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 of weighted graph Laplacians and 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 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 manner, we are able to build a multilevel structure at a low computational cost. The proposed PC-αAMG provides a mechanism to efficiently re-build 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 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

1. Introduction. As weighted graphs frequently being employed as the 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. In recent works, solving weighted graph Laplacians has been applied to solve ranking and user recommendation problems [20, 24, 14]. In [12, 13, 31], similarities of proteins are calculated by solving the graph Laplacian associated with the protein interaction network, which are further used in clustering and labeling proteins’ by their functions. The marriage of graph Laplacian and heated computer science topics such as Convolutional Neural Networks and tensor decomposition has also been a trend [18, 9, 26, 19, 10, 43, 30]. Researchers advanced algorithms that adapt graph Laplacian properties to improve tasks including image reconstruction, clustering image data-sets and classification [1, 37, 23, 28]. To efficiently solve graph Laplacians, algebraic multigrid (AMG) is often applied. The standard AMG method was proposed to solve partial differential equations (PDEs) and involves mainly 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 is proven to be one of the most successful iterative methods in practical applications and many AMG methods have been developed to solving graph Laplacians, such as combinatorial multigrid [27], Lean AMG [32]. 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 convergence rate
is slow, which leads to computational waste. To better reduce the computational redundancy, adaptivity becomes essential. Substantial efforts have been made to incorporate adaptivity in iterative methods. Date back to 1984, the original adaptive AMG algorithm [5] was proposed, which laid foundation for the development of self-learning and bootstrap AMG. The bootstrap AMG approach was further developed in [35, 4] which 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-nullspace [7]. In the following year, the same authors further proposed an operator-induced interpolation approach that automatically represent smooth components [8]. In both of the literatures, the basic idea is to solve the homogeneous problem \((Ax = 0)\) to expose the slow-to-converge errors, coarsening based on algebraically smooth errors, and then applying back to improve the AMG process. MacLachlan et al. 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 UA-AMG method is fairly recent. In [15], D’Ambra et al. applied coarsening scheme that uses compatible weighted matching, which avoids the reliance on the characteristics of connection strength in defining both the coarse space and the interpolation operators. Through previous endeavors, authors enlightened different ideas on how to build smooth errors into the ranges of adaptively constructed interpolation operators, assuming coarsening is done. This motivates us to the idea of using the smooth error for not only building interpolation operators but also determining coarsening scheme.

In this paper, an adaptive AMG method based on path cover is presented. Specifically, we choose to use UA-AMG [2] for its low computational complexity, which is favorable for parallel computing. We setup the multilevel structure by first applying path cover [39] to approximate 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, therefore is useful for parallel computing, especially on solving matrices with large condition numbers. While the existed adaptive methods need to first solve the homogeneous problem, \((Ax = 0)\), to determine the near-null-space components and use this knowledge to solve \((Ax = b)\), our adaptive method solves \((Ax = b)\) directly, for a general right-hand-side \(b\).

The paper is arranged as follows. In section 2, basic subroutines, such as a modified version of path cover finding algorithm, are presented. Our main adaptive AMG algorithm is discussed in section 3. Numerical results are demonstrated and analyzed in section 4. Finally, in section 5, we conclude the main contributions of this paper 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 maximal independent set [33], 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. Here, we briefly recall MWM, since in the implementations, we adopt this aggregating scheme on weighted adjacency matrix $A$ of each model graph as a default and set the performance benchmark of regular AMG method.

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 et al. 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, $\tilde{S}$ of $G$, in $O(|E| \cdot \log |E|)$ time, such that

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

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 Path Cover**

1: procedure $[\text{cover}] = \text{PathCover}(A)$
2: Input: $A$ - adjacency matrix of an undirected positive weighted graph $G = (V, E, \omega)$
3: Output: $\text{cover}$ - a path cover of graph $G$

4: sorted_edges $\leftarrow$ Sort the edges in descending order based on weights.
5: for $e(u, v) \in$ sorted_edges do
6: if both $u$ and $v$ are endpoints then
7: if neither $u$ nor $v$ is in any paths then
8: Add $\{u, v\}$ as a new path in $\text{cover}$
9: else if $u$ is in a path, but $v$ is not then
10: Append $\{v\}$ to cover\{path that contains $u$\}
11: else if $v$ is in a path, but $u$ is not then
12: Append $\{u\}$ to cover\{path that contains $v$\}
13: else $\triangleright$ ($u$ and $v$ are in different nonzero-length paths)
14: Merge two paths
15: end if
16: end if
17: end for
18: end procedure
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 $O(|V| \log |V|)$ when the graph is sparse, $O(|E|) = O(|V|)$. 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. Therefore, 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 PC-\(\alpha\)AMG are presented in this section. In subsection 3.1, the main idea of the algorithm is presented and demonstrated with an intuitive example. subsection 3.2 and subsection 3.3 comprise essential subroutines of approximating smooth error using previously built multilevel hierarchies (Algorithm 3.1), building aggregations and prolongation operator from path cover (Algorithm 3.2), and setting up multilevel structure using path cover and approximated error (Algorithm 3.3). Finally, the full PC-\(\alpha\)AMG algorithm is presented in subsection 3.4.

3.1. Basic Ideas and Rationale of the Adaptive Algorithm. As previously 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 preserving the smooth error well onto the coarse levels, so that the adaptively designed multilevel hierarchy can eliminate the current smooth error efficiently on the coarsest level. Specifically, we utilize path cover finding algorithm 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 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 desired accuracy is met. In our opinion, it could be beneficial to have multilevel hierarchies which approximate the errors well when \(b \neq 0\). 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, PC-\(\alpha\)AMG algorithm proposed here integrates the setup and solve phase together by identifying the smooth errors while solving the linear system $Ax = b$.

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 (Figure 2 lower right).
right). The norm of difference between the two smooth errors is within $10^{-10}$. 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. In subsection 3.2, we include procedures that can obtain the estimated smooth error when solving slows down, which is symbolized by the manufactured error (Figure 2 upper left). We present subroutines in subsection 3.3 to automatically aggregate along each path in path cover that approximates the level sets of the approximated smooth error. It is verifiable that we can achieve similar effect as 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 on the coarse levels rely on the fact that the smooth error itself can be well approximated on the fine level, one essential step of our algorithm is to approximate the smooth error accurately when the convergence rate becomes slow. The fact is, during the iterations, the approximated solution $x_k$ is available at hands, but not straight-forwardly the error $e_k$. Based on the well-known error equation $Ae_k = r_k = b - Ax_k$, we try to approximately solve the error equation by applying several steps of multigrid preconditioned conjugate gradient (PCG) method and detailed approach is presented in Algorithm 3.1. For the first several times of re-setups, we use several W-cycles to get a good approximation (this corresponds to the case “$\text{Re} \leq \text{numW}$”). In practice, we minimized 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]:

$$e \leftarrow \prod_{j=0}^{2\text{Re}+1} (I - B_j A)e,$$

where $\text{Re}$ is the number of re-setups, and $B_{\text{Re}+j} = B_{\text{Re}+1-j}$, $j = 1, \ldots, \text{Re} + 1$. Each $B_j$ corresponds to the preconditioning effect of multilevel hierarchy built from the $j$-th re-setup, i.e. $\text{P_hist}(j)$. In this manner, we recycle all the hierarchies ever built and reduce the computational waste.
Algorithm 3.1 Approximate Smooth Error

```
1: procedure \[ e \] = \textsc{ApproximateSmoothError}(A, \{A_\ell\}_{\ell=2}^L, r, P\_hist, Re, e)
2: Inputs:
   a: \( r \) - residual vector
   b: \( P\_hist \) - all the hierarchies ever created
   c: \( Re \) - re-setup count

3: Choose parameters:
   a: \( \text{numW} \) \quad \triangleright \text{Number of smooth errors (re-setups) using W-cycle}
   b: \( \text{iterW} \) \quad \triangleright \text{Number of iterations of W-cycle PCG}
   c: \( \text{iterV} \) \quad \triangleright \text{Number of iterations of V-cycle PCG}

4: if \( Re \leq \text{numW} \) then
5:   for \( i \leftarrow 1 : \text{iterW} \) do
6:     \( e \leftarrow \text{W\_cycle\_PCG}(A, r, e, P\_hist\{Re\}, \{A_\ell\}_{\ell=2}^L) \)
7:   end for
8: else
9:   for \( i = 1 : \text{iterV} \) do
10:  \( e \leftarrow \text{V\_cycle\_PCG}(A, r, e, P\_hist\{1:Re\}, \{A_\ell\}_{\ell=2}^L) \)
11: end for
12: end if
13: Normalize \( e \)
14: Make \( e \) orthogonal to \( 1 \) \quad \triangleright \text{This is because we are solving Graph Laplacian}
15: 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 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. \( A_{ij} = 1/\|e_i - e_j\| \), so that edges between nodes of similar values (in another word, those nodes should be on the same level set) have heavy weights and are more likely to be chosen to form the path cover. However, the edges 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 \( A^2 \) to include more edges while keeping the sparsity of the graph relatively low (For the example shown in Figure 3, sparsity pattern of \( A^2 \) 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 approximate level sets of the smooth error as we expected. Note that, since the smooth error are
Algorithm 3.2 Path Cover Aggregation

1: procedure $[ P ] = \text{PATHCOVERAGGREGATE}(\text{cover}, \tilde{A}, \ell, e)$
2: Input:
   a: $\tilde{A}$ - graph Laplacian (possibly reweighted)
   b: $\ell$ - level count
   c: $e$ - smooth error
3: Output: $P$ - prolongation operator
4: $n \leftarrow$ number of nodes
5: for each path in cover do
6:   Match each vertex with one of its neighbor
7: end for
8: $\text{numAgg} \leftarrow$ number of paths in cover
9: for each unaggregated vertex $u$ do
10:   $v \leftarrow u$’s neighbor with the maximal edge weight based on $\tilde{A}$
11:   if $v$ is also unaggregated then
12:      Match $u$ and $v$ together
13:   else if size(aggregation of $v$) = 2 then
14:      Aggregate $u$ into $v$’s aggregation
15:   else
16:      Match $u$ and $v$ and remove $v$ from previous aggregation of $v$
17:      $\text{numAgg} \leftarrow \text{numAgg} + 1$
18: end if
19: end for
20: if $\ell = 1$ then
21:   Build prolongation operator $P \in \mathbb{R}^{n \times \text{numAgg}}$ using smooth error $e$
22: else
23:   Build boolean prolongation operator $P \in \mathbb{R}^{n \times \text{numAgg}}$
24: end if
25: end procedure

Algorithm 3.3 Path Cover AMG Setup

1: procedure $[ \{A_\ell\}_{\ell=1}^L, \{P_\ell\}_{\ell=1}^L ] = \text{PATHCOVERAMG}_{\text{setup}}(A, e)$
2: $\tilde{A}_{ij} \leftarrow 1/\|e_i - e_j\|$, if edge $e(i,j)$ is in the graph induced by $A^2$
3: $\text{cover} \leftarrow \text{PathCover}(\tilde{A})$
4: $A_1 \leftarrow A$
5: for $\ell \leftarrow 1 : L$ do
6:   $P_\ell \leftarrow \text{PathCoverAggregate}(\text{cover}, \tilde{A}, \ell, e)$
7:   $\tilde{A} \leftarrow P_\ell^T \tilde{A} P_\ell$
8:   Shorten each path in cover by merging the vertices in one aggregate into a vertex
9:   $A_{\ell+1} \leftarrow P_\ell^T A_\ell P_\ell$
10: end for
11: end procedure

not perfectly linear near the boundaries, the level sets are not strictly diagonals.

As previously demonstrated, there might be isolated points resulting from creating
Aggregating the isolated points

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 is no isolated vertex and the resulting aggregations are all of size 2 or 3.

Here, 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 coarse-level path cover and recurse this process for all coarse levels.

Also notice that we use the smooth error again in building prolongation operator in Algorithm 3.2, namely, the prolongation operator $P_\ell$ is defined as

$$ (P_\ell)_{ki} = \begin{cases} e_k & \text{if } k \in V_i^\ell, \ell = 1, \\ 1 & \text{if } k \in V_i^\ell, \ell > 1, \\ 0 & \text{otherwise}, \end{cases} $$

where $\ell$ is the level count and $e_k$ is the $k$th entry of the (approximate) smooth error $e$. Obviously, we have $e \in \text{Range}(P_1)$.

3.4. Main Algorithm. The overall PC-\(\alpha\)AMG algorithms are present in Algorithm 3.4 and Algorithm 3.5. Algorithm 3.4 is for the general case $Ax = b$. Algorithm 3.5 is a special case for solving $Ax = 0$. One reason we include the homogeneous case is that we have the knowledge that $x = 0$ is the true solution in this case, so that the process of attaining smooth error is much simpler. Another reason is that following the basic idea of adaptive AMG methods, Algorithm 3.5 can also be used as standalone setup phase and build several multilevel structures that are effective for eliminating smooth errors. The two versions of algorithms include all of the aforementioned components, yet the main procedure of both versions is rather straight-forward. We use MWM to build the initial multilevel structure and use it as the preconditioner to solve the model problem. For simple graph Laplacian problems, MWM might already give good performance, then there is no need to use the adaptive procedure. Our PC-\(\alpha\)AMG aims at difficult problems which have error components that are slow-to-converge for MWM, the current multilevel hierarchy is not desirable anymore. We then re-setup using aggregations based on path cover of the current smooth error. This is done whenever the convergence rate becomes higher than a certain pre-described threshold. Such adaptive approach efficiently eliminates slow-to-converge smooth errors. We use Figure 5 to demonstrate our PC-\(\alpha\)AMG.

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 performance of V-cycle UA-AMG using MWM coarsening scheme and Gauss-Seidel smoother with proposed Algorithm 3.4 and Algorithm 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 forces the multilevel hierarchy to be re-built 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 every iteration is
10
X. Hu, J. Lin, and L. T. Zikatanov

Fig. 5. Procedure of PC-αAMG

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

For other parameters used in Algorithm 3.4, we set numW = 2, iterW = 7 and iterV = 4 to generate the approximated smooth error. That is, we use W-cycle 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 \( b \in \mathbb{R}^{|V|} = [1, 1, \ldots, 1 - |V|]^T \), a randomly generated \( b \) with zero sum, and \( b = 0 \). For all the experiments below, the solver stops when the residual reaches the pre-described tolerance. Basically, we use a relatively small tolerance \( 10^{-8} \) for the scalability tests in subsection 4.1 and subsection 4.2 and more practical choice \( 10^{-6} \) for real world graphs. “–” in any columns of tables means the number of iterations exceeds 2500.

Numerical experiments are conducted using an 3GHz Intel Xeon ‘Sandy Bridge’ CPU with 256 GB of RAM. The software used is an algebraic multigrid package written by the authors and implemented in Matlab. We report the following results in each table. The “ConvR” under MWM column is the algebraic average of convergence rate for the last 10 iterations, and since we restart whenever convergence rate at current iteration or on average is above a threshold, we omit this factor for our adaptive algorithms. The “Iter” column reports number of iterations needed for the residual to reach certain tolerance, and “Re” records the number of re-setups needed specifically for Algorithm 3.4 with high threshold 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 in Algorithm 3.4 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 unweighted graph Laplacians of 2D regular uniform grids in Table 1-Table 3. This is related to solving a Poisson equation with Neumann boundary condition on a 2D square.
Notice that for regular grids, while the numbers of iterations for regular V-cycle UA-AMG in all cases grow rapidly and quickly exceed 2500, number of iterations of Algorithm 3.4 in Table 1 and Table 2 is nearly uniform and in Table 3 is uniform. Total CPU time is plotted in Figure 6 and we can see that the total CPU time of PC-αAMG increases nearly linearly with respect to the matrix size for both homogeneous $b$ and non-zero $b$ (the line with slope 1 is added for reference). The growth rate of the CPU time is slightly slower than linear for small $n$, which is probably due to the fact that overhead cost is more pronounced than actually computing cost when $n$ is small. But for large $n$, we can see the nearly linear growth asymptotically, which demonstrate 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 less times than Algorithm 3.4(1). This justifies our choices of different threshold.

4.2. Tests for Ring Graphs. For the second example, we use Watts Strogatz [48] model and set the rewiring probability $\beta = 0$ and the mean node degree to be 4 in order to produce ring graphs as in Figure 7. The condition numbers of the graph Laplacians of the ring graphs also grow rapidly when the size of the graphs increases.

The results are present in Table 4, Table 5 and Table 6. Like the performance
of solving the regular grids, PC-αAMG method on ring graphs requires small numbers of iterations and re-setups to converge, while the standard V-cycle UA-AMG eventually cannot converge within 2500 iterations for large graphs. In Figure 8, we report the total CPU time for both zero and nonzero $b$. The results are similar to the regular grids’ results. When matrix size is small, the CPU time grows slower than linear. Asymptotically, the CPU time grows nearly linear with respect to $n$, which demonstrate the nearly optimal computational complexity of our PC-αAMG. Moreover, compared to Algorithm 3.4(1), Algorithm 3.4(2) still takes slightly shorter CPU time for ring graphs.

4.3. Tests for Real World Graphs. Besides the graphs generated above, we also tested real world graphs from Stanford large network datasets collection [29] 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
AN ADAPTIVE AMG BASED ON PATH COVER

Table 4
Performance of Solving Ring Graphs with Low-Frequency \( b \), \( \text{tol}=1e-8 \)

| \( n \) | UA-AMG w/MWM | Algorithm 3.4(1) | Algorithm 3.4(2) |
|---|---|---|---|
| 2\(^{10} \) | 149 | 0.894 | 1.59 |
| | 7 | 1.57 | 14 | 6 | 1.57 |
| 2\(^{12} \) | 593 | 0.974 | 1.59 |
| | 10 | 1.59 | 19 | 9 | 1.59 |
| 2\(^{14} \) | 2292 | 0.993 | 1.60 |
| | 12 | 1.60 | 21 | 11 | 1.59 |
| 2\(^{16} \) | – | 0.998 | 1.60 |
| | 18 | 1.60 | 24 | 15 | 1.60 |

Table 5
Performance of Solving Ring Graphs with Zero-Sum Random \( b \), \( \text{tol}=1e-8 \)

| \( n \) | UA-AMG w/MWM | Algorithm 3.4(1) | Algorithm 3.4(2) |
|---|---|---|---|
| 2\(^{10} \) | 144 | 0.10 | 1.56 |
| | 8 | 1.57 | 14 | 6 | 1.58 |
| 2\(^{12} \) | 599 | 0.974 | 1.59 |
| | 10 | 1.60 | 18 | 10 | 1.60 |
| 2\(^{14} \) | 2250 | 0.993 | 1.60 |
| | 14 | 1.60 | 23 | 12 | 1.60 |
| 2\(^{16} \) | – | 0.998 | 1.60 |
| | 21 | 1.60 | 25 | 18 | 1.60 |

quite challenging for standard AMG method.

We pre-processed 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 path cover finding algorithm. We also made the largest connected components undirected if they are originally directed. In Table 7 and Table 8, the basic information of pre-processed graphs collected from two sources are presented.

![Fig. 9. CPU Time Elapsed for Solving Real World Graphs with Low Frequency \( b \) using regular AMG and PC-\( \alpha \)AMG Algorithm 3.4](image)

This manuscript is for review purposes only.
Table 6
Performance of Solving Ring Graphs with $b = 0$, $tol=1e-8$

| $n$  | UA-AMG w/MWM | Algorithm 3.5 |
|------|--------------|---------------|
|      | $n$ | Iter | ConvR | OC | Iter | Re | OC |
| $2^{10}$ | 67  | 0.881 | 1.50 | 15  | 4   | 1.61 |
| $2^{12}$ | 311 | 0.980 | 1.50 | 15  | 4   | 1.62 |
| $2^{14}$ | 858 | 0.996 | 1.51 | 15  | 4   | 1.63 |
| $2^{16}$ | 995 | 0.997 | 1.51 | 15  | 4   | 1.63 |

Table 7
Largest connected components of the networks from Stanford large network datasets collection

|   | $n$ | nnz | mnz | 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 co-purchasing network |

Fig. 10. CPU Time Elapsed for Solving Real World Graphs with Randomly Generated Zero-Sum $b$ using regular AMG and PC-$\alpha$AMG Algorithm 3.4

The results are presented in Table 9 and Table 10. As we can see, for those graph Laplacians corresponding to more complicated (real-life) graphs, whose properties are far from standard, standard V-cycle UA-AMG method struggles to converge and actually fail 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-$\alpha$AMG converges within less than 20 iterations for all tested graphs and building the coarse grid hierarchy need 10 re-setups on average, which demonstrates the effectiveness of PC-$\alpha$AMG. Moreover, the average operator complexity in our adaptive algorithm is just slightly 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
Table 8
Largest connected components of the networks from the University of Florida sparse matrix collection (UF)

| Description          | nnz   | nnz   | Description                                      |
|----------------------|-------|-------|-------------------------------------------------|
| 333SP                | 3.71282e6 | 2.21175e7 | 2-dimensional FE triangular meshes             |
| belgium_osm          | 1.144129e6 | 3.09994e6 | Belgium street network                          |
| M6                   | 3.50177e6 | 2.10038e7 | 2-dimensional FE triangular meshes              |
| NACA0015             | 1.03918e6 | 6.22963e6 | 2-dimensional FE triangular meshes              |
| packing-500x100x100-b050 | 2.21669e6 | 4.88247e6 | Netherlands street network                      |
| roadNet-CA           | 1.35114e6 | 3.75840e6 | Texas road network                               |
| roadNet-PA           | 1.95703e6 | 5.52078e6 | Philadelphia road network                       |
| roadNet-TX           | 1.135114e6 | 3.75840e6 | Texas road network                               |
| fl2010               | 1.169642e6 | 2.21884e7 | Autonomous systems by Skitter                   |
| hollywood-2009       | 1.06913e6 | 1.13682e8 | Hollywood movie actor network                   |

Table 9
Performance of Solving Graphs Collected from UF and Stanford with Low-Frequency $b$, tol=$1e$-$6$

| UA-AMG w/MWM | Algorithm 3.4(1) | Algorithm 3.4(2) |
|--------------|-------------------|-------------------|
|              | Iter  | ConvR | OC | Iter  | OC  | Iter  | Re  | OC  |
| UF large network datasets collection |
| 333SP        | –     | 0.997 | 1.89 | 9     | 2.01 | 14    | 7   | 2.08 |
| belgium_osm  | 1629  | 0.996 | 1.99 | 11    | 2.02 | 15    | 9   | 2.02 |
| M6           | –     | 0.997 | 1.86 | 10    | 2.11 | 15    | 8   | 2.11 |
| NACA0015     | –     | 0.995 | 1.86 | 9     | 2.10 | 14    | 7   | 2.10 |
| netherlands_osm | –     | 0.997 | 1.98 | 10    | 2.02 | 16    | 9   | 2.02 |
| packing      | –     | 0.999 | 1.06 | 11    | 2.46 | 19    | 10  | 2.46 |
| roadNet-CA   | 878   | 0.991 | 2.05 | 8     | 2.08 | 14    | 7   | 2.08 |
| roadNet-PA   | 1382  | 0.991 | 2.05 | 8     | 2.10 | 14    | 7   | 2.09 |
| roadNet-TX   | 1424  | 0.994 | 2.04 | 9     | 2.08 | 14    | 7   | 2.08 |
| fl2010       | –     | 0.998 | 1.83 | 9     | 2.19 | 15    | 7   | 2.19 |
| as-Skitter   | –     | 0.997 | 1.21 | 10    | 3.13 | 19    | 8   | 3.14 |
| hollywood-2009 | –     | 0.999 | 1.01 | 5     | 3.17 | 11    | 3   | 3.18 |
| Stanford large network datasets collection |
| com-DBLP      | 297   | 0.986 | 2.01 | 4     | 3.22 | 11    | 2   | 3.22 |
| web-NotreDame | –     | 0.999 | 1.26 | 7     | 2.43 | 13    | 6   | 2.40 |
| amazon0601   | –     | 0.998 | 1.58 | 5     | 3.49 | 12    | 4   | 3.52 |

of V-cycle, makes our adaptive AMG method attractive for parallel computing and large-scale graphs.

In Figure 9 and Figure 10, we report the CPU time (more precisely, CPU time per number of non-zeros) 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 $b$ (Figure 9), we can observe that when the density (nnz/n) of the matrix is large, it is more likely that the V-cycle UA-AMG fails to converge within 2500 iterations. However, PC-$\alpha$AMG (both Algorithm 3.4(1) and Algorithm 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 Algorithm 3.4(1) and Algorithm 3.4(2), the CPU times are comparable while Algorithm 3.4(2) is slightly better for some graphs. For randomly generated zero-sum $b$ (Figure 10), the relationship between the convergence of V-cycle UA-AMG and the density of the matrices is more unpredictable. However, we can still
Table 10

| Algorithm 3.4(1) | Algorithm 3.4(2) |
|------------------|------------------|
| Iter | ConvR | OC | Iter | OC | Iter | Re | OC |
| UF large network datasets collection |
| 333SP | - | 0.997 | 1.89 | 9 | 2.09 | 6 | 1 | 2.08 |
| belgium_osm | - | 0.996 | 1.99 | 11 | 2.02 | 15 | 9 | 2.02 |
| M6 | - | 0.997 | 1.86 | 8 | 2.11 | 5 | 1 | 2.11 |
| NACA0015 | 1565 | 0.995 | 1.86 | 8 | 2.10 | 5 | 1 | 2.10 |
| netherlands_osm | - | 0.997 | 1.98 | 12 | 2.02 | 17 | 11 | 2.02 |
| packing | - | 0.999 | 1.06 | 11 | 2.46 | 17 | 10 | 2.47 |
| roadNet-CA | 1308 | 0.994 | 2.08 | 8 | 2.08 | 15 | 7 | 2.08 |
| roadNet-PA | 970 | 0.991 | 2.05 | 8 | 2.09 | 14 | 6 | 2.08 |
| roadNet-TX | 1168 | 0.992 | 2.04 | 9 | 2.08 | 14 | 7 | 2.08 |
| fl2010 | - | 0.998 | 1.83 | 8 | 2.19 | 16 | 7 | 2.19 |
| as-Skitter | - | 0.998 | 1.21 | 10 | 3.04 | 17 | 7 | 3.06 |
| hollywood-2009 | - | 0.999 | 1.01 | 7 | 3.17 | 13 | 5 | 3.18 |
| Stanford large network datasets collection |
| com-DBLP | 573 | 0.987 | 2.01 | 4 | 3.23 | 11 | 3 | 3.22 |
| web-NotreDame | - | 0.999 | 1.26 | 7 | 2.47 | 15 | 6 | 2.56 |
| amazon0601 | - | 0.998 | 1.58 | 6 | 3.49 | 10 | 4 | 3.50 |

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.

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

5. Conclusions. In this paper, we proposed PC-αAMG, an adaptive UA-AMG algorithm, for solving linear systems based on optimal path cover. 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 error 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 errors. 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 build a coarse space based on such aggregation contains a good approximation of the smooth error. 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 scheme.

For $b = \mathbf{0}$ case (Table 3 and Table 6), on each test problem, uniform convergence is observed. The work for generating a new multilevel hierarchy is of order $O(|V|\log(|V|))$, as the path covering algorithm runs only on the fine level, which costs $O(|V|\log(|V|))$. We also note that in the numerical tests, the number of re-
sets needed is small (4-5 on average) and is independent of the size or the type of the model problem considered. Notice that when \( b = 0 \), the exact error is known and these benchmark problems are just to show how the PC-\( \alpha \)AMG works. In addition, in this case, PC-\( \alpha \)AMG can also be used as standalone setup phase for traditional adaptive AMG methods.

In the case of a non-zero right-hand-side, 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 Laplacians corresponding to real world graphs requires flexibility in choosing when to re-build a new multilevel hierarchy. With such flexible choice, Algorithm 3.4 requires less than 20 iterations to achieve specified tolerance and the number of re-setups remains relatively small, which results in faster CPU time compared to standard V-cycle UA-AMG. This shows the robustness and generality of our adaptive algorithm. We see such behavior on a wide range of matrices (graphs) tested (Table 9 and Table 10).

While the proposed algorithms clearly have the robustness to be practical, we would also like to mention a few future research plans which could improve such robustness. In our opinion, it is crucial to design aggregation algorithms which approximate the errors well when \( b \neq 0 \). As we mentioned earlier, a viable approach for this is to use the adaptive aggregations based on a posteriori error estimates as proposed in [50]. Another enhancement of our consideration 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.

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, http://dx.doi.org/10.1016/0377-0427(88)90355-X. Iterative Methods for the Solution of Linear Systems.
[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 Journal on Scientific Computing, 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, 1984.
[6] J. Brannick, Y. Chen, J. Kraus, and L. Zikatanov, Algebraic multilevel preconditioners for the graph laplacian based on matching in graphs, SIAM Journal on Numerical Analysis, 51 (2013), pp. 1805–1827.
[7] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge, Adaptive smoothed aggregation (aSA) multigrid, SIAM review, 47 (2005), pp. 317–346.
[8] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge, Adaptive algebraic multigrid, SIAM Journal on Scientific Computing, 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 Processing Magazine, 34 (2017), pp. 18–42.
[10] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, Spectral networks and locally connected networks on graphs, arXiv preprint arXiv:1312.6203, (2013).
[11] E. Bullmore and O. Sporns, Complex brain networks: graph theoretical analysis of structural and functional systems, Nature Reviews Neuroscience, 10 (2009), p. 186.
[12] M. Cao, C. M. Pietras, X. Feng, K. J. Doroschak, T. Schaffner, J. Park, H. Zhang, L. J. Cowley, and B. J. Hescott, New directions for diffusion-based network prediction of protein function: incorporating pathways with confidence, Bioinformatics, 30 (2014),
pp. 1219–1227.

[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), p. e76339.

[14] C. COLLEY, J. LIN, X. HU, AND S. AERON, Algebraic multigrid for least squares problems on graphs with applications to hodge rank, in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 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, Computing and Visualization in Science, 16 (2013), pp. 59–76.

[16] T. A. DAVIS AND Y. HU, The university of florida sparse matrix collection, ACM Transactions on Mathematical Software (TOMS), 38 (2011), p. 1.

[17] Z. GALIL, S. MICALI, AND H. GABOW, An $O(\text{EV}^{1/2}\log V)$ algorithm for finding a maximal weighted matching in general graphs, SIAM Journal on Computing, 15 (1986), pp. 120–130.

[18] X. HU, 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, arXiv preprint arXiv:1506.05163, (2015).

[20] A. N. HIRANI, K. KALYANARAMAN, AND S. WATTS, Least squares ranking on graphs, arXiv preprint arXiv:1011.1716, (2011).

[21] Á. J. HOLMGREN, Using graph models to analyze the vulnerability of electric power networks, Risk analysis, 26 (2006), pp. 955–969.

[22] M. HENAFF, J. BRUNA, AND Y. LE CUN, A simple parallel algorithm for the maximal independent set problem, SIAM Journal on Scientific Computing, 34 (2012), pp. B499–B522.

[23] I. KOUTIS, G. L. MILLER, AND D. TOLLIVER, Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing, Computer Vision and Image Understanding, 115 (2011), pp. 1638–1646.

[24] B. LANDA AND Y. SHKOLNISKY, The steerable graph laplacian and its application to filtering image data-sets, CoRR, abs/1802.01894 (2018), http://arxiv.org/abs/1802.01894, https://arxiv.org/abs/1802.01894.

[25] O. E. LIVNE AND A. BRANDT, Lean algebraic multigrid (lamg): Fast graph laplacian linear solver, SIAM Journal on Scientific Computing, 34 (2012), pp. B499–B522.

[26] M. LUBY, A simple parallel algorithm for the maximal independent set problem, SIAM journal on computing, 15 (1986), pp. 1036–1053.

[27] S. MACLACHLAN, T. MANTEUFFEL, AND S. MCCORMICK, Adaptive reduction-based amg, Numerical Linear Algebra with Applications, 13 (2006), pp. 599–620.

[28] T. MANTEUFFEL, S. MCCORMICK, M. PARK, AND J. RUGE, Operator-based interpolation for bootstrap algebraic multigrid, Numerical Linear Algebra with Applications, 17 (2010), pp. 519–537.
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 and Knowledge Discovery, 25 (2012), pp. 298–324.

[38] Y. Notay, An aggregation-based algebraic multigrid method, Electronic transactions on numerical analysis, 37 (2010), pp. 123–146.

[39] O. Ore, Arc coverings of graphs, Annali di Matematica Pura ed Applicata, 55 (1961), pp. 315–321.

[40] J. Ruge and K. Stüben, Efficient solution of finite difference and finite element equations by algebraic multigrid AMG, Gesellschaft f. Mathematik u. Datenverarbeitung, 1984.

[41] J. W. Ruge, Algebraic multigrid (AMG) for geodetic survey problems, in Preliminary Proc. Internat. Multigrid Conference, Fort Collins, CO, 1983.

[42] J. W. Ruge and K. Stüben, Algebraic multigrid, in Multigrid methods, vol. 3 of Frontiers Appl. Math., SIAM, Philadelphia, PA, 1987, pp. 73–130.

[43] U. Shaham, K. Stanton, H. Li, B. Nadler, R. Basri, and Y. Kluger, Spectraltag: Spectral clustering using deep neural networks, arXiv 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, Journal of Computational Mathematics, 33 (2015), pp. 209–226.

[45] P. Van, M. Brezina, J. Mandel, et al., Convergence of algebraic multigrid based on smoothed aggregation, Numerische Mathematik, 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.

[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 Numerica, 26 (2017), pp. 591–721.

[50] W. Xu and L. T. Zikatanov, Adaptive aggregation on graphs, Journal of Computational and Applied Mathematics, 340 (2018), pp. 718–730, https://doi.org/10.1016/j.cam.2017.10.032, http://www.sciencedirect.com/science/article/pii/S0377042717305344. 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.