c^{-1}P^TA\vec{e}_k - B_{post}APA_c^{-1}P^TAB_{pre}A\vec{e}_k + B_{post}AB_{pre}A\vec{e}_k \\
&= (I - B_{post}A + B_{post}APA_c^{-1}P^TA - B_{post}APA_c^{-1}P^TAB_{pre}A + B_{post}AB_{pre}A \\
&\quad - PA_c^{-1}P^TA + PA_c^{-1}P^TAB_{pre}A - B_{pre}A)\vec{e}_k \\
&= (I + B_{post}A(-I + PA_c^{-1}P^TA - PA_c^{-1}P^TAB_{pre}A + B_{pre}A) - PA_c^{-1}P^TA + (PA_c^{-1}P^TA - I)B_{pre}A)\vec{e}_k \\
&= (-B_{post}A(I - PA_c^{-1}P^TA + (I - PA_c^{-1}P^TA)B_{pre}A) + I - PA_c^{-1}P^TA - (I - PA_c^{-1}P^TA)B_{pre}A)\vec{e}_k \\
&= ((I - PA_c^{-1}P^TA)(I - B_{pre}A) - B_{post}A((I - PA_c^{-1}P^TA)(I - B_{pre}A))+)\vec{e}_k \\
&= ((I - B_{post}A)(I - PA_c^{-1}P^TA)(I - B_{pre}A))\vec{e}_k
\end{aligned}
$$

This yields an error iteration matrix $E = (I - B_{post}A)(I - PA_c^{-1}P^TA)(I - B_{pre}A)$.

0.9.3 Two Level UA-AMG Smoothing Operator Derivation

with E, the fact that $A\vec{e}_k = \vec{r}_k$, the kth residual, we can compute the smoothing operator B by considering

$$\begin{aligned}
\vec{e}_{k+1} &= (I - B_{post}A)(I - PA_c^{-1}P^TA)(I - B_{pre}A)\vec{e}_k \\
&= (I - B_{post}A)(I - PA_c^{-1}P^TA)(\vec{e}_k - B_{pre}A\vec{e}_k) \\
&= (I - B_{post}A)(\vec{e}_k - B_{pre}\vec{r}_k - PA_c^{-1}P^TA\vec{e}_k + PA_c^{-1}P^TAB_{pre}(\vec{r}_k)) \\
&= (I - B_{post}A)(\vec{e}_k - B_{pre}\vec{r}_k - PA_c^{-1}P^T(\vec{r}_k) + PA_c^{-1}P^TAB_{pre}(\vec{r}_k)) \\
&= \vec{e}_k - B_{pre}\vec{r}_k - PA_c^{-1}P^T\vec{r}_k + PA_c^{-1}P^TAB_{pre}\vec{r}_k - B_{post}(\vec{r}_k) \\
&\quad + B_{post}AB_{pre}\vec{r}_k + B_{post}APA_c^{-1}P^T\vec{r}_k - B_{post}APA_c^{-1}P^TAB_{pre}\vec{r}_k \\
&= \vec{e}_k - (B_{pre} + B_{post} - B_{post}AB_{pre} + PA_c^{-1}P^T - PA_c^{-1}P^TAB_{pre} \\
&\quad - B_{post}APA_c^{-1}P^T + B_{post}APA_c^{-1}P^TAB_{pre})\vec{r}_k \\
&= \vec{e}_k - (B_{pre} + B_{post} - B_{post}AB_{pre} \\
&\quad + PA_c^{-1}P^T(I - AB_{pre}) - B_{post}A(PA_c^{-1}P^T)(I - AB_{pre}))\vec{r}_k \\
\vec{e}_{k+1} &= \vec{e}_k - (B_{pre} + B_{post} - B_{post}AB_{pre} + (I - B_{post}A)(PA_c^{-1}P^T)(I - AB_{pre}))\vec{r}_k \\
\vec{x} - \vec{x}_{k+1} &= (\vec{x} - \vec{x}_k) - (B_{pre} + B_{post} - B_{post}AB_{pre} + (I - B_{post}A)(PA_c^{-1}P^T)(I - AB_{pre}))\vec{r}_k \\
\vec{x}_{k+1} &= \vec{x}_k + (B_{pre} + B_{post} - B_{post}AB_{pre} + (I - B_{post}A)(PA_c^{-1}P^T)(I - AB_{pre}))\vec{r}_k.
\end{aligned}$$

Thus we conclude that

$$B = B_{pre} + B_{post} - B_{post}AB_{pre} + (I - B_{post}A)(PA_c^{-1}P^T)(I - AB_{pre})$$

0.9.4 Proof of Theorem 1

The proof for theorem 1 comes from [12] and is as follows. Due the fact that L is SSPD, we restrict the analysis to the quotient space $\mathbb{V} := \mathbb{R}^n \setminus \text{span}\{\vec{1}\}$, on which L is SPD. According to the XZ-identity [**?**], we have:

$$\|I - BL\|_L^2 = 1 - \frac{1}{\mathcal{K}},\ \mathcal{K} = \sup_{\vec{x}\in\mathbb{V}} \inf_{\vec{x}_c\in\mathbb{V}_c} \frac{\|\vec{x} - P\vec{x}_c\|_{\bar{R}^{-1}}^2}{\|\vec{x}\|_L^2}.$$

Here, $\mathbb{V}_c := \text{Range}(P) \setminus \text{span}\{\vec{1}\}$, B_{TL} denotes the two-level UA-AMG preconditioner, $\bar{R} := 2D^{-1} - D^{-1}LD^{-1}$ is the symmetrized Jacobi smoother where $D = \text{diag}(L)$. It is easy to show that

$$\|\vec{x} - P\vec{x}_c\|_{\bar{R}^{-1}}^2 \leq C\|\vec{x} - P\vec{x}_c\|_D^2, \tag{7}$$

Denote $n_i = |V^i|$, we have

$$\begin{aligned}
\|\vec{x} - P\vec{x}_c\|_D^2 &= \sum_{i=1}^{n_\alpha} \sum_{v_s\in V^i} L_{ss}\left((\vec{x})_s - \frac{\sum_{v_t\in V^i}(\vec{x})_t}{n_i}\right)^2 \\
&= \sum_{i=1}^{n_a} \sum_{v_s\in V^i} L_{ss}\left(\frac{\sum_{v_t\in V^i}(\vec{x})_s - (\vec{x})_t}{n_i}\right)^2 \\
&= \sum_{i=1}^{n_a} \sum_{v_s\in V^i} \frac{L_{ss}}{n_i^2}\left(\sum_{v_t\in V^i}(\vec{x})_s - (\vec{x})_t\right)^2 \\
&\leq \sum_{i=1}^{n_a} \sum_{v_s\in V^i} \frac{L_{ss}}{n_i} \sum_{v_t\in V^i}\left((\vec{x})_s - (\vec{x})_t\right)^2.
\end{aligned}$$

If we zoom in on $((\vec{x})_s - (\vec{x})_t)^2$, it concerns the distance between a single pair of nodes. By the assumption that each aggregation is simply connected, there exists a path, P_{st} from vertex s to t in G^i. Therefore, we have

$$\begin{aligned}
((\vec{x})_s - (\vec{x})_t)^2 &= (\sum_{(p,q)\in P_{st}} ((\vec{x})_p - (\vec{x})_q))^2 \\
&\leqslant \{\text{diam}(G^i)\} \sum_{(p,q)\in P_{st}} ((\vec{x})_p - (\vec{x})_q)^2,
\end{aligned}$$

which leads to

$$
\begin{aligned}
&\|\vec{x} - P\vec{x}_c\|_D^2 \\
&\leqslant \sum_{i=1}^{n_a} \sum_{v_s \in V^i} \frac{L_{ss}}{n_i} \sum_{v_t \in V^i} \{\mathrm{diam}(G^i)\} \sum_{(p,q) \in P_{st}} ((\vec{x})_p - (\vec{x})_q)^2 \\
&\leqslant \sum_{i=1}^{n_a} \sum_{v_s \in V^i} \frac{L_{ss}}{n_i} \sum_{v_t \in V^i} \{\mathrm{diam}(G^i)\} \sum_{(p,q) \in E^i} ((\vec{x})_p - (\vec{x})_q)^2 \\
&= \sum_{i=1}^{n_a} \{\mathrm{diam}(G^i)\} \sum_{v_s \in V^i} L_{ss} \sum_{(p,q) \in E^i} ((\vec{x})_p - (\vec{x})_q)^2 \\
&= \sum_{i=1}^{n_a} \{\mathrm{diam}(G^i)\}\{\mathrm{vol}(G^i)\} \sum_{(p,q) \in E^i} \frac{\omega_{pq}}{\omega_{pq}} ((\vec{x})_p - (\vec{x})_q)^2 \\
&\leqslant \sum_{i=1}^{n_a} \{\mathrm{diam}(G^i)\}\{\mathrm{vol}(G^i)\} (\min_{e \in E^i} \omega(e))^{-1} \|\vec{x}\|_L^2.
\end{aligned}
$$

Together with (7), we can easily get the upper bound of $\mathcal{K}$ which completes the proof.

Bibliography

[1] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," http://grouplens.org/datasets/movielens/20m/, 2015, aCM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872.

[2] Y. Webscope, "Yahoo! instant messenger friends connectivity graph (g6)," https://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[3] "Disease module identification dream challenge," https://www.synapse.org/#!Synapse:syn6156761/wiki/, accessed: 2016-11-25.

[4] P. I. Series. Voting systems and the condorcet paradox. PBS. [Online]. Available: https://www.youtube.com/watch?v=HoAnYQZrNrQ&t=314s

[5] K. J. Arrow, "A difficulty in the concept of social welfare," *Journal of political economy*, vol. 58, no. 4, pp. 328–346, 1950.

[6] X. Jiang, L.-H. Lim, Y. Yao, and Y. Ye, "Statistical ranking and combinatorial hodge theory," *Mathematical Programming*, vol. 127, no. 1, pp. 203–244, 2011.

[7] A. N. Hirani, K. Kalyanaraman, and S. Watts, "Least squares ranking on graphs," *arXiv preprint arXiv:1011.1716*, 2010.

[8] C. Börgers, *Mathematics of social choice: voting, compensation, and division*. SIAM, 2010.

[9] L.-H. Lim, "Hodge laplacians on graphs," *arXiv preprint arXiv:1507.05379*, 2015.

[10] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[11] W. N. Bell, L. N. Olson, and J. B. Schroder, "PyAMG: Algebraic multigrid solvers in Python v3.0," 2015, release 3.0. [Online]. Available: http://www.pyamg.org

[12] X. Hu and J. Lin, "A Note on Aggregation-based AMG for Solving Graph Laplacians," *in preparation*, 2017.

[13] P. ERDdS and A. R&WI, "On random graphs i," *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.

[14] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[15] J. C. Urschel, X. Hu, J. Xu, and L. Zikatanov, "A simple cascade algorithm for computing the fiedler vector of graph laplacians," *Journal of Computational Mathematics*, vol. 33, no. 2, pp. 209–226, 2015.

[16] C. Colley, J. Lin, X. Hu, and S. Aeron, "Algebraic multigrid for least squares problems on graphs with applications to hodgerank," *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 627–636, 2017.