SIAM J. SCI. COMPUT.
Vol. 41, No. 5, pp. S220–S241

AN ADAPTIVE MULTIGRID METHOD BASED ON PATH COVER*

XIAOZHE HU†, JUNYUAN LIN†, AND LUDMIL T. ZIKATANOV‡

Abstract. We propose a path cover adaptive algebraic multigrid (PC-αAMG) method for solving linear systems with weighted graph Laplacians that can also be applied to discretized second order elliptic partial differential equations. The PC-αAMG is based on unsmoothed aggregation AMG (UA-AMG). To preserve the structure of smooth error down to the coarse levels, we approximate the level sets of the smooth error by first forming a vertex-disjoint path cover with paths following the level sets. The aggregations are then formed by matching along the paths in the path cover. In such a manner, we are able to build a multilevel structure at a low computational cost. The proposed PC-αAMG provides a mechanism to efficiently rebuild the multilevel hierarchy during the iterations and leads to a fast nonlinear multilevel algorithm. Traditionally, UA-AMG requires more sophisticated cycling techniques, such as AMLI-cycle or K-cycle, but as our numerical results show, the PC-αAMG proposed here leads to a nearly optimal standard V-cycle algorithm for solving linear systems with weighted graph Laplacians. Numerical experiments for some real-world graph problems also demonstrate PC-αAMG's effectiveness and robustness, especially for ill-conditioned graphs.

Key words. unsmoothed aggregation algebraic multigrid method, adaptive method, graph Laplacian, path cover

AMS subject classifications. 65N55, 65F10

DOI. 10.1137/18M1194493

1. Introduction.

Weighted graphs are frequently employed as data representations to describe a rich spectrum of application fields, including social, sensor, energy, and neuronal networks [3, 51, 21, 11]. The associated graph Laplacians naturally arise in large-scale computations of various application domains. For example, solving systems with weighted graph Laplacians is the core component for solving ranking and user recommendation problems [20, 24, 14]. In [12, 13, 31], similarities of proteins are calculated by solving graph Laplacian systems associated with the protein interaction networks. Furthermore, these similarities are used in clustering and labeling proteins. In addition, the marriage of graph Laplacian and popular computer-science topics such as convolutional neural networks and tensor decomposition has also been a dominating trend [18, 9, 26, 19, 10, 43, 30]. Advanced algorithms that adapt graph Laplacian properties to improve tasks include image reconstruction, clustering image datasets, and classification [1, 37, 23, 28]. To efficiently solve graph Laplacian systems, algebraic multigrid (AMG) is often applied. The standard AMG method was proposed to solve partial differential equations (PDEs) and mainly involves two parts: smoothing out the high-frequency errors on the fine levels, and eliminating the low-frequency errors on the coarse grids [49, 47, 42, 5, 40, 41]. AMG has been proven to be one of the most successful iterative methods in practical applications and many AMG methods have been developed for solving graph Laplacian systems, such as

*Received by the editors June 18, 2018; accepted for publication (in revised form) May 13, 2019; published electronically October 29, 2019.

https://doi.org/10.1137/18M1194493

Funding: The work of the first and second authors was partially supported by NSF grants DMS-1620063 and DMS-1812503. The work of the third author was supported by NSF grants DMS-1720114 and DMS-1819157.

†Department of Mathematics, Tufts University, Medford, MA 02155 (xiaozhe.hu@tufts.edu, junyuan.lin@tufts.edu).

‡Department of Mathematics, The Pennsylvania State University, University Park, PA 16802 (ltz@math.psu.edu).

combinatorial multigrid [27], lean AMG [32], and algebraic multilevel preconditioner based on matchings/aggregations [25, 38, 6, 15].

Traditional AMG methods usually build the multilevel structures beforehand and there is no interplay between the remaining errors and coarsening schemes. This results in many computational inefficiencies. For example, the cycling scheme applies the same multilevel hierarchy during the solve phase even when the convergence rate is slow, which is inefficient. To reduce the computational redundancy, adaptivity becomes essential. Substantial efforts have been made to incorporate adaptivity in iterative methods. Back in 1984, the original adaptive AMG algorithm [5] was proposed, laying the foundation for the development of self-learning and bootstrap AMG. The bootstrap AMG approach was further developed in [35, 4] and starts with several test vectors, while the adaptive approaches typically start with only one test vector. Directed by the theory of smoothed aggregation AMG (SA-AMG) developed in [46, 45], Brezina et al. introduced adaptive SA-AMG that determines interpolation operators based on information from the system itself rather than on explicit knowledge of the near-kernel space [7]. In the following year, the same authors further proposed an operator-induced interpolation approach that automatically represents smooth components [8]. In both these works, the basic idea is to iterate with the currently constructed AMG method (initially this is just a simple relaxation) toward the trivial solution $\boldsymbol{x} = \mathbf{0}$ of the homogeneous problem ($\boldsymbol{A}\boldsymbol{x} = \mathbf{0}$). These terations start with a random initial guess. In this way, the slow-to-converge errors are exposed after few iterations and the adaptive coarsening process is improved by accurately interpolating the algebraically smooth errors. MacLachlan, Manteuffel, and McCormick further developed a two-level, reduction-based nonlinear adaptive AMG and achieved local convergence and possibly global convergence in special cases [34]. The development of the unsmoothed aggregation AMG (UA-AMG) method is fairly recent. In [15], D'Ambra and Vassilevski applied a coarsening scheme that uses compatible weighted matching, which avoids reliance on the characteristics of connection strength in defining both the coarse space and the interpolation operators. There have also been other ideas on how to provide a good approximation to the smooth errors from the ranges of adaptively constructed interpolation operators. The majority of the existing works, however, use the assumption that the coarsening (aggregation or choice of coarse nodes) is already done and is independent of the right-hand side or the error distribution during iterations. This motivates us to focus on algorithms that use the smooth error for not only building interpolation operators but also determining the aggregates. We further follow such an idea and in this paper we propose an adaptive AMG method based on UA-AMG [2] where the aggregates are formed along the constancy sets of the slow-to-converge error.

We setup the multilevel structure by first finding a path cover [39] that approximates the level sets of the smooth error, then coarsening along the paths so that the structure of smooth error is well represented on the coarse levels, and building the smooth error into the range of interpolation operator to effectively eliminate it. The setup phase is reactivated when the convergence rate becomes slow, which makes the scheme adaptive to the slow-to-converge errors. Instead of using AMLI- or K-cycles for standard UA-AMG [6, 22, 47], we simply use V-cycle and achieve nearly uniform convergence for model problems. The design of combining UA-AMG and V-cycle has advantages in its simplicity and efficiency and therefore is favorable for solving a large linear system with ill-conditioned matrices. While the existing adaptive methods need to first solve the homogeneous problem, $\boldsymbol{A}\boldsymbol{x} = \mathbf{0}$, to determine the near-nullspace components and use this knowledge to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, our adaptive method solves $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ directly, for a general right-hand-side $\boldsymbol{b}$.

The rest of the paper is organized as follows. In section 2, we discuss the path cover finding algorithm that we use for coarsening. Our main adaptive AMG algorithm is discussed in section 3. Numerical results are shown in section 4. Finally, in section 5, we provide concluding remarks and enumerate possible future directions.

2. Preliminaries. In this section, we review basic aggregating methods and, more importantly, the path cover approximation algorithm [36]. We use these components in our main algorithm presented in section 3.

2.1. Basic algorithms for aggregation. There are various off-the-shelf aggregating methods that are frequently applied to graphs, such as the maximal independent set [33], the heavy edge coarsening algorithm [44], and maximal weighted matching (MWM) [17]. In this paper, we choose to compare the performance of our matching-like aggregating method (Algorithm 3.2) with the most closely related MWM. Similar comparison results could be observed between Algorithm 3.2 and other graph matching methods as well. Since we adopt an MWM aggregating scheme in our numerical experiments for comparison, we briefly recall the procedure of MWM.

The MWM algorithm forms matchings by visiting edges in the graph, from heaviest to lightest, and matches two endpoints of the edge if they are unaggregated. Commonly, there might be isolated nodes after applying MWM. We add those isolated nodes to existing matchings in order to keep the number of aggregates low, and at the same time we set three as an upper bound to the diameter of each aggregate.

2.2. Constructing a vertex-disjoint path cover. Consider a graph $G = (V, E, \omega)$ with positive weights but no self-loops or parallel edges, where V is the vertex set, E is the edge set, and $\omega > 0$ represents the weight set. A *path cover*, S, of G is a set of vertex-disjoint paths that covers all the vertices of G [39]. Now define

$$\omega^*(G) := \max_S \sum_{e \in S} \omega(e),$$

which is the maximum possible weight of a path cover S of G. In [36], Moran, Newman, and Wolfstahl pointed out that to find the exact maximal weighted path cover of a graph is an NP-complete problem, but on the other hand, they showed that one can find a $\frac{1}{2}$-approximated path cover, $\widetilde{S}$ of G, in $\mathcal{O}(|E| \cdot \log |E|)$ time, such that

$$\omega_{\widetilde{S}}(G) = \sum_{e \in \widetilde{S}} \omega(e) \geqslant \frac{1}{2}\omega^*(G).$$

The $\frac{1}{2}$-approximation path cover finding algorithm presented in Algorithm 2.1 is a slightly modified version of the algorithm from [36].

Algorithm 2.1 is a greedy algorithm that checks each edge from the heaviest to the lightest and incorporates edges as long as the set still contains only paths. The complexity of Algorithm 2.1 is $\mathcal{O}(|V| \log |V|)$ when the graph is sparse, $\mathcal{O}(|E|) = \mathcal{O}(|V|)$. Figure 1 illustrates a resulting path cover of a random graph, with weights displayed on its edges. Notice that there might be isolated points after finding the path cover. Later we include the isolated points using Algorithm 3.2, so that all the points on the original graph are represented on coarse levels.

3. Adaptive algorithm. The main idea and details of our path cover adaptive AMG (PC-αAMG) method are presented in this section. In subsection 3.1, the main idea of the algorithm is presented and demonstrated with an intuitive example. We present in subsections 3.2 and 3.3 the essential subroutines of approximating

Algorithm 2.1. Path cover.

procedure [`cover`] = `PathCover`($\boldsymbol{A}$)
  Input: $\boldsymbol{A}$—graph Laplacian of an undirected positive weighted graph $G = (V, E, \omega)$
  Output: `cover`—a path cover of graph G

  `sorted_edges` ← **Sort** the edges in descending order based on weights suggested in $\boldsymbol{A}$
  for $e(u, v) \in$ `sorted_edges` **do**
    if neither u nor v is in any paths in `cover` **then**
      Add $\{u, v\}$ as a new path in `cover`
    else if u is the endpoint of a path in `cover` and v is not in any paths **then**
      Append $\{v\}$ to `cover`{path that contains u}
    else if v is the endpoint of a path in `cover` and u is not in any paths **then**
      Append $\{u\}$ to `cover`{path that contains v}
    else if v and u are the endpoints of different paths in `cover` **then**
      Merge two paths
    end if
  end for
end procedure

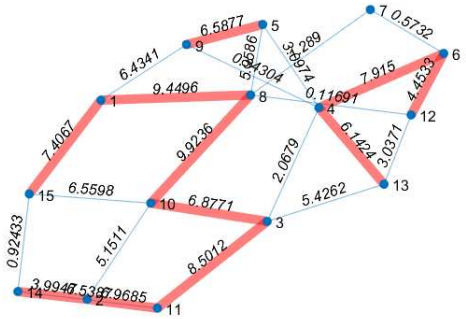

FIG. 1. *Path cover found by Algorithm* 2.1 *on a random graph with weights.*

smooth error using previously built multilevel hierarchies (Algorithm 3.1), building aggregations and prolongation operators from path cover (Algorithm 3.2), and setting up a multilevel structure using path cover and approximated error (Algorithm 3.3). Finally, the full PC-αAMG algorithm is presented in subsection 3.4.

3.1. Basic ideas and rationale of the adaptive algorithm. As mentioned, AMG reduces error during two procedures: smoothing out the high-frequency errors and eliminating low-frequency errors that are restricted to the coarse grids. Our adaptive algorithm contributes to the latter aspect by using adaptively designed multilevel hierarchies to preserve the current smooth error onto the coarse levels well enough, so that the current dominating smooth error can be efficiently eliminated on the coarsest level. Specifically, we utilize the path cover finding algorithm, Algorithm 2.1, to capture the level sets of the smooth error on the finest level, aggregate/match along these paths to preserve the smooth error on the coarser

levels, and reconstruct the AMG hierarchy to aim at this specific remaining smooth error. In this manner, we can eliminate the dominating smooth errors which cause slow convergence one by one, until the desired accuracy is met. In our opinion, it could be beneficial to have multilevel hierarchies which approximate the errors well when $\boldsymbol{b} \neq \boldsymbol{0}$. One possible approach, albeit beyond the scope of our considerations here, is to use adaptive aggregations based on a posteriori error estimates on graphs as proposed in [50]. Unlike other adaptive AMG methods, the PC-αAMG algorithm proposed here integrates the setup and solve phase together by identifying the smooth errors while solving the linear system $\boldsymbol{Ax} = \boldsymbol{b}$.

To test if the aggregations/matchings along the level sets of smooth error can successfully preserve the smooth error onto coarse levels, we take a manufactured smooth error (Figure 2, upper left), manually build matchings (Figure 2, lower left) on its level sets (Figure 2, upper right), and use the matchings to build the prolongation operator to restrict and prolongate the error back to the original level. We include a plot of the difference between the two smooth errors (Figure 2, lower right). The l^2-norm of the difference is below 10^{-15}. This reassures us that the aggregating scheme based on level set is efficient in capturing the smooth error. In our algorithm, we use path cover to approximate level sets, where each path in the path cover represents one level set. In this way, finding level sets can be done purely algebraically using only the matrices. The smooth error, which is symbolized by the manufactured error (Figure 2, upper left), can be approximated by subsection 3.2 when solving slows down. We present subroutines in subsection 3.3 to automatically aggregate along each path in the path cover that approximates the level sets of the approximated smooth error.

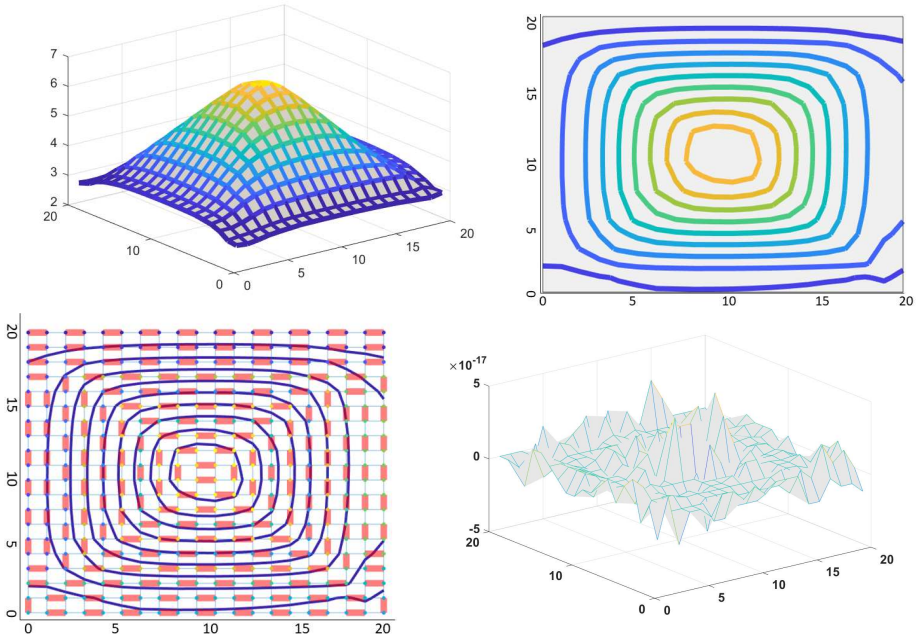

FIG. 2. *Upper left: smooth error. Upper right: level sets of smooth error. Lower left: matchings on level sets. Lower right: difference between the original smooth error and the error after restricting and prolongating.*

It is verifiable that we can achieve a similar effect that the manufactured aggregates (Figure 2, lower left) have in preserving the smooth error structure.

3.2. Approximating the smooth error. Since all the subroutines that aim to preserve and ultimately eliminate the smooth error rely on the fact that the smooth error on the fine level can be well approximated, one essential step of our algorithm is to approximate the smooth error accurately when the convergence rate becomes slow. The fact is, at the kth iteration, the approximated solution $\boldsymbol{x}^k$ is available at hand; however, the error $\boldsymbol{e}^k$ cannot be computed straightforwardly. Based on the well-known error equation $\boldsymbol{A}\boldsymbol{e}^k = \boldsymbol{r}^k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k$, we try to approximately solve the error equation by applying several steps of the multigrid preconditioned conjugate gradient (PCG) method (see Algorithm 3.1 for details). For the first several re-setups, we use several W-cycles to get a good approximation (this corresponds to the case "$\texttt{Re} \leqslant \texttt{numW}$" in Algorithm 3.1). In practice, we minimize the usage of W-cycle (see section 4 for details). For later re-setups, we simply use V-cycles based on the existing multilevel structures. More precisely, in step 10 of Algorithm 3.1, we use a symmetric composite preconditioner proposed in [15]:

$$\boldsymbol{e} \leftarrow \prod_{j=0}^{2\cdot\texttt{Re}+1} (\boldsymbol{I} - \boldsymbol{B}_j \boldsymbol{A})\boldsymbol{e},$$

where Re is the number of re-setups, and $\boldsymbol{B}_{\texttt{Re}+j} = \boldsymbol{B}_{\texttt{Re}+1-j}, j = 1, \ldots \texttt{Re}+1$. Each $\boldsymbol{B}_j$ corresponds to the preconditioning effect of multilevel hierarchy built from the jth re-setup, i.e., $\texttt{hist}\{j\}$. In this manner, we recycle all the hierarchies ever built and reduce the computational waste.

Algorithm 3.1. Approximate smooth error.

1: **procedure** [$\boldsymbol{e}$] = ApproximateSmoothError($\boldsymbol{A}, \{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \boldsymbol{r}$, hist, Re, $\boldsymbol{e}$)
2: Inputs:
a: $\{\boldsymbol{A}_\ell\}_{\ell=2}^{L}$—a set of graph Laplacians of the 2nd to Lth level graphs
b: $\boldsymbol{r}$—residual vector
c: hist—all the hierarchies ever created
d: Re—re-setup count

3: Choose parameters:
a: numW ▷ Number of smooth errors (re-setups) using W-cycle
b: iterW ▷ Number of iterations of W-cycle PCG
c: iterV ▷ Number of iterations of V-cycle PCG
4: **if** $\texttt{Re} \leqslant \texttt{numW}$ **then**
5: **for** $i \leftarrow 1 : \texttt{iterW}$ **do**
6: $\boldsymbol{e} \leftarrow \texttt{W\_cycle\_PCG}(\boldsymbol{A}, \boldsymbol{r}, \boldsymbol{e}, \texttt{hist}\{\texttt{Re}\})$
7: **end for**
8: **else**
9: **for** $i \leftarrow 1 : \texttt{iterV}$ **do**
10: $\boldsymbol{e} \leftarrow \texttt{V\_cycle\_PCG}(\boldsymbol{A}, \boldsymbol{r}, \boldsymbol{e}, \texttt{hist}\{\texttt{1:Re}\})$
11: **end for**
12: **end if**
13: Make $\boldsymbol{e}$ orthogonal to $\mathbf{1}$ ▷ Since $\mathbf{1}$ is in the nullspace of $\boldsymbol{A}$
14: **end procedure**

3.3. Coarsening based on path cover. We use the information of the (approximate) smooth error in forming aggregations. In Figure 3, we illustrate the process of approximating the level sets of a smooth error (Figure 3, left) using the path cover finding algorithm (Algorithm 2.1) and aggregating along the path cover (Algorithm 3.2). We reassign the weights in Algorithm 3.3, step 2, i.e., $\widetilde{\boldsymbol{A}}_{ij} = 1/\|\boldsymbol{e}_i - \boldsymbol{e}_j\|$, so that edges between nodes of similar values (nodes on the same level set) have heavy weights and are more likely to be chosen to form the path cover. However, the edges

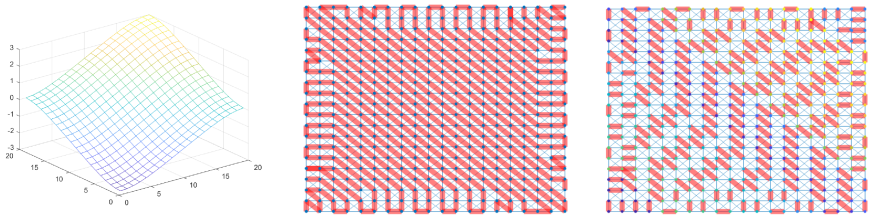

FIG. 3. *Left: smooth error. Middle: path cover. Right: matching/aggregates on path cover.*

Algorithm 3.2. Path cover aggregation.

procedure [$\boldsymbol{P}$] = PATHCOVERAGGREGATE(COVER, $\widetilde{\boldsymbol{A}}, \ell, \boldsymbol{e}$)
  Input:
  a: $\widetilde{\boldsymbol{A}}$—graph Laplacian (possibly reweighted)
  b: ℓ—level count
  c: $\boldsymbol{e}$—smooth error
  Output: $\boldsymbol{P}$—prolongation operator

  $n \leftarrow$ number of nodes in graph induced by $\widetilde{\boldsymbol{A}}$
  for each path in `cover` **do**
    Match each vertex with one of its neighbors in the path
  end for
  `numAgg` $\leftarrow$ number of matchings
  for each unaggregated vertex u **do**
    $v \leftarrow u$'s neighbor with the maximal edge weight based on $\widetilde{\boldsymbol{A}}$
    if v is also unaggregated **then** ▷ Case 1
      Match u and v together
    else if `size`(aggregation of v) $= 2$ **then** ▷ Case 2
      Aggregate u into v's aggregation
    else ▷ Case 3, `size`(aggregation of v) $= 3$
      Match u and v and remove v from previous aggregation of v
      `numAgg` $\leftarrow$ `numAgg` $+ 1$
    end if
  end for
  if $\ell = 1$ **then**
    Build prolongation operator $\boldsymbol{P}_\ell \in \mathbb{R}^{n \times \texttt{numAgg}}$ by (3.1)
  else
    Build prolongation operator $\boldsymbol{P}_\ell \in \mathbb{R}^{n \times \texttt{numAgg}}$ by (3.2)
  end if
end procedure

Algorithm 3.3. Path cover AMG setup.

1: **procedure** $[\ \{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L}\ ] = \textsc{PathCoverAMG\_setup}(\boldsymbol{A}, \boldsymbol{e})$
2: $\quad \widetilde{\boldsymbol{A}}_{ij} \leftarrow 1/\|\boldsymbol{e}_i - \boldsymbol{e}_j\|$, if edge $e(i,j)$ is in the graph induced by $\boldsymbol{A}^2$
3: $\quad$ `cover` $\leftarrow$ `PathCover`$(\widetilde{\boldsymbol{A}})$
4: $\quad \boldsymbol{A}_1 \leftarrow \boldsymbol{A}$
5: $\quad$ **for** $\ell \leftarrow 1 : L$ **do**
6: $\quad\quad \boldsymbol{P}_\ell \leftarrow$ `PathCoverAggregate`$(\texttt{cover}, \widetilde{\boldsymbol{A}}, \ell, \boldsymbol{e})$
7: $\quad\quad \widetilde{\boldsymbol{A}} \leftarrow \boldsymbol{P}_\ell^T \widetilde{\boldsymbol{A}} \boldsymbol{P}_\ell$
8: $\quad\quad$ Shorten each path in `cover` by merging the vertices in one aggregate into a vertex
9: $\quad\quad \boldsymbol{A}_{\ell+1} \leftarrow \boldsymbol{P}_\ell^T \boldsymbol{A}_\ell \boldsymbol{P}_\ell$
10: $\quad$ **end for**
11: **end procedure**

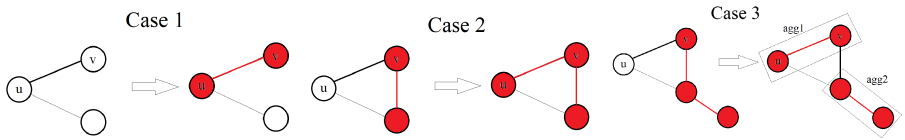

FIG. 4. *Aggregating the isolated points.*

in the original graph might not align with the heavy edges (e.g., the diagonal lines in the leftmost subplot in Figure 3). We therefore use the sparsity pattern of $\boldsymbol{A}^2$ to include more edges while keeping the sparsity of the graph relatively low (for the example shown in Figure 3, the sparsity pattern of $\boldsymbol{A}^2$ includes the diagonal edges which are exactly on the level sets of the smooth error). After reassigning weights and changing the sparsity pattern, Algorithm 2.1 finds the path cover. As shown in the middle subplot of Figure 3, as an example, the path cover approximates level sets of the smooth error as expected. Note that since smooth error is not perfectly linear near the boundaries, the level sets are not strictly diagonal.

As previously demonstrated, there might be isolated points after creating a path cover, or possibly other sources. Algorithm 3.2 handles the isolated points altogether. Figure 4 corresponds to steps 9–19 in Algorithm 3.2 and shows all three possible cases. As we can see in the right subplot of Figure 3, after rearrangement, there are no isolated vertices and the resulting aggregations are all of size 2 or 3.

Figure 3 only explicitly shows the process of coarsening on the finest level. If the smooth error is well preserved onto the coarse level, its level sets should not change much on the coarse levels. Thus, we simply match the nodes on each path in the fine-level path cover to generate the coarse-level path cover and recurse this process for all coarse levels.

We also use the smooth error in building prolongation operator $\boldsymbol{P}_\ell$ for the coarsest level ($\ell = 1$)

$$(\boldsymbol{P}_\ell)_{ki} = \begin{cases} \boldsymbol{e}_k & \text{if } k \in V_\ell^i, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

For all other levels ($\ell > 1$), $\boldsymbol{P}_\ell$ is defined as the standard UA-AMG prolongation operator

$$(\boldsymbol{P}_\ell)_{ki} = \begin{cases} 1 & \text{if } k \in V_\ell^i, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Here V_ℓ^i is the ith aggregate on ℓ's level graph, where $V_\ell = \bigcup_{i=1} V_\ell^i$ and $V_\ell^i \cap V_\ell^j = \emptyset, i \neq j$. $\boldsymbol{e}_k$ is the kth entry of the (approximate) smooth error $\boldsymbol{e}$. Obviously, we have $\boldsymbol{e} \in \text{Range}(\boldsymbol{P}_1)$.

3.4. Main algorithm. The overall PC-αAMG algorithms are presented in Algorithms 3.4 and 3.5. Algorithm 3.4 is for the general case $\boldsymbol{Ax} = \boldsymbol{b}$. In Algorithm 3.4, we save a small number of hierarchical structures in order to approximate the smooth error more accurately in step 18, which is the key step for the adaptive scheme to work well. While such an approach increases the storage, it pays off for graph Laplacian systems that are hard to solve (very ill-conditioned) because the convergence rate is under control. As the numerical experiments (section 4) clearly show, storing the additional coarse-level graph Laplacians adds no more than $|E|$ real numbers to the total storage. Furthermore, since we use a matching-like aggregation method, storing the prolongation operators requires storing at most $2|V|$ real numbers. As a result, storing all the hierarchical structures created from `Re` times re-setups requires $(|E|+2|V|)\cdot$`Re` real numbers. In all of the test cases, we observed that `Re` averages to less than 10. Thus, the increase in storage is asymptotically negligable and the total required memory is of order $\mathcal{O}(|V|+|E|)$, which is optimal. Notice that Algorithm 3.5 is a special algorithm for solving $\boldsymbol{Ax} = \boldsymbol{0}$. One of the main reasons to include such a case is that the solution is trivial to find, and hence, the (smooth) error can be directly computed with no additional storage cost. Another reason is that following the basic idea of adaptive AMG methods, Algorithm 3.5 can also be used as a standalone setup phase and build several multilevel structures that are effective for eliminating smooth errors. The two versions of the algorithm include all of the aforementioned components, yet the main procedure of both versions is rather straightforward. We use MWM to build the initial multilevel structure and use it as the preconditioner to solve the model problem. For well-conditioned graph Laplacian problems, MWM might already give good performance, so there is no need to use the adaptive procedure. Our PC-αAMG aims at difficult problems which have error components that are slow to converge for MWM. When the current multilevel hierarchy is not desirable anymore, we re-setup using aggregations based on path cover of the current smooth error. Re-setup is invoked when the convergence slows down, in order to create hierarchical structures that efficiently eliminate slow-to-converge smooth errors. We use Figure 5 to demonstrate our PC-αAMG for the homogeneous case (Algorithm 3.5). The procedure for general $\boldsymbol{b}$ (Algorithm 3.4) is very similar, except that the smooth error $\boldsymbol{e}$ can only be approximated rather than directly computed.

4. Numerical results. In this section, we conduct numerical experiments on different types of graph Laplacians. Some of them are related to discrete PDEs and the rest of them are from real-world networks.

We compare the performances of V-cycle UA-AMG using MWM as a coarsening scheme and Gauss–Seidel as a smoother with proposed Algorithms 3.4 and 3.5. For Algorithm 3.4, we choose two values for `threshold`, which results in two different re-setup strategies. In the first case (denoted as Algorithm 3.4(1)), we choose `threshold` $= 10^{-6}$, which essentially rebuilds the multilevel hierarchy after each iteration. Since the newly built hierarchy is specifically for eliminating the current smooth error, we expect that re-setup at each step would give the best performance in terms of iteration counts. However, re-setup at every iteration is computationally

Algorithm 3.4. PC-αAMG (for general $\boldsymbol{b}$).

procedure $[\ \boldsymbol{x}\ ] = \textsc{AdaptiveAMG}(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}, \texttt{TOL}, \texttt{MAX\_ITER}, \texttt{THRESHOLD})$
  $\{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L} \leftarrow \texttt{MWM\_setup}(\boldsymbol{A})$
  $k \leftarrow 1$ ▷ Initialization for iteration number
  $\texttt{Re} \leftarrow 1$ ▷ Initialization for re-setup counts
  $\texttt{hist}\{\texttt{Re}\} \leftarrow \{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L}$ ▷ `hist` saves all the hierarchical structures
  $\boldsymbol{e} \leftarrow \boldsymbol{0}$ ▷ Initialize approximated smooth error at $\boldsymbol{0}$
  $\boldsymbol{x}^k \leftarrow \boldsymbol{x}$
  $\boldsymbol{r}^k \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^k$
  while $k <$ `max_iter` and $\|\boldsymbol{r}^k\|/\|\boldsymbol{r}^1\| >$ `tol` **do**
    $\boldsymbol{x}^{k+1} \leftarrow \texttt{V\_cycle}(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}^k, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L}, \{\boldsymbol{A}_\ell\}_{\ell=2}^{L})$
    Make $\boldsymbol{x}^k$ orthogonal to $\boldsymbol{1}$ ▷ Since $\boldsymbol{1}$ is in the nullspace of $\boldsymbol{A}$
    $\boldsymbol{r}^{k+1} \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{k+1}$
    $\texttt{ConvF} \leftarrow \frac{\|\boldsymbol{r}^{k+1}\|}{\|\boldsymbol{r}^k\|}$ ▷ Compute convergence factor at $(k+1)$ step
    if average `ConvF` of iterations after the last re-setup > `threshold` **then**
      $\boldsymbol{e} \leftarrow \texttt{ApproximateSmoothError}(\boldsymbol{A}, \boldsymbol{r}^{k+1}, \texttt{hist}, \texttt{Re}, \boldsymbol{e})$
      $\{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L} \leftarrow \texttt{PathCoverAMG\_setup}(\boldsymbol{A}, \boldsymbol{e})$
      $\texttt{Re} \leftarrow \texttt{Re} + 1$
      $\texttt{hist}\{\texttt{Re}\} \leftarrow \{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L}$
      $\boldsymbol{x}^{k+1} \leftarrow \boldsymbol{x}^{k+1} + \boldsymbol{e}$
    end if
    $k \leftarrow k + 1$
  end while
end procedure

Algorithm 3.5. PC-αAMG (for $\boldsymbol{b} = \boldsymbol{0}$).

procedure $[\ \boldsymbol{x}\ ] = \textsc{AdaptiveAMG}(\boldsymbol{A}, \boldsymbol{0}, \boldsymbol{x}, \texttt{TOL}, \texttt{MAX\_ITER}, \texttt{THRESHOLD})$
  $\{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L} \leftarrow \texttt{MWM\_setup}(\boldsymbol{A})$
  $k \leftarrow 1$ ▷ Initialization for iteration number
  $\boldsymbol{x}^k \leftarrow \boldsymbol{x}$
  $\boldsymbol{r}^k \leftarrow \boldsymbol{0} - \boldsymbol{A}\boldsymbol{x}^k$
  while $k <$ `max_iter` and $\|\boldsymbol{r}^k\|/\|\boldsymbol{r}^1\| >$ `tol` **do**
    $\boldsymbol{x}^{k+1} \leftarrow \texttt{V\_cycle}(\boldsymbol{A}, \boldsymbol{0}, \boldsymbol{x}^k, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L}, \{\boldsymbol{A}_\ell\}_{\ell=2}^{L})$
    Make $\boldsymbol{x}^k$ orthogonal to $\boldsymbol{1}$ ▷ Since $\boldsymbol{1}$ is in the nullspace of $\boldsymbol{A}$
    $\boldsymbol{r}^{k+1} \leftarrow \boldsymbol{0} - \boldsymbol{A}\boldsymbol{x}^{k+1}$
    $\texttt{ConvF} \leftarrow \frac{\|\boldsymbol{r}^{k+1}\|}{\|\boldsymbol{r}^k\|}$ ▷ Compute convergence factor at $(k+1)$ step
    if average `ConvF` of iterations after the last re-setup > `threshold` **then**
      $\boldsymbol{e} \leftarrow \frac{\boldsymbol{x}^{k+1} - \boldsymbol{0}}{\|\boldsymbol{x}^{k+1} - \boldsymbol{0}\|}$
      $\{\boldsymbol{A}_\ell\}_{\ell=2}^{L}, \{\boldsymbol{P}_\ell\}_{\ell=1}^{L} \leftarrow \texttt{PathCoverAMG\_setup}(\boldsymbol{A}, \boldsymbol{e})$
    end if
    $k \leftarrow k + 1$
  end while
end procedure

expensive. Therefore, we consider another case where we set `threshold` = 0.4 (denoted as Algorithm 3.4(2)) so that re-setup will not be triggered every iteration. This choice balances the re-setup times and error reduction efficiency, which potentially

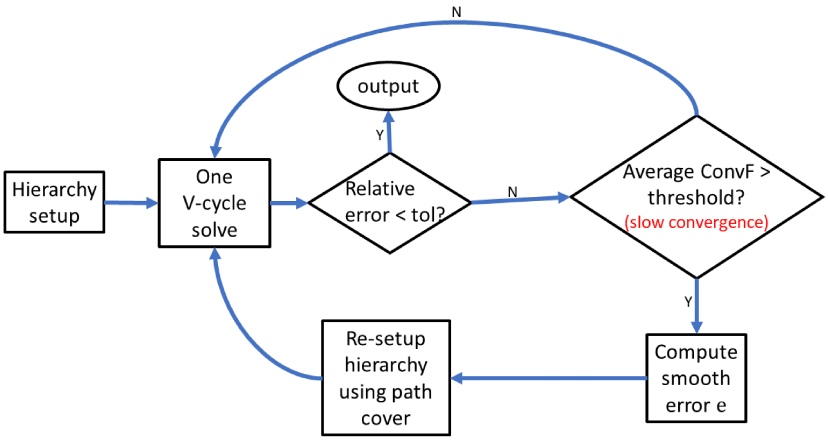

FIG. 5. *Procedure of PC-αAMG (Algorithm 3.5).*

reduces the total CPU time. For Algorithm 3.5, we simply use `threshold` $= 0.5$, which seems to give the best performance in our numerical experiments.

For other parameters used in Algorithm 3.4, we set `numW` $= 2$, `iterW` $= 7$ and `iterV` $= 4$ to approximate the smooth errors. We use W-cycles for the first two re-setups in order to get a good approximated error at the early stage and, later on, just use a few V-cycles to approximate the error. The relaxation scheme is Gauss–Seidel and the solving step adopts a basic multigrid V-cycle.

We chose to test our model problems with three different right-hand sides: a low-frequency $\boldsymbol{b} \in \mathbb{R}^{|V|} = [1, 1, \ldots, 1 - |V|]^T$, a randomly generated $\boldsymbol{b}$ that sums to zero, and $\boldsymbol{b} = \mathbf{0}$. We used random initial guess $\boldsymbol{x}$ when testing with the homogeneous case and zero initial guess $\boldsymbol{x}$ when testing with nonzero $\boldsymbol{b}$'s. For all the experiments below, the solver stops when the residual reaches the predescribed tolerances. We use a relatively small tolerance 10^{-8} for the scalability tests in regular grids (subsection 4.1) and 10^{-6} for ring graphs (subsection 4.2) as well as real-world graphs (subsection 4.3). The symbol "–" in any column of the tables means the number of iterations exceeds 2500.

Numerical experiments were conducted using a 3 GHz Intel Xeon Sandy Bridge CPU with 256 GB of RAM. We tested with a single core without parallelization. The software used is an AMG package written by the authors and implemented in MATLAB. We report the following results in each table. The "ConvF" in the MWM column is the algebraic average of convergence factor for the last 10 iterations, where the convergence factor is calculated as in step 13 of Algorithm 3.4. The higher the convergence factor, the slower the convergence rate. We omit this factor for the adaptive algorithms columns because we restart when the average convergence factor is above a threshold. The "Iter" column reports the number of iterations needed for the residual to reach certain tolerances, and "Re" records the number of re-setups needed specifically for Algorithm 3.4(2) and Algorithm 3.5 (since Algorithm 3.4(1) re-setups every time, Re $=$ Iter $- 1$). "OC" is the operator complexity defined in [42], i.e., the ratio of the total number of nonzeros of matrices on all levels divided by the number of nonzeros of the finest-level matrix. The operator complexity is calculated as the algebraic average of the operator complexities of all generated hierarchies. Finally, we report "t" as the total CPU times in seconds.

4.1. Tests on graphs corresponding to regular grids. We first tested the performance of the algorithms on graph Laplacians of unweighted two-dimensional (2D) regular uniform grids in Tables 1–3. Those regular grids correspond to solving a Poisson equation with Neumann boundary condition on a 2D square using the finite-difference or finite-element method (see Figure 6 for the case $|V| = 16$). In this case, $\boldsymbol{A}$ is the graph Laplacians of the unweighted grids.

Notice that for regular grids, while the number of iterations for regular V-cycle UA-AMG in all cases grows rapidly and quickly exceeds 2500, the number of iterations for Algorithm 3.4 in Tables 1 and 2 is nearly uniform and in Table 3 is uniform. Total CPU time is plotted in Figure 7 and we can see that the total CPU time of PC-αAMG increases nearly linearly with respect to the matrix size for both zero and nonzero $\boldsymbol{b}$ (the line with slope 1 is added for reference). The growth rate of the CPU time is

TABLE 1
Performance of solving regular grids with low-frequency $\boldsymbol{b}$, *tol*=10^{-8}.

	UA-AMG w/MWM				Algorithm 3.4(1)			Algorithm 3.4(2)			
n	Iter	ConvF	t	OC	Iter	t	OC	Iter	Re	t	OC
64^2	278	0.948	0.34	2.07	7	0.78	2.10	13	6	0.73	2.10
100^2	431	0.967	0.73	2.08	8	1.37	2.10	15	7	1.63	2.10
128^2	878	0.979	2.07	2.08	9	2.93	2.12	15	7	2.81	2.11
200^2	1000	0.986	5.85	2.09	9	5.35	2.12	18	7	5.85	2.12
256^2	1960	0.990	18.10	2.09	10	10.11	2.12	19	9	9.56	2.13
400^2	2065	0.994	48.29	2.09	11	28.12	2.13	19	9	26.27	2.13
512^2	–	0.996	–	2.09	11	44.39	2.14	21	10	53.15	2.15

TABLE 2
Performance of solving regular grids with zero-sum random $\boldsymbol{b}$, *tol*=10^{-8}.

	UA-AMG w/MWM				Algorithm 3.4(1)			Algorithm 3.4(2)			
n	Iter	ConvF	t	OC	Iter	t	OC	Iter	Re	t	OC
64^2	269	0.949	0.25	2.07	7	0.65	2.10	14	6	0.80	2.10
100^2	392	0.967	0.69	2.08	8	1.39	2.11	16	6	1.36	2.13
128^2	625	0.979	1.65	2.08	8	2.19	2.15	16	7	2.19	2.14
200^2	960	0.986	6.43	2.09	9	5.73	2.15	17	8	6.33	2.15
256^2	1383	0.991	13.02	2.09	10	12.20	2.16	18	8	10.45	2.15
400^2	1459	0.994	35.96	2.10	11	31.73	2.16	20	10	34.08	2.15
512^2	–	0.996	–	2.10	11	59.83	2.17	20	10	54.08	2.16

TABLE 3
Performance of solving regular grids with $\boldsymbol{b} = \boldsymbol{0}$, *tol*=$10^{-8}$.

	UA-AMG w/MWM				Algorithm 3.5			
n	Iter	ConvF	t	OC	Iter	Re	t	OC
64^2	112	0.948	0.12	2.06	22	4	0.32	2.07
100^2	152	0.967	0.27	2.08	22	4	0.34	2.11
128^2	203	0.979	0.60	2.08	22	4	0.54	2.13
200^2	246	0.984	1.66	2.09	23	4	1.78	2.14
256^2	283	0.986	2.77	2.09	22	4	3.07	2.15
400^2	313	0.989	5.56	2.13	22	4	6.25	2.16
512^2	375	0.992	17.42	2.10	22	4	10.76	2.17

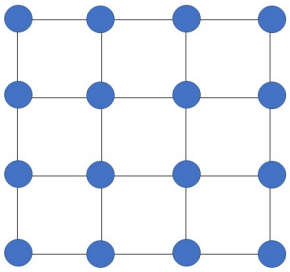

FIG. 6. *Example of a regular grid with* $|V| = 16$.

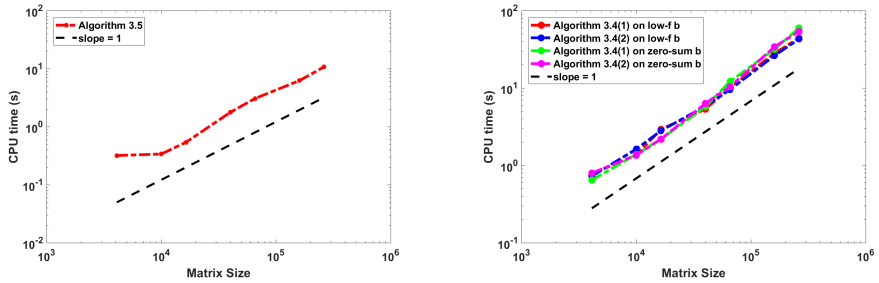

FIG. 7. *CPU time elapsed for solving regular grids with* $\boldsymbol{b} = \mathbf{0}$ *(left) and nonzero* $\boldsymbol{b}$ *(right).*

slightly slower than linear for small n, which is probably due to the fact that overhead cost is more pronounced than the actual computing cost when n is small. But for large n, we can see the nearly linear growth asymptotically, which demonstrates that the computational cost of our PC-αAMG is nearly optimal. In addition, although compared to Algorithm 3.4(1), Algorithm 3.4(2) takes more iterations to converge, it needs slightly less CPU time because it re-setups fewer times than Algorithm 3.4(1). This justifies our choices of different thresholds.

To investigate different choices of smoothers and the potential of applying them in the parallel version of PC-αAMG algorithm, we included additional numerical experiments based on using the Jacobi method as the smoother in Algorithms 3.4(1) and 3.5. We compared the results obtained by using Jacobi and sequential Gauss–Seidel smoothers in Tables 4 and 5. In both tables, we can see that the Gauss–Seidel smoother outperforms the Jacobi smoother as expected since it provides a more accurate approximate smooth error than the Jacobi smoother. However, when applying the Jacobi smoother, our adaptive AMG algorithm is still nearly optimal in terms of number of iterations and computational cost. Note that since our current implementation for the Jacobi method is still sequential, we should expect much faster CPU times if we use parallel implementation.

4.2. Tests for ring graphs. For the second example, we use the Watts–Strogatz [48] model and set the rewiring probability $\beta = 0$ (this removes the randomness in generating edges) and set the mean node degree to be 4 in order to produce unweighted ring graphs as in Figure 8. For this test problem, $\boldsymbol{A}$ is the graph Laplacians of these unweighted ring graphs. The condition numbers of the graph Laplacians of the ring graphs also grow rapidly when the size of the graphs increases.

TABLE 4
Performance of solving regular grids with low-frequency $\boldsymbol{b}$ *and different smoothers using Algorithm* 3.4(1), *tol*=10^{-8}.

	Gauss–Seidel			Jacobi		
n	Iter	t	OC	Iter	t	OC
64^2	7	0.78	2.10	9	0.71	2.10
100^2	8	1.37	2.10	10	1.44	2.13
128^2	9	2.93	2.12	10	2.28	2.15
200^2	9	5.35	2.12	11	6.52	2.15
256^2	10	10.11	2.12	12	11.67	2.16
400^2	11	28.12	2.13	13	34.62	2.16
512^2	11	44.39	2.14	15	74.05	2.17

TABLE 5
Performance of solving regular grids with $\boldsymbol{b} = \boldsymbol{0}$ *and different smoothers using Algorithm* 3.5, *tol*=10^{-8}.

	Gauss–Seidel				Jacobi			
n	Iter	Re	t	OC	Iter	Re	t	OC
64^2	22	4	0.32	2.07	32	22	1.19	2.16
100^2	22	4	0.34	2.11	32	22	2.48	2.18
128^2	22	4	0.54	2.13	33	22	4.28	2.20
200^2	22	4	1.78	2.14	32	22	10.73	2.21
256^2	22	4	3.07	2.15	31	22	17.03	2.21
400^2	22	4	6.25	2.16	31	22	48.65	2.22
512^2	22	4	10.76	2.17	31	22	84.54	2.22

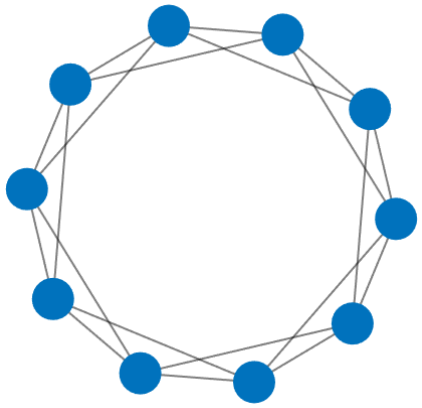

FIG. 8. *Example of a ring graph with* $|V| = 10$.

The results for nonzero $\boldsymbol{b}$ are presented in Tables 6 and 7. Like the results of the regular grids, the PC-αAMG method on ring graphs requires a small number of iterations and re-setups to converge, while the standard V-cycle UA-AMG eventually cannot converge within 2500 iterations for large graphs. In Table 8, we show the results for the homogeneous case. Compared to V-cycle UA-AMG, Algorithm 3.5 solves all the test cases with a steady number of iterations and re-setups. Therefore,

TABLE 6
Performance of solving ring graphs with low-frequency $\boldsymbol{b}$*,* tol=10^{-6}.

	UA-AMG w/MWM				Algorithm 3.4(1)			Algorithm 3.4(2)			
n	Iter	ConvF	t	OC	Iter	t	OC	Iter	Re	t	OC
32^2	108	0.894	0.07	1.59	6	0.24	1.57	12	5	0.20	1.57
50^2	217	0.947	0.11	1.58	7	0.26	1.59	11	5	0.25	1.59
64^2	421	0.974	0.29	1.59	7	0.42	1.59	15	6	0.34	1.59
100^2	832	0.987	1.33	1.60	9	1.12	1.60	17	8	1.08	1.60
128^2	1598	0.993	3.61	1.60	10	1.98	1.60	17	8	1.73	1.59
200^2	–	0.996	–	1.60	11	5.32	1.60	18	9	4.82	1.60
256^2	–	0.998	–	1.60	14	12.41	1.60	18	10	10.84	1.60

TABLE 7
Performance of solving ring graphs with zero-sum random $\boldsymbol{b}$*,* tol=10^{-6}.

	UA-AMG w/MWM				Algorithm 3.4(1)			Algorithm 3.4(2)			
n	Iter	ConvF	t	OC	Iter	t	OC	Iter	Re	t	OC
32^2	103	0.894	0.06	1.56	6	0.22	1.57	11	5	0.22	1.58
50^2	225	0.947	0.17	1.58	7	0.24	1.59	12	6	0.24	1.59
64^2	428	0.973	0.31	1.59	8	0.46	1.60	13	7	0.40	1.60
100^2	832	0.987	1.47	1.60	9	1.12	1.60	15	8	1.08	1.60
128^2	1561	0.993	3.91	1.60	10	1.96	1.60	17	9	1.98	1.60
200^2	–	0.997	–	1.60	12	6.12	1.60	18	10	6.13	1.60
256^2	–	0.998	–	1.60	13	11.25	1.60	18	11	11.50	1.60

TABLE 8
Performance of solving ring graphs with $\boldsymbol{b} = \boldsymbol{0}$*,* tol=10^{-6}.

	UA-AMG w/MWM				Algorithm 3.5			
n	Iter	ConvF	t	OC	Iter	Re	t	OC
32^2	33	0.859	0.03	1.56	15	2	0.04	1.58
50^2	47	0.923	0.03	1.58	17	2	0.05	1.59
64^2	56	0.948	0.04	1.59	17	2	0.05	1.60
100^2	65	0.961	0.10	1.60	18	2	0.11	1.61
128^2	76	0.964	0.16	1.60	17	2	0.19	1.61
200^2	85	0.966	0.35	1.60	17	2	0.42	1.61
256^2	96	0.968	0.58	1.60	17	2	0.71	1.61

when solving larger ring graphs with sizable condition numbers, the efficiency gain of Algorithm 3.5 would be more significant. Since the tested linear systems are relatively well-conditioned when the matrix sizes are small, they could be quickly solved even using V-cycle UA-AMG.

In Figure 9, we plotted the total CPU times in seconds for both zero and nonzero $\boldsymbol{b}$. The results are similar to those of the regular grids. When the matrix size is small, the CPU time grows slower than linear. Asymptotically, the CPU time grows nearly linearly with respect to n, which demonstrates that our PC-αAMG has nearly optimal convergence. Moreover, compared to Algorithm 3.4(1), Algorithm 3.4(2) still takes slightly shorter CPU times to converge for most cases.

4.3. Tests for real-world graphs. Besides the graphs generated above, we also tested real-world graphs from the Stanford Large Network Dataset Collection [29]

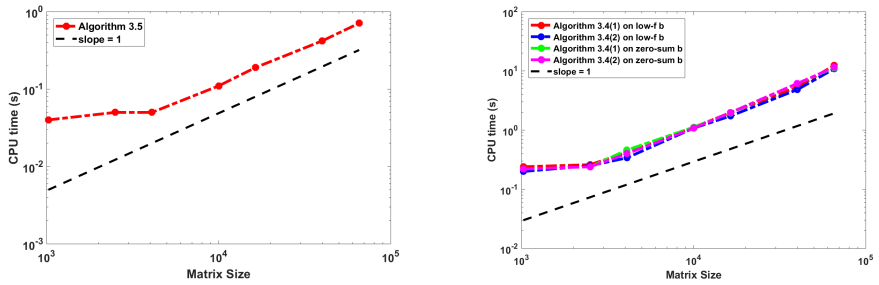

FIG. 9. *CPU time elapsed for solving ring graphs with* $\boldsymbol{b} = \mathbf{0}$ *(left) and nonzero* $\boldsymbol{b}$ *(right).*

TABLE 9
Largest connected components of the networks from Stanford Large Network Dataset Collection.

	n	nnz	Description
com-DBLP	3.17080e5	2.41681e6	DBLP collaboration network
web-NotreDame	3.25729e5	1.09011e6	Web graph of Notre Dame
amazon0601	4.03364e5	5.28999e6	Amazon product copurchasing network

TABLE 10
Largest connected components of the networks from the UF sparse matrix collection.

	n	nnz	Description
333SP	3.71282e6	2.22173e7	2D FE triangular meshes
belgium_osm	1.44129e6	3.09994e6	Belgium street network
M6	3.50177e6	2.10038e7	2D FE triangular meshes
NACA0015	1.03918e6	6.22963e6	2D FE triangular meshes
netherlands_osm	2.21669e6	4.88247e6	Netherlands street network
packing-500x100x100-b050	2.14584e6	3.49765e7	DIMACS Implementation Challenge
roadNet-CA	1.95703e6	5.52078e6	California road network
roadNet-PA	1.08756e6	3.08303e6	Philadelphia road network
roadNet-TX	1.35114e6	3.75840e6	Texas road network
fl2010	4.84466e5	2.83072e6	Florida census 2010
as-Skitter	1.69642e6	2.21884e7	Autonomous systems by Skitter
hollywood-2009	1.06913e6	1.13682e8	Hollywood movie actor network

and from the University of Florida Sparse Matrix Collection (UF) [16]. We selected graphs that are ill-conditioned and have relatively higher density. Those graphs are quite challenging for standard AMG methods.

We preprocessed the graphs as follows. The largest connected component of each graph is extracted, any self-loops from the extracted component are discarded, and edge weights of the component are modified to be their absolute values to satisfy the requirements of the path cover finding algorithm. We also made the largest connected components undirected if they were originally directed. In Tables 9 and 10, the basic information of graphs collected from two sources is presented after preprocessing.

The results are presented in Tables 11 and 12. As we can see, for those graph Laplacians corresponding to more complicated (real-life) graphs whose properties are far from standard, a typical V-cycle UA-AMG method struggles to converge and actually fails to converge for more than half of the graphs with low-frequency right-hand side and zero-sum random right-hand side. However, our PC-αAMG converges in fewer than 20 iterations for all tested graphs and building the coarse grid hierarchy needs 10 re-setups on average, which demonstrates the effectiveness of PC-αAMG. Moreover, the average operator complexity in our adaptive algorithm is just slightly

TABLE 11
Performance of solving graphs collected from UF and Stanford with low-frequency $\boldsymbol{b}$*, tol*$=10^{-6}$.

	UA-AMG w/MWM				Algorithm 3.4(1)			Algorithm 3.4(2)			
	Iter	ConvF	t	OC	Iter	t	OC	Iter	Re	t	OC
UF large network datasets collection											
333SP	–	0.997	–	1.89	9	1125.69	2.01	14	7	958.31	2.08
belgium_osm	1629	0.996	553.67	1.99	11	270.80	2.02	15	9	248.65	2.02
M6	–	0.997	–	1.86	10	1464.31	2.11	15	8	1268.61	2.11
NACA0015	–	0.995	–	1.86	9	313.54	2.10	14	7	284.47	2.10
netherlands_osm	–	0.997	–	1.98	10	387.94	2.02	16	9	418.64	2.02
packing	–	0.999	–	1.06	11	1623.80	2.46	19	10	1624.28	2.46
roadNet-CA	878	0.991	458.47	2.05	8	294.54	2.08	14	7	323.46	2.08
roadNet-PA	1382	0.991	294.36	2.05	8	157.31	2.10	14	7	166.42	2.09
roadNet-TX	1424	0.994	460.09	2.04	9	228.66	2.08	14	7	195.26	2.08
fl2010	–	0.998	–	1.83	9	121.48	2.19	15	7	102.18	2.19
as-Skitter	–	0.998	–	1.21	10	393.75	3.13	19	8	393.86	3.14
hollywood-2009	–	0.999	–	1.01	5	470.46	3.17	11	3	455.01	3.18
Stanford Large Network Dataset Collection											
com-DBLP	297	0.986	41.31	2.01	4	78.45	3.22	11	2	55.48	3.22
web-NotreDame	–	0.999	–	1.26	7	1059.29	2.43	13	6	1026.46	2.40
amazon0601	–	0.998	–	1.58	5	347.60	3.49	12	4	337.97	3.52

TABLE 12
Performance of solving graphs collected from UF and Stanford with zero-sum random $\boldsymbol{b}$*, tol*$=10^{-6}$.

	UA-AMG w/MWM				Algorithm 3.4(1)			Algorithm 3.4(2)			
	Iter	ConvF	t	OC	Iter	t	OC	Iter	Re	t	OC
UF large network datasets collection											
333SP	–	0.997	–	1.89	9	1142.14	2.09	6	1	159.94	2.08
belgium_osm	–	0.996	–	1.99	11	274.49	2.02	15	9	268.83	2.02
M6	–	0.997	–	1.86	8	1146.93	2.11	5	1	174.75	2.11
NACA0015	1565	0.995	770.03	1.86	8	294.76	2.10	5	1	42.66	2.10
netherlands_osm	–	0.997	–	1.98	12	549.34	2.02	17	11	524.62	2.02
packing	–	0.999	–	1.06	11	1777.93	2.46	17	10	1800.77	2.47
roadNet-CA	1308	0.994	492.38	2.08	8	323.84	2.08	15	7	351.60	2.08
roadNet-PA	970	0.991	188.99	2.05	8	171.80	2.09	14	6	139.95	2.08
roadNet-TX	1168	0.992	285.48	2.04	9	248.76	2.08	14	7	204.42	2.08
fl2010	–	0.998	–	1.83	8	111.81	2.19	16	7	104.45	2.19
as-Skitter	–	0.998	–	1.21	10	478.38	3.04	17	7	468.94	3.06
hollywood-2009	–	0.999	–	1.01	7	495.46	3.17	13	5	502.04	3.18
Stanford Large Network Dataset Collection											
com-DBLP	573	0.987	65.25	2.01	4	86.36	3.23	11	3	83.41	3.22
web-NotreDame	–	0.999	–	1.26	7	1121.71	2.47	15	6	1109.56	2.56
amazon0601	–	0.998	–	1.58	6	467.71	3.49	10	4	378.44	3.50

above 2, which suggests that our path covering aggregation scheme keeps the sparsity pattern on the coarse levels relatively well. This fact, together with the usage of V-cycle, makes our adaptive AMG method attractive for computing large-scale graphs.

In Figures 10 and 11, we reported the CPU times (more precisely, CPU times per number of nonzeros) for real-world graphs. We capped the height for all the cases where V-cycle UA-AMG did not converge within 2500 steps. For low-frequency $\boldsymbol{b}$ (Figure 10), we observe that when the density (nnz/n) of the matrix is large, it is more likely that the V-cycle UA-AMG would fail to converge within 2500 iterations. However, PC-αAMG (both Algorithms 3.4(1) and 3.4(2)) converges for all the cases and is faster than the regular AMG for all the tested graphs, especially for denser graphs. Between Algorithms 3.4(1) and 3.4(2), the CPU times are comparable, while Algorithm 3.4(2) is slightly better for some graphs. For randomly generated zero-sum $\boldsymbol{b}$ (Figure 11), the relationship between the convergence of V-cycle UA-AMG and the density of the matrices is more unpredictable. However, we can still observe that our PC-αAMG outperforms V-cycle UA-AMG for all the tested graphs. In addition, Algorithm 3.4(2) seems to be faster than Algorithm 3.4(1) for most of the tested graphs, which guides us to choose Algorithm 3.4(2) over Algorithm 3.4(1) for its flexibility.

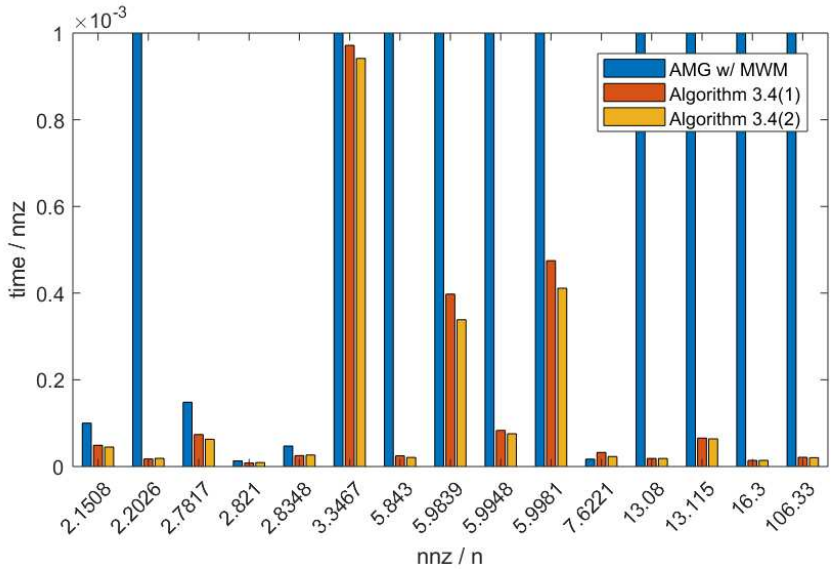

FIG. 10. *CPU time elapsed for solving real-world graphs with low-frequency* $\boldsymbol{b}$ *using regular AMG and PC-*α*AMG Algorithm* 3.4.

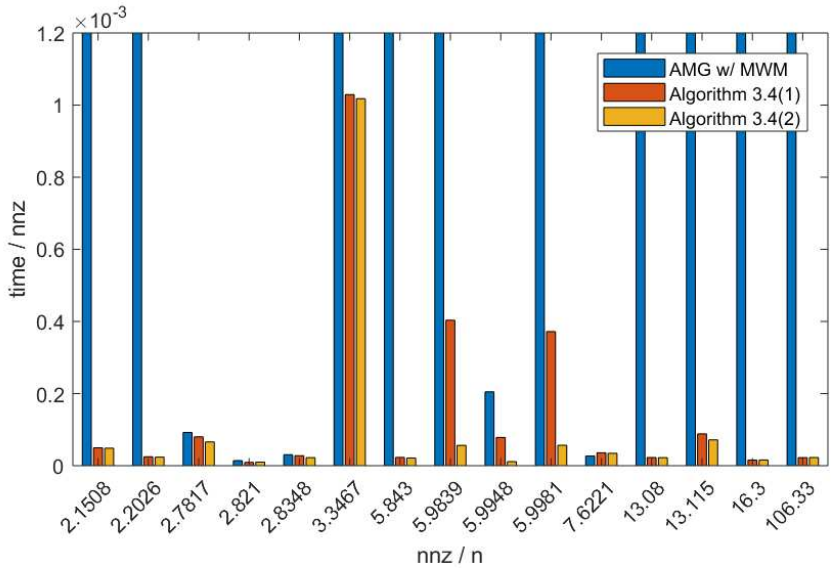

FIG. 11. *CPU time elapsed for solving real-world graphs with randomly generated zero-sum* $\boldsymbol{b}$ *using regular AMG and PC-*α*AMG Algorithm* 3.4.

Overall, our PC-αAMG performs quite robustly and efficiently for graphs from real-world applications, especially the highly ill-conditioned and large-scale graphs.

5. Conclusions. In this paper, we propose the PC-αAMG algorithm for solving linear systems with weighted graph Laplacians. The algorithm uses a novel adaptive strategy and UA-AMG where the aggregations along paths are from an

optimal path cover—specifically, Algorithm 3.4 (with two different re-setup conditions) for solving general $\boldsymbol{b}$ and Algorithm 3.5 for the homogeneous problem. The basic idea relies on adaptive construction of a multilevel hierarchy as follows: (1) use a standard smoother to quickly reduce the high-frequency errors; (2) approximate the (algebraically) smooth errors on an adapted coarse grid using matching. As the error changes during the iterations, the second step may require a re-setup (constructing a new multilevel hierarchy) to efficiently eliminate the current smooth error. We approximate the level set of the smooth error using a path cover and aggregate along the paths (because the error is constant along paths following the level set). We then approximate the smooth error on the coarse grid based on such aggregation. The numerical tests on different model problems show that, after each re-setup, the dominating low-frequency errors are quickly damped (with damping factor < 0.2 on average). Thus, the proposed algorithm effectively eliminates the algebraically smooth errors using several multilevel hierarchies and, according to our numerical experiments, scales nearly optimally with respect to the size of the testing matrices, even when applying standard V-cycle and unsmoothed aggregation schemes.

For the $\boldsymbol{b} = \mathbf{0}$ case (Tables 3 and 8) on each test problem, uniform convergence is observed. The work for generating a new multilevel hierarchy is of order $\mathcal{O}(|V|\log(|V|))$, as the path cover algorithm runs only on the fine level, which costs $\mathcal{O}(|V|\log(|V|))$. We also note that in the numerical tests, the number of re-setups needed is small (three or four on average) and is independent of the size or the type of the model problem considered. Notice that when $\boldsymbol{b} = \mathbf{0}$, the exact error is known and these benchmark problems are just to show how the PC-αAMG works. In addition, in this case, PC-αAMG can also be used as a standalone setup phase for traditional adaptive AMG methods.

In the case of a general right-hand side, the iteration count increases slightly with the matrix size, since we can only use an approximation of the smooth error in this case. Total CPU time scales nearly linearly according to the numerical tests. Solving graph Laplacian systems corresponding to real-world graphs requires flexibility in choosing when to rebuild a new multilevel hierarchy. With such flexible choices, Algorithm 3.4 requires fewer than 20 iterations to achieve the specified tolerance and the number of re-setups remains relatively small, which results in faster CPU time compared to the standard V-cycle UA-AMG. This shows the robustness of our adaptive algorithm for general graph Laplacians. Indeed, we observe such behavior on a wide range of real-world graphs tested (Tables 11 and 12).

While the proposed algorithm clearly has the qualities needed to be useful in practice, we would also like to comment on several ideas that will improve robustness and the efficiency of the PC-αAMG algorithm. In our opinion, it is crucial to design aggregation algorithms which approximate the errors well when $\boldsymbol{b} \neq \mathbf{0}$. As we mentioned, a viable approach for this is to use the adaptive aggregations based on a posteriori error estimates on the underlying graph as proposed in [50]. Another enhancement is to involve more advanced aggregations/cycles/solvers to approximate the smooth error. A combination of such approaches has the potential to provide robust multilevel algorithms for solving linear systems with graph Laplacians. We also want to point out that our algorithm has the potential to tackle time-dependent and nonlinear problems efficiently. A typical situation is when an implicit time stepping or a linearization leads to a sequence of linear systems with graph Laplacians. In such cases, if the near kernels of the corresponding graph Laplacians do not change much with time/nonlinear iterations, then the hierarchical structure can be reused and new AMG setups are not needed. Clearly, in such cases, the resulting PC-αAMG

should perform reasonably well since the space of smooth errors will be (almost) invariant with respect to time. However, if the near kernels do change dramatically as the time/iterations progress, then we cannot reuse the AMG hierarchy and finding a reliable and fast strategy for re-setup constitutes an important research line in the future development of the PC-αAMG method.

Furthermore, there are several interesting questions related to the parallelization of the proposed AMG algorithm on different computer architectures. Most of the components of the algorithm, such as matchings on the path cover and smoothing the error, are suitable for parallelization and can further expedite the algorithm. In section 4, we also showed numerical results demonstrating that the adaptive PC-αAMG algorithm with parallel smoother (such as Jacobi relaxation) works efficiently. PC-αAMG, in its current stage, still needs more work to scale well in a parallel setting. A careful look at the issues related to the parallelization of the PC-αAMG algorithm shows that the challenge is due to the setup phase, which is sequential in our current implmentation (for example, finding a path cover is sequential). For the solve phase, PC-αAMG mainly uses standard V-cycles and/or W-cycles. Therefore, we expect that the solver part would be scalable as traditional AMG methods. Hence, parallelizing the path cover component is the only challenging task in designing fully parallel PC-αAMG. We have encountered several interesting questions in such a process and they are subjects of our current and future research.

REFERENCES

[1] S. Agarwal, K. Branson, and S. Belongie, *Higher order learning with graphs*, in Proceedings of the 23rd International Conference on Machine Learning, ACM, 2006, pp. 17–24.

[2] R. Blaheta, *A multilevel method with overcorrection by aggregation for solving discrete elliptic problems*, J. Comput. Appl. Math., 24 (1988), pp. 227–239, https://doi.org/10.1016/0377-0427(88)90355-X.

[3] S. P. Borgatti, A. Mehra, D. J. Brass, and G. Labianca, *Network analysis in the social sciences*, Science, 323 (2009), pp. 892–895.

[4] A. Brandt, J. Brannick, K. Kahl, and I. Livshits, *Bootstrap AMG*, SIAM J. Sci. Comput., 33 (2011), pp. 612–632.

[5] A. Brandt, S. F. McCormick, and J. W. Ruge, *Algebraic multigrid (AMG) for sparse matrix equations*, in Sparsity and Its Applications, D. J. Evans, ed., Cambridge University Press, Cambridge, UK, 1984.

[6] J. Brannick, Y. Chen, J. Kraus, and L. Zikatanov, *Algebraic multilevel preconditioners for the graph laplacian based on matching in graphs*, SIAM J. Numer. Anal., 51 (2013), pp. 1805–1827.

[7] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge, *Adaptive smoothed aggregation* (αSA) *multigrid*, SIAM Rev., 47 (2005), pp. 317–346.

[8] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge, *Adaptive algebraic multigrid*, SIAM J. Sci. Comput., 27 (2006), pp. 1261–1286.

[9] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, *Geometric deep learning: Going beyond euclidean data*, IEEE Signal Process. Mag., 34 (2017), pp. 18–42.

[10] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, *Spectral Networks and Locally Connected Networks on Graphs*, preprint, arXiv:1312.6203, 2013.

[11] E. Bullmore and O. Sporns, *Complex brain networks: Graph theoretical analysis of structural and functional systems*, Nature Rev. Neurosci., 10 (2009), pp. 186–198.

[12] M. Cao, C. M. Pietras, X. Feng, K. J. Doroschak, T. Schaffner, J. Park, H. Zhang, L. J. Cowen, and B. J. Hescott, *New directions for diffusion-based network prediction of protein function: Incorporating pathways with confidence*, Bioinformatics, 30 (2014), pp. i219–i227.

[13] M. Cao, H. Zhang, J. Park, N. M. Daniels, M. E. Crovella, L. J. Cowen, and B. Hescott, *Going the distance for protein function prediction: A new distance metric for protein interaction networks*, PLOS One, 8 (2013), e76339.

[14] C. Colley, J. Lin, X. Hu, and S. Aeron, *Algebraic multigrid for least squares problems on graphs with applications to hodgerank*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, 2017, pp. 627–636, https://doi.org/10.1109/IPDPSW.2017.163.

[15] P. D'Ambra and P. S. Vassilevski, *Adaptive AMG with coarsening based on compatible weighted matching*, Comput. Vis. Sci., 16 (2013), pp. 59–76.

[16] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011).

[17] Z. Galil, S. Micali, and H. Gabow, *An $\mathcal{O}(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs*, SIAM J. Comput., 15 (1986), pp. 120–130.

[18] X. He, D. Cai, and P. Niyogi, *Tensor subspace analysis*, in Advances in Neural Information Processing Systems, 2006, pp. 499–506.

[19] M. Henaff, J. Bruna, and Y. LeCun, *Deep Convolutional Networks on Graph-Structured Data*, preprint, arXiv:1506.05163, 2015.

[20] A. N. Hirani, K. Kalyanaraman, and S. Watts, *Least Squares Ranking on Graphs*, preprint, arXiv:1011.1716, 2011.

[21] Å. J. Holmgren, *Using graph models to analyze the vulnerability of electric power networks*, Risk Analysis, 26 (2006), pp. 955–969.

[22] X. Hu, P. S. Vassilevski, and J. Xu, *Comparative convergence analysis of nonlinear amlicycle multigrid*, SIAM J. Numer. Anal., 51 (2013), pp. 1349–1369.

[23] B. Jiang, C. Ding, J. Tang, and B. Luo, *Image representation and learning with graph-laplacian Tucker tensor decomposition*, IEEE Trans. Cybernet., 49 (2018), pp. 1417–1426, https://doi.org/10.1109/TCYB.2018.2802934.

[24] X. Jiang, L.-H. Lim, Y. Yao, and Y. Ye, *Statistical ranking and combinatorial hodge theory*, Math. Program., 127 (2011), pp. 203–244.

[25] H. Kim, J. Xu, and L. Zikatanov, *A multigrid method based on graph matching for convection-diffusion equations*, Numer. Linear Algebra Appl., 10 (2003), pp. 181–195, https://doi.org/10.1002/nla.317.

[26] T. N. Kipf and M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, preprint, arXiv:1609.02907, 2016.

[27] I. Koutis, G. L. Miller, and D. Tolliver, *Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing*, Comput. Vision Image Understanding, 115 (2011), pp. 1638–1646.

[28] B. Landa and Y. Shkolnisky, *The Steerable Graph Laplacian and Its Application to Filtering Image Data-sets*, CoRR abs/1802.01894, 2018.

[29] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*, http://snap.stanford.edu/data, (2014).

[30] R. Li, S. Wang, F. Zhu, and J. Huang, *Adaptive Graph Convolutional Neural Networks*, preprint, arXiv:1801.03226, 2018.

[31] J. Lin, L. Cowen, B. Hescott, and X. Hu, *Computing the diffusion state distance on graphs via algebraic multigrid and random projections*, Numer. Linear Algebra Appl., 25 (2018), e2156, https://doi.org/10.1002/nla.2156.

[32] O. E. Livne and A. Brandt, *Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver*, SIAM J. Sci. Comput., 34 (2012), pp. B499–B522.

[33] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.

[34] S. MacLachlan, T. Manteuffel, and S. McCormick, *Adaptive reduction-based AMG*, Numer. Linear Algebra Appl., 13 (2006), pp. 599–620.

[35] T. Manteuffel, S. McCormick, M. Park, and J. Ruge, *Operator-based interpolation for bootstrap algebraic multigrid*, Numer. Linear Algebra Appl., 17 (2010), pp. 519–537.

[36] S. Moran, I. Newman, and Y. Wolfstahl, *Approximation algorithms for covering a graph by vertex-disjoint paths of maximum total weight*, Networks, 20 (1990), pp. 55–64.

[37] A. Narita, K. Hayashi, R. Tomioka, and H. Kashima, *Tensor factorization using auxiliary information*, Data Mining Knowledge Discovery, 25 (2012), pp. 298–324.

[38] Y. Notay, *An aggregation-based algebraic multigrid method*, Electron. Trans. Numer. Anal., 37 (2010), pp. 123–146.

[39] O. Ore, *Arc coverings of graphs*, Ann. Mat. Pura Appl., 55 (1961), pp. 315–321.

[40] J. Ruge and K. Stüben, *Efficient solution of finite difference and finite element equations by algebraic multigrid (AMG)*, in Multigrid Methods for Integral and Differential Equations, D. J. Paddon and H. Holsten, eds., Clarendon Press, Oxford, UK, 1985, pp. 169–212.

[41] J. W. Ruge, *Algebraic multigrid (AMG) for geodetic survey problems*, in Prelimary Proceedings of the International Multigrid Conference, Fort Collins, CO, 1983.

[42] J. W. Ruge and K. Stüben, *Algebraic multigrid*, in Multigrid Methods, Frontiers in Appl. Math. 3, SIAM, Philadelphia, 1987, pp. 73–130.
[43] U. Shaham, K. Stanton, H. Li, B. Nadler, R. Basri, and Y. Kluger, *Spectralnet: Spectral Clustering Using Deep Neural Networks*, preprint, arXiv:1801.01587, 2018.
[44] J. C. Urschel, X. Hu, J. Xu, and L. Zikatanov, *A simple cascade algorithm for computing the Fiedler vector of graph Laplacians*, J. Comput. Math., 33 (2015), pp. 209–226.
[45] P. Van, M. Brezina, and J. Mandel, *Convergence of algebraic multigrid based on smoothed aggregation*, Numer. Math., 88 (2001), pp. 559–579.
[46] P. Vaněk, J. Mandel, and M. Brezina, *Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems*, Computing, 56 (1996), pp. 179–196.
[47] P. S. Vassilevski, *Multilevel Block Factorization Preconditioners*: Springer, New York, 2008. Matrix-Based Analysis and Algorithms for Solving Finite Element Equations, in Multigrid Methods for Integral and Differential Equations, D. J. Paddon and H. Holsten, eds., Clarendon Press, Oxford, UK, 1985, pp. 169–212.
[48] D. J. Watts and S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature, 393 (1998), pp. 440–442.
[49] J. Xu and L. Zikatanov, *Algebraic multigrid methods*, Acta Numeri., 26 (2017), pp. 591–721.
[50] W. Xu and L. T. Zikatanov, *Adaptive aggregation on graphs*, J. Comput. Appl. Math., 340 (2018), pp. 718–730, https://doi.org/https://doi.org/10.1016/j.cam.2017.10.032.
[51] P. Yang, R. A. Freeman, G. J. Gordon, K. M. Lynch, S. S. Srinivasa, and R. Sukthankar, *Decentralized estimation and control of graph connectivity for mobile sensor networks*, Automatica, 46 (2010), pp. 390–396.