Learning Algebraic Multigrid Using Graph Neural Networks

Ilay Luz [1] **Meirav Galun** [1] **Haggai Maron** [2] **Ronen Basri** [1] **Irad Yavneh** [3]

Abstract

Efficient numerical solvers for sparse linear systems are crucial in science and engineering. One of the fastest methods for solving large-scale sparse linear systems is algebraic multigrid (AMG). The main challenge in the construction of AMG algorithms is the selection of the prolongation operator—a problem-dependent sparse matrix which governs the multiscale hierarchy of the solver and is critical to its efficiency. Over many years, numerous methods have been developed for this task, and yet there is no known single right answer except in very special cases. Here we propose a framework for learning AMG prolongation operators for linear systems with sparse symmetric positive (semi-) definite matrices. We train a single graph neural network to learn a mapping from an entire class of such matrices to prolongation operators, using an efficient unsupervised loss function. Experiments on a broad class of problems demonstrate improved convergence rates compared to classical AMG, demonstrating the potential utility of neural networks for developing sparse system solvers.

1. Introduction

Algebraic multigrid (AMG) is a well-developed efficient numerical approach for solving large ill-conditioned sparse linear systems and eigenproblems. Introduced in the 1980's (Brandt et al., 1984; Ruge, 1983; Ruge & Stüben, 1987), AMG and its many variants have been applied to diverse problems, including partial differential equations (PDEs), sparse Markov chains, and problems involving graph Laplacians, (e.g., Brezina et al. (2000); Henson & Vassilevski (2001); Heys et al. (2005); Stüben (2001); Horton & Leutenegger (1994); Virnik (2007); H. De Sterck et al. (2008; 2010); Treister & Yavneh (2010); Livne & Brandt (2013); Napov & Notay (2016); Fox & Manteuffel (2018)). While AMG is mathematically well grounded, its application involves the selection of problem-dependent parameters and heuristics, requiring expert knowledge and experience. Machine learning may therefore offer effective tools for developing *efficient* AMG algorithms.

AMG is a multi-level iterative method for linear systems,

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$ is a sparse matrix and $x, b \in \mathbb{R}^n$, with x the unknown solution vector. Given an initial approximate solution $x_0 \in \mathbb{R}^n$, the $(k+1)$st iteration of AMG proceeds as follows, with details provided in Section 3. Given the kth iteration, $x^{(k)}$, a few steps of a simple iterative solver (typically Gauss-Seidel relaxation) are applied, followed by the construction of a smaller linear system for the error at a "coarser scale". This is done by selecting a subset of $n_c < n$ "representative" variables (called the coarse variables), and constructing a *prolongation* operator $P \in \mathbb{R}^{n\times n_c}$ relating the coarse variables in $\mathbb{R}^{n_c}$ to the variables in $\mathbb{R}^n$. This smaller problem is treated recursively, by applying relaxation and appealing to a still coarser representation, and so on. Using P, the resulting solution is then "prolongated" back to the fine level to update the approximate solution, and a few additional relaxation sweeps are applied, yielding $x^{(k+1)}$. Note that the AMG procedure is analogous to the classical geometric multigrid algorithm (GMG) (Brandt, 1977; Briggs et al., 2000; Trottenberg et al., 2001), but unlike GMG (and other common multilevel algorithms) the variables need not lie on a regular grid or even be associated with a metric space.

The AMG procedure involves two critical heuristics which are applied at each level of the recursion, selection of coarse variables and construction of the prolongation matrix. Here we will use machine learning to address the latter heuristic. The choice of prolongation matrix P critically depends on A, and it strongly influences the efficiency of the AMG algorithm. After decades of research which yielded numerous theoretical insights and practical developments, there is still no single recipe for constructing prolongation operators that are optimal for a given class of problems. This paper proposes a framework for learning maps from entire classes of sparse symmetric positive definite and semidefinite (SPD/SPSD) matrices to prolongation opera-

[1]Weizmann Institute of Science, Rehovot, Israel. [2]NVIDIA Research [3]Technion, Israel Institute of Technology, Haifa, Israel. Correspondence to: Ilay Luz <ilayluz@gmail.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

tors, yielding efficient AMG solvers. Given a class of sparse SPD/SPSD matrices (e.g., low-degree graph Laplacian operators whose entries are drawn from a given distribution), we train a single network to solve any linear system of equations with a matrix drawn from that class. To train our network, we first represent the matrix A in (1) by a graph $G_A = (V_A, E_A)$, where the vertex set V_A contains a vertex per each variable x_i and the edge set E_A contains an edge e_{ij}, with a corresponding weight A_{ij}, if and only if $A_{ij} \neq 0$. Learning prolongation operators then becomes a graph learning problem which takes as input edge and node features and outputs edge weights on a subset of E_A. Specifically, we utilize a graph learning algorithm based on message passing (Gilmer et al., 2017; Battaglia et al., 2018).

This paper generalizes the work of Greenfeld et al. (2019), which is restricted to 2D diffusion partial differential equations discretized on a rectangular grid, and therefore cannot be applied to unstructured problems. In contrast, AMG handles general sparse SPD/SPSD matrices, with varying node degree in the graph G_A. Furthermore, development of new AMG approaches by experts is challenging, so the potential gain in using machine learning might be significant. Finally, in order to achieve efficient training, we introduce a novel Fourier analysis for locally unstructured problems, by constructing a block-periodic triangular mesh. Our experiments demonstrate the utility of our approach, showing in particular that our method can generalize across problem size, graph topology, and distribution, demonstrating better convergence rates than those achieved by classical AMG.

2. Related work

Amongst recent papers on using machine learning for linear system solvers, our work most closely follows Greenfeld et al. (2019), which uses a multilayer perceptron (MLP) with skip-connections to produce prolongation operators for 2D diffusion partial differential equations discretized on a rectangular grid. As noted above, here we lift all such structure restrictions. To the best of our knowledge, our work is the first to apply neural networks for solving such broad classes of sparse linear systems.

Another notable related paper is Hsieh et al. (2019), which uses a convolutional network to improve on an existing linear iterative solver. In particular, learning is applied to improve a GMG algorithm for structured Poisson problems in an end-to-end manner, by using a U-Net architecture with several downsampling and upsampling layers, and learning from supervised data. Schmitt et al. (2019) use evolutionary methods to optimize a GMG solver. Katrutsa et al. (2017) optimize restriction and prolongation operators for GMG, by formulating the entire two-grid algorithm as a deep neural network, and approximately minimizing the spectral radius of the resulting iteration matrix. They evaluate their method on single instances of various structured-grid differential equations in 1D. Sun et al. (2003) use a tailored network with a single hidden layer to solve the Poisson equation on a specific mesh.

Graph Neural Networks. Learning graph-structured data is an important and challenging learning setup that has received significant attention in recent years. The main challenge stems from the fact that graphs vary in size and topology, and also that graphs adhere to specific data symmetries (e.g., node reordering), which hinders the ability to use simple models such as MLPs. The first neural networks for graphs were proposed in Gori et al. (2005); Scarselli et al. (2009). Since then, a plethora of architectures were proposed, which can be roughly divided into two types: (1) spectral-based methods (e.g., Bruna et al. (2013); Henaff et al. (2015); Defferrard et al. (2016)), that define graph convolutions as diagonal operators in the Graph Laplacian eigenbasis, and (2) *message-passing neural networks* (Gilmer et al., 2017; Battaglia et al., 2018), which are currently the most popular and flexible architectures. In a nutshell, these models maintain a feature vector for each node in the graph, and update it by applying a parametric function (often an MLP) to the features of neighboring nodes.

Graph neural networks have been applied to various problems including molecule property prediction (Gilmer et al., 2017), social network analysis (Kipf & Welling, 2016) and point-cloud and shape analysis (Wang et al., 2019b). Recently, several papers (e.g., Selsam et al. (2018); Li et al. (2018)), targeted the task of solving combinatorial optimization problems efficiently using graph neural networks. Similarly to our work, their network is trained on small problems and is able to generalize to much larger problems, and to different distributions.

3. AMG background

AMG algorithms employ a hierarchy of progressively coarser approximations to the linear system under consideration, to accelerate the convergence of classical simple and cheap iterative processes called *relaxation* (most commonly Gauss-Seidel). For the SPD/SPSD problems we are considering, relaxation is known to be efficient for reducing so-called high-energy error modes, that is, error comprised primarily of eigenvectors of A with relatively large eigenvalues. On the other hand, relaxation is extremely inefficient for low-energy error comprised of eigenvectors with small corresponding eigenvalues (Falgout, 2006). The coarse-level correction, as described below, complements the relaxation by efficiently reducing low-energy modes, resulting in an efficient solver.

For a basic description of AMG, consider again the linear system $Ax = b$, where A is a real sparse SPD matrix of

size $n \times n$, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$ is the unknown solution vector. The two-level AMG algorithm is defined in Algorithm 1, with "relaxation sweeps" referring to iterations of the prescribed relaxation process, typically the classical Gauss-Seidel relaxation as defined below. For a detailed description of the two-level and multi-level AMG algorithm we refer the reader to classical textbooks (Briggs et al., 2000; Stüben, 2001).

Algorithm 1 Two-Level Algorithm

1: **Input:** SPD matrix $A \in \mathbb{R}^{n\times n}$, initial approximation $x^{(0)} \in \mathbb{R}^n$, right-hand side $b \in \mathbb{R}^n$, full-rank prolongation matrix $P \in \mathbb{R}^{n\times n_c}$, a relaxation scheme, $k = 0$, residual tolerance δ.
2: **repeat**
3: Perform s_1 relaxation sweeps starting with the current approximation $x^{(k)}$, obtaining $\tilde{x}^{(k)}$.
4: Compute the residual: $r^{(k)} = b - A\tilde{x}^{(k)}$.
5: Project the error equations to the coarser level and solve the coarse-level system: $A_c e_c^{(k)} = P^T r^{(k)}$, with $A_c = P^T AP$.
6: Prolongate and add the coarse-level solution: $\tilde{x}^{(k)} = \tilde{x}^{(k)} + Pe_c^{(k)}$.
7: Perform s_2 relaxation sweeps obtaining $x^{(k+1)}$.
8: $k = k + 1$.
9: **until** $||r^{(k-1)}|| < \delta$.

The prolongation P in Algorithm 1 is a sparse, full column-rank matrix, with $n_c < n$, and therefore A_c is a sparse SPD matrix of size $n_c \times n_c$, hence smaller than A. This allows us to apply the algorithm recursively. That is, in the multi-level (or multigrid) version of the algorithm, the exact solution in Step 5 is replaced by one or more recursive calls to the two-level algorithm, employing successively coarser levels (smaller matrices). An iteration with a single recursive call is known as a *V-cycle*, whereas an iteration with two calls is known as a *W-cycle* (motivated by the shape of the recursive call tree). These recursive calls are repeated until reaching a very small problem, which is solved cheaply by relaxation or an exact solve. Thus, the multi-level AMG algorithm applies iterations until convergence, with each iteration employing the recursive structure as described.

It can be seen that the two-level AMG algorithm is comprised of two main components: the relaxation sweeps performed in Line 3 and Line 7 of the algorithm, and the coarse-level correction process described in Lines 4-6. For relaxation, we adopt Gauss-Seidel iteration, which is induced by the splitting $A = L + U$, where L is the lower triangular part of A, including the diagonal, and U is the strictly upper triangular part of A. The resulting iterative scheme,

$$x^{(m)} = x^{(m-1)} + L^{-1}\left(b - Ax^{(m-1)}\right), \tag{2}$$

which defines the relaxation sweeps in Line 3 and Line 7, is convergent for SPD matrices[1]. Here, (m), the superscript, denotes the iteration number of the Gauss-Seidel relaxation. The error after iteration m, $e^{(m)} = x - x^{(m)}$, is related to the error before the iteration by the error propagation equation,

$$e^{(m)} = Se^{(m-1)}, \tag{3}$$

where $S = I - L^{-1}A$ is called the error propagation matrix of Gauss-Seidel relaxation, with I denoting the identity matrix of the same dimension as A.

The error propagation equation of the entire two-level algorithm is given by

$$e^{(k)} = Me^{(k-1)}, \tag{4}$$

where $M = M(A, P) = M(A, P; S, s_1, s_2)$ is the two-level error propagation matrix

$$M = S^{s_2} C S^{s_1}. \tag{5}$$

Here, s_1 and s_2 are the number of relaxation sweeps performed before and after the coarse-level correction process, and C is the error propagation matrix of the coarse-level correction, given by

$$C = I - P\left[P^T AP\right]^{-1} P^T A. \tag{6}$$

For a given operator A, the error propagation matrix M defined in (5) governs the convergence behavior of the two-level (and consequently multi-level) cycle. The key to designing effective AMG algorithms of this form lies in the selection of the prolongation matrix P. The relaxation and coarse-level correction process play complementary roles. That is, the solver may be efficient only if the error propagation matrix C of the coarse-level correction process significantly reduces low energy errors, because S only reduces efficiently high-energy errors, as noted above. Observe, on the other hand, that $CP = 0$, implying that the coarse-level correction eliminates any error that is in the subspace spanned by the columns of P. Indeed, the matrix $P\left[P^T AP\right]^{-1} P^T A$ in (6) is an A-orthogonal projection onto the range of P. It follows that we must construct P such that all low-energy errors will approximately be in its range. At the same time, P also needs to be very sparse for computational efficiency.

3.1. Constructing P

AMG algorithms typically divide the task of constructing P into three phases. The first step is a partitioning of the nodes

[1] The total number of arithmetic operations required for a single Gauss-Seidel relaxation sweep is roughly equal to the number of non-zero elements in A, assumed to be $O(n)$.

of the graph G_A into "C-nodes" and "F-nodes", where C and F stand for *coarse* and *fine*. The C-nodes comprise the "coarse grid", which is a subset of the "fine grid" comprised of all the nodes. The partitioning is performed on the basis of the nonzero off-diagonal elements of A (see, e.g., Briggs et al. (2000); Stüben (2001)) for detailed examples). The resulting n_c C-nodes correspond to the columns of P, while all the nodes of A correspond to the rows. The second step is selecting the sparsity pattern of P, also based on the elements of A. The final step is to select the values of the nonzero elements P. If row i of P corresponds to a C-point, say the one corresponding to column j, then $P_{i,j}$ is set to 1. The remaining nonzero values of P are selected by formulas or processes depending locally on the elements of A, that is, on the ith row of A and rows corresponding to nodes that are a short distance from node i on the graph of A.

In this paper we focus on the final step, the goal of selecting the nonzero values of P. To this end, we select the C-nodes and the sparsity pattern of P (first and second steps) according to the well-known classical AMG (CAMG) algorithm, as implemented in Olson & Schroder (2018). Then, we employ a learning process for deriving network-based formulas for the nonzero values of P based locally on the elements of the matrix A. We then compare the resulting solver to classical AMG, demonstrating improved convergence rates. This suggests that machine learning methods can provide an improvement over formulas that have been developed by experts over decades of research. The details of the learning process are provided in the next section.

4. Learning Method

Our task is to learn a mapping $P = P_\theta(A)$, where A is a sparse square matrix, θ are the learned parameters, and P is the resulting prolongation matrix. As discussed above, P should satisfy two objectives: it should be very sparse, and the resulting two-level algorithm should yield fast convergence. The first objective is satisfied by imposing a sparsity pattern on P derived from the classical AMG algorithm. For the second objective, the asymptotic convergence rate of the two-level algorithm is governed by the spectral radius of the error propagation matrix $M(A, P)$ (5), which we aim to approximately minimize.

Since backpropagation through Eigendecomposition tends to be numerically unstable (Wang et al., 2019a), we relax the objective to the squared Frobenius norm, which bounds the spectral radius from above. Hence, given a distribution $\mathcal{D}$ over linear operators, A, for some fixed relaxation S and parameters s_1 and s_2, we define the following unsupervised learning problem

$$\min_\theta \mathbf{E}_{A\sim\mathcal{D}} \left\|(M(A, P_\theta(A)))\right\|_F^2 \tag{7}$$

where the data are only the elements of A, which are drawn from some distribution $\mathcal{D}$.

4.1. Learning the prolongation operator

As explained in the introduction, the linear system A is represented as a graph $G_A = (V_A, E_A)$, with nodes corresponding to the variables, and edges corresponding to non-zero elements of A. Therefore, the problem of setting values to the prolongation matrix P amounts to assigning a set of values $\{p_e\}_{e\in E_A^c}$, where $E_A^c \subset E_A$ is defined according to the given sparsity pattern, i.e., a set of edges that connect C-nodes to F-nodes (and to themselves). See illustration of a small problem in Figure 1. Using this formalism, the task of selecting the prolongation weights can naturally be formulated as a graph learning problem: given the matrix A, a set of node features $\{f_v\}_{v\in V_A}$ and a set of edge features $\{f_e\}_{e\in E_A}$, we construct the graph G_A and use a graph neural network

$$P_\theta\left(G_A, \{f_v\}_{v\in V_A}, \{f_e\}_{e\in E_A}\right)$$

to predict the prolongation weights $\{p_e\}_{e\in E_A^c}$. In our case, the vertex features indicate whether the vertex is a C-point or not, and the edge features are comprised of the edge weights A_{ij} as well as the indicator of E_A^c that represents the sparsity pattern. As a final step, we scale each row of P to have the same row sum as the prolongation produced by the classical AMG algorithm[2]. The resulting prolongation operator is not guaranteed to be a full-rank matrix, but since singular matrices result in high loss, in our experiments the trained networks produced only full-rank matrices.

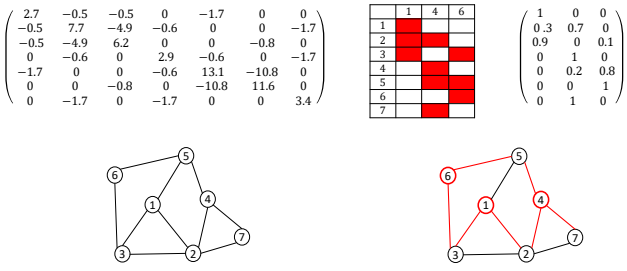


Figure 1. Example of sparse matrix A (upper left) and its associated graph G_A (lower left). The sparsity pattern (upper middle) defines the set of C-nodes $1, 4, 6$, and to which nodes each C-node contributes (denoted by the red cells). In the graph at the lower right panel, red edges represent the set E_A^c corresponding to the sparsity pattern. At the upper right is the prolongation matrix P, which is formed by setting weights to the sparsity pattern. Self-edges are omitted, for clarity.

[2]The important task of learning the optimal scaling is left to future research.

4.2. Network Architecture

Layers. Three main considerations come up when choosing a concrete GNN architecture that includes appropriate layers for our problem: (1) efficiency: the run-time of the mapping from A to P should be proportional to the number of nonzero elements, $O(n)$; (2) flexibility: the architecture should be able to process graphs of different size and connectivity; (3) edge-features: ability to process and output edge features. The first requirement rules out recently suggested layers as in Maron et al. (2019); Chen et al. (2019), which suffer from higher complexity, while the second requirement rules out spectral methods (e.g., Bruna et al. (2013); Henaff et al. (2015); Defferrard et al. (2016)). One type of layer that does fulfill all these requirements is the layer suggested in the Graph Network (GN) framework of Battaglia et al. (2018), which generalizes many message passing variants and extends them to allow using edge features. Each such layer is comprised of two steps: a vertex feature update step and an edge feature update step. Each of these steps is implemented by a parameterized update function (an MLP) and a summation operation for aggregating multiple neighboring features into a single feature vector.

Architecture. We use a variant of the encode-process-decode architecture suggested in Battaglia et al. (2018). This architecture is composed of three main parts: (1) an encoder followed by (2) a message-passing block and finally (3) a decoder. The encoder applies an MLP to the input features resulting in features of dimension 64. The message-passing block[3] is composed of three message passing layers, each of which receives as input the output of the previous layer, concatenated with the encoder features. This is intended to allow each message passing round to efficiently utilize the edge weights, the coarse nodes and sparsity pattern information. Finally, an MLP decoder independently maps each edge feature to a feature of size one that represents the prolongation weight. All MLPs have four layers of width 64, and apply ReLU activation.

For efficiency reasons, existing AMG algorithms derive the prolongation weights from local information. Similarly, we use a small number of message passing rounds, so the prediction on each edge is a function of edges only a few hops away. For a bounded-degree graph, the run-time of each message-passing round, for the entire graph, is proportional to the size of the graph, therefore we achieve the required $O(n)$ run-time. Moreover, the local nature of the computation allows the network to learn rules for constructing prolongation operators of arbitrary size, as is demonstrated in the experiments section.

[3]Because the message passing architecture we use applies only to directed graphs, we represent the symmetric matrix A as a directed graph with a pair of anti-parallel edges if two nodes are connected.

4.3. Efficient Training on Block-Circulant Matrices

Our network is able to generalize to problems considerably larger than the problems it saw during training, but moderately large problems are still required for training. The main computational bottleneck when training the network is the computation of the error propagation matrix M (5), which involves inversion of the coarse-level matrix P^TAP of size $n^c \times n^c$, where n^c is the number of nodes in the coarse-level graph. The cost of inverting a matrix may be as high as $O(n^3)$, because $n/n^c = O(1)$. The cost of other computations in training is $O(n)$ if implemented efficiently[4], therefore, for large problems the run-time of each training step is dominated by the inversion of P^TAP.

Generalizing to unstructured problems an approach used in Greenfeld et al. (2019), we reduce the training complexity by training on a limited class of A matrices called block-circulant matrices. A block-circulant matrix A of size $n \times n$, with $n = kb$, takes the form

$$A = \begin{pmatrix} A^{(0)} \\ A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(b-1)} \end{pmatrix},$$

where the blocks $A^{(m)}, m = 0, \ldots, b-1$ are $k \times n$ submatrices whose elements satisfy

$$A^{(m)}_{l,j} = A^{(m-1)}_{l,\ \bmod (j-k,n)},\ m = 1, \ldots, b-1, \quad (8)$$

and hence $A_{l,j} = A_{\bmod (l-k,n),\ \bmod (j-k,n)}$, where $\mathrm{mod}(x, y)$ is the remainder obtained when dividing integer x by integer y. In Greenfeld et al. (2019) and the associated supplementary material, it is proved that such matrices are unitarily block-diagonalized by an appropriate Fourier basis. Furthermore, because the graphs associated with the matrices we use for training (here as well as in Greenfeld et al. (2019)) are doubly block-periodic in the plane, each of the b blocks of size k by k in the matrix resulting from the block-diagonalization is itself block-circulant, comprised of b blocks of size c, with $k = bc$. The upshot is that the matrix A can be unitarily transformed into a similar matrix that is block-diagonal with b^2 blocks of size c. Thus, if we wish to compute the spectral radius or Frobenius norm of A, we can compute these values for the block-diagonal matrix, requiring us to process b^2 matrices of size c by c rather than a large matrix of size n by n, with $n = b^2c$. Since the block diagonalization itself is done analytically using the Fourier basis, it is cheap, and the overall cost of the entire computation is just linear in n (assuming c is a constant independent

[4]Since the automatic differentiation software we use does not have complete support for sparse matrix operations, these computations have cost $O(n^2)$. In practice, this has not been a bottleneck in our work.

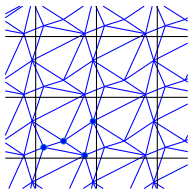


Figure 2. Example of a portion of a block-periodic Delaunay mesh with blocks of size $c = 4$

of n). To use this approach for training, we must make sure that M in (5) inherits the block-circulant form of A (with a smaller value of c due to the coarsening). We explain how we ensure this below.

To create a locally unstructured block-circulant matrix A, we select c random points on a square, and tile a large square domain with b by b such identical blocks. Now we apply Delaunay triangulation in the entire domain, and modify the edges near the boundaries of the domain so as to impose periodicity. Figure 2 depicts a small portion of such a graph. Next, we number the nodes consistently, such that the c nodes within each block are ordered contiguously and with the same ordering in all the blocks, while the blocks are ordered by the standard column-first ordering. Finally, we randomly select edge-weights for a single block according to the prescribed distribution, and replicate them to all the blocks. We thus obtain a graph whose Laplacian A of size b^2c by b^2c is block-circulant as explained above, and can be transformed into a similar matrix that is block-diagonal with b^2 blocks of size c by c.

To ensure that M inherits the block-circulant structure of A (with smaller c as mentioned above), we must impose that the prolongation P and relaxation S have the same block-circulant form as A. (For this statement to be formally well-defined, we must make P square by inserting a column of zeros per each F-node, but this has no influence on M.) Even though A is block-circulant, the matrix P (in its square form) is not a priori guaranteed to be block-circulant, but in practice, we found that for the standard algorithms it is very close to block-circulant. To make it exactly block-circulant, we choose the block with the most common sparsity pattern, and tile P with it. The S matrix corresponding to Gauss-Seidel relaxation on the block-circulant matrix A, is block-circulant (for the bounded-degree graphs we are considering) only in the limit of infinite n. Nevertheless, the approximation of treating it as such for finite graphs by the Fourier analysis, as is commonly done in standard multigrid Fourier analysis, does not unduly affect performance in our experiments.

Finally, we remark on another advantage of the block Fourier analysis. In strictly positive semidefinite problems, such as the graph Laplacian, the matrix P^TAP is singular, so M in (5) is undefined. The block-diagonalization allows us to isolate the single singular block and simply ignore it, and thus we do not need to artificially force A to be nonsingular by adding a positive diagonal term.

5. Experiments

We compare the performance of our network based solver[5] to the well-known classical AMG (CAMG) algorithm of Ruge & Stüben (1987), as implemented in Olson & Schroder (2018). We evaluate performance by measuring the number of iterations (V-cycles or W-cycles) required to reach a specified accuracy and by estimating the asymptotic convergence factor per iteration (often called cycle).

We focus on two different tasks: solving linear systems associated with graph Laplacian matrices with a variety of topologies, and solving diffusion partial differential equations discretized by linear finite elements over triangulated domains. Although the network is trained on a limited class of operators, namely block-circulant Laplacian matrices of relatively small size, where the coefficients are drawn from a lognormal distribution, it is able to generalize to larger problems, with diverse structure and distribution. This indicates that our network learns effective *rules* for constructing prolongation operators, not just solvers for specific problems, due to the local nature of the computation. In addition, we test our network based solver in the role of a preconditioner in spectral clustering applications.

Input and output representation. As discussed above, the input to the network is a graph $G_A = (V_A, E_A)$ with a set of node features $\{f_v\}_{v\in V_A}$ and a set of edge features $\{f_e\}_{e\in E_A}$. The output is a set of scalar prolongation weights $\{p_e\}_{e\in E_A^c}$, where E_A^c is defined by the given prolongation sparsity pattern. We represent node features by a one-hot encoding designating whether the node is a C-node

$$f_v = \begin{cases} [1,0] & \text{if } v \text{ is a C-node} \\ [0,1] & \text{if } v \text{ is not a C-node} \end{cases}.$$

We represent edge features by a concatenation of the nonzero element of A that corresponds to it, and a one-hot encoding designating whether the edge is part of the prolongation sparsity pattern

$$f_e = \begin{cases} [A_{ij}, 1, 0] & \text{if } e \in E_A^c \\ [A_{ij}, 0, 1] & \text{if } e \notin E_A^c \end{cases}.$$

Basis for comparison. The algorithm we use for comparison, and for setting the sparsity pattern and row sum of the prolongation operator, is the CAMG algorithm (Ruge &

[5]Code for reproducing experiments is available at https://github.com/ilayluz/learning-amg.

Stüben, 1987), implemented in PyAMG (Olson & Schroder, 2018). For the selection of the coarse nodes, we use the strategy of CLJP (Cleary et al., 1998; Alber & Olson, 2007), which selects a denser set of nodes than the default Ruge-Stuben algorithm (Ruge & Stüben, 1987). As is demonstrated in Table 1, the CLJP algorithm has better asymptotic convergence rates on graph Laplacian problems than other C-node selection algorithms implemented in PyAMG, including Ruge-Stuben, Smoothed Aggregation (Vanek et al., 1996), Root-node Aggregation (Olson et al., 2011), and PMIS (Sterck et al., 2006). We use Gauss-Seidel relaxation, with $s_1, s_2 = 1$. Because we use the same parameters in our method and the CAMG algorithm to which we compare, the run-time per iteration of the two algorithms is essentially the same. Of course, our setup time (which is applied once per test instance) is more expensive, because CAMG uses explicit formulas for computing the nonzero elements of P, whereas we use the trained network.

Training details. The training data are comprised of block-circulant graph Laplacian matrices, composed of 4×4 blocks with 64 points in each block, yielding 1024 variables. The construction of such matrices follows the description in Sec. 4.3, where the weights on the edges are drawn from standard lognormal distribution. The network is trained to minimize the Frobenius norm of the two-level error propagation matrix M in (5). In similar spirit as Greenfeld et al. (2019), the training is performed in two stages, first on the original problems and then on a training set comprised of the original problems and the once-coarsened problems as elaborated below.

At the first stage we train on 256000 problems with 4×4 blocks of size 64, with a single epoch. Then, we generate 128000 problems of 4×4 blocks of size 128 and we apply the trained network to generate prolongation operators for each of those problems, and compute the block-circulant coarse matrices $A_c = P^T AP$. The CLJP C-node selection algorithm (Cleary et al., 1998; Alber & Olson, 2007) selects roughly half of the nodes, so the coarsened problems are of approximately the same size as the original problems. We then generate 128000 additional problems with 4×4 blocks of size 64, shuffle them with the coarsened problems, and continue training the network on the combined set of 256000 problems for another epoch[6].

All experiments were conducted using the TensorFlow framework (Abadi et al., 2016) using NVIDIA V100 GPU. We use a batch size of 32 and employ the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 3×10^{-3}. Training took roughly 12 hours for first phase, another 12 hours for second phase.

[6]We may continue this process by training on twice-coarsened problems and so on. In practice however, we found that a network trained on a mixture of the original problem and the once-coarsened problem achieves good results even for large problems with multiple coarsening levels

5.1. Evaluation

Graph Laplacians. We first evaluate the performance of our network based solver on random graph Laplacian problems. To this end, we sample points uniformly on the unit square, and compute a Delaunay triangulation. Each edge is then given by a random weight sampled from a standard lognormal distribution, and the corresponding graph Laplacian matrix is constructed. We perform experiments on a range of problem sizes, with both V-cycles and W-cycles. We measure the *asymptotic convergence factor* per cycle by initializing with a random $x^{(0)}$, performing 80 AMG cycles on the homogeneous problem[7] $Ax = 0$, and computing the ratio of the residual norms of the last two iterations, $\frac{||r^{(k+1)}||_2}{||r^{(k)}||_2}$. For W-cycles, this value is almost equal to the spectral radius of the error iteration matrix M. Figure 4a shows the asymptotic convergence factor on problem sizes ranging from 1024 to 400000, for CAMG and for our model. Table 2 shows the success rate of the network, defined as the percentage of problems where our model outperformed CAMG. Figure 4b shows the asymptotic convergence factor for graph Laplacian problems where the edge weights are sampled from a uniform $U(0, 1)$ distribution, rather than the lognormal distribution used in training. The results indicate that the network based solver performs better than CAMG, and generalizes to large problems and other distributions, structure and topology.

Table 1. Asymptotic convergence factors for graph Laplacian problem with lognormal distributions of size 65536, for heuristic CAMG solvers. Tested on W-cycle, averaged over 100 runs for each C-node selection algorithm

C-node algorithm	average convergence factor
CLJP	0.21
Ruge-Stuben	0.24
Smoothed Aggregation	0.68
Root-node Aggregation	0.70
PMIS	0.98

Diffusion equations. We test the network based solver on a variety of diffusion partial differential equations,

$$-\nabla \cdot (\mathbf{g}\nabla \mathbf{u}) = \mathbf{f}, \tag{9}$$

discretized on 2D triangular meshes. Given a 2D triangular mesh, for each triangle we randomly select a positive

[7]The asymptotic convergence factor is independent of the right-hand side b, so long as b is in the range of A, i.e., has zero mean. We use $b = 0$ so that we can perform many iterations without encountering roundoff errors (so long as we subtract off the mean so that the exact solution is zero), allowing us to measure accurately the asymptotic factor.

Table 2. Success rate measured for graph Laplacian problems with lognormal (columns 2,3) and uniform (columns 4,5) distributions. Tested on V- and W-cycles, averaged over 100 runs for each problem size

size	V-cycle	W-cycle	V-cycle	W-cycle
1024	97%	83%	83%	83%
2048	98%	91%	84%	85%
4096	98%	91%	84%	84%
8192	99%	84%	91%	84%
16384	99%	79%	92%	80%
32768	98%	78%	89%	81%
65536	100%	79%	88%	80%
131072	100%	76%	91%	82%
262144	100%	83%	94%	72%
400000	98%	82%	93%	78%

diffusion coefficient and construct the corresponding linear system, using linear finite elements (FEM). The mesh is generated using the Triangle mesh generation software of Shewchuk (1996). The diffusion coefficients $\mathbf{g}_i$ are sampled from a lognormal distribution with a log-mean of zero and log-standard deviation of 0.5. Finally, we modify the operator at the boundaries to impose Dirichlet boundary conditions. The resulting matrix A is SPD.

We test the same trained network as in the graph Laplacian problem (without any additional training) on a circular domain with a square hole and variable triangle density (see Figure 3).

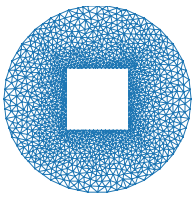


Figure 3. Example of a FEM mesh

Figure 4c shows the asymptotic convergence factor for problem sizes ranging from 1024 to 400000, for CAMG and for the network based solver, averaged over 100 runs. Table 3 shows the success rate of the network, defined as the percentage of problems where our model outperforms CAMG.

Spectral Clustering. Spectral clustering is a widely used clustering algorithm (Von Luxburg, 2007). It involves computing eigenvectors associated with the smallest nonzero eigenvalues of a Laplacian matrix A derived from a pairwise similarity measure of the data, and then performing a standard clustering algorithm (e.g., k-means) on them. In the case of large-scale sparse problems, these eigenvalues can be efficiently computed by an iterative preconditioned conjugate gradient method, such as LOBPCG (Knyazev, 2001) used in the popular Scikit-learn library (Pedregosa et al., 2011). At each iteration i, a matrix-vector product

Table 3. Success rate measured for FEM diffusion equations. Tested on V and W-cycle, averaged over 100 runs for each problem size

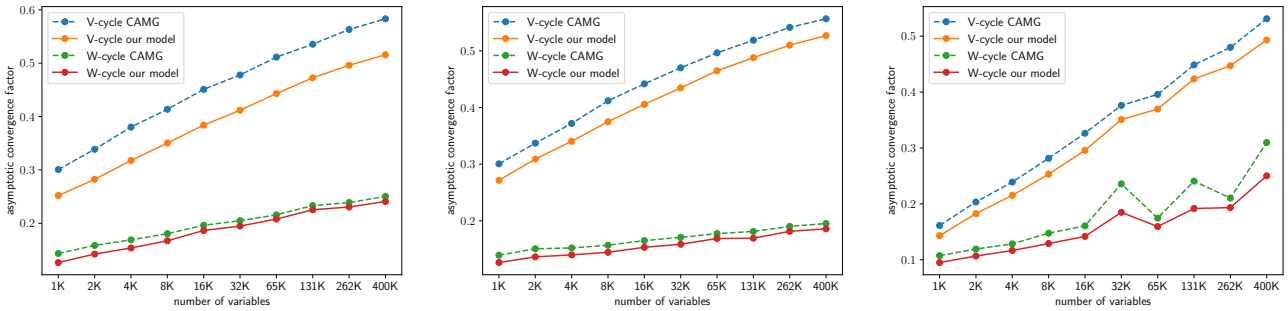
size	V-cycle	W-cycle
1024	87%	88%
2048	94%	85%
4096	99%	84%
8192	99%	90%
16384	96%	88%
32768	96%	96%
65536	98%	87%
131072	96%	94%
262144	97%	77%
400000	96%	89%

of the pseudo-inverse of A and a residual vector r_i, i.e., $A^\dagger r_i$, is approximately computed by applying CAMG as a pre-conditioner to estimate the solution of the linear system $Ax = r_i$.

We evaluate the efficiency of our network based solver as a preconditioner by estimating the number of iterations needed to converge to a certain accuracy, and comparing with the CAMG preconditioner. To this end, we train our network with the same hyper-parameters and generate training data as follows. Each training problem is produced by 1024 points sampled from two dimensional isotropic Gaussian distributions, one with standard deviation 1.0, the other with standard deviation 2.5, and the two centers are uniformly sampled from $[-10, 10]^2$ (see Figure 5a, for example). We compute the Euclidean k-nearest neighbors for $k = 10$, and convert the distances to affinity measures by setting $S_{ij} = e^{-d_{ij}^2}$, where d_{ij} is the distance between two different points i and j ($S_{ii} = 0$). We then compute the symmetric normalized Laplacian matrix $A = I - D^{-\frac{1}{2}} S D^{-\frac{1}{2}}$, where D is a diagonal matrix, $D_{ii} = \sum_{j=1}^{n} S_{ij}$.

We train the network in a more limited manner, in a single phase without Fourier analysis, on 256000 problems. To avoid inverting singular matrices when training, we modify the Laplacian matrices to be non-singular by adding random positive values to the diagonal of the matrix, from distribution $U(0, 0.2)$. Evaluation is done on the original singular matrices. To evaluate, we measure the number of LOBPCG iterations required to reach residual tolerance of 10^{-12} on a variety of problems, where the linear solver is a single W cycle. Table 4 shows results on several distributions. Evidently, the network is able to generalize to different number of points, number of clusters, dimensions, and distributions.

Ablation Study. We run a number of experiments to determine how the performance of the network is influenced by our design decisions. We evaluate performance on graph Laplacian problems, with networks trained with the following modifications: less message-passing layers, lower depth of MLPs, no concatenation of encoder features as input to



(a) Graph Laplacian problems with lognormal distribution.

(b) Graph Laplacian problems with uniform distribution.

(c) FEM diffusion equations.

Figure 4. Asymptotic convergence factors (smaller is better) for various problems. Each problem is tested on V and W-cycle, and averaged over 100 runs for each problem size

Table 4. Comparison of number of LOBPCG iterations required to reach specified tolerance in spectral clustering problems, averaged over 100 runs for each distribution

distribution	size	CAMG	ours	ratio
two Gaussians	10^3	15.67	13.44	85.8%
two Gaussians	10^4	20.95	18.82	89.8%
two Gaussian 5-NN	10^3	22.53	23.45	104.1%
five Gaussians	10^3	19.99	17.41	87.1%
two Gaussians 3D	10^3	12.58	11.26	89.5%
two moons	10^3	23.44	21.47	91.6%
two moons	10^4	37.17	35.02	94.2%
two concentric circles	10^3	19.48	16.84	86.5%

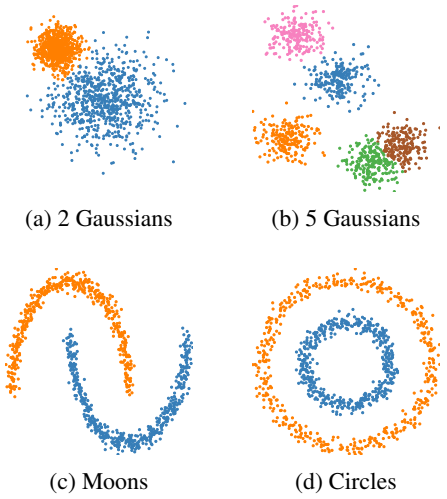
Table 5. Success rate measured for graph Laplacian problems with lognormal distribution of size 65536. Tested on W-cycle, averaged over 100 runs for each architecture

architecture	success rate
Suggested architecture	79%
Depth 2 MLP	74%
2 message-passing layers	63%
No encoder concatenation	75%
No indicator features	68%

message-passing layers, and no one-hot indicators on edge and node input features. Table 5 shows the success rate of these networks on problems with lognormal distribution of size 65536. As can be seen, performance moderately drops when lowering the depth of the MLPs or removing encoder concatenation, and significantly drops when lowering the number of message-passing layers, and removing indicator features.

Conclusion

In this paper we propose a framework for learning Algebraic Multigrid (AMG) prolongation operators for linear systems which are defined directly on graphs, rather than



(a) 2 Gaussians

(b) 5 Gaussians

(c) Moons

(d) Circles

Figure 5. Examples of our results for two dimensional spectral clustering problems

on structured grids. We treat linear systems that can be expressed by sparse symmetric positive (semi-) definite matrices. We formulate the problem as a learning task and train a single graph neural network, with an efficient message-passing architecture, to learn a mapping from an entire class of such matrices to prolongation operators. We employ an efficient and unsupervised training on a limited class of block-circulant matrices. Our experiments indicate success, i.e, improved convergence rates compared to classical AMG, on a variety of problems. This includes graph Laplacian problems over a triangulated mesh, where the edge weights are drawn randomly from some distribution, diffusion partial differential equations discretized on 2D triangular meshes and spectral clustering problems. An interesting and important direction for future research is learning to select the coarse representatives as well as the sparsity pattern of the prolongation matrix.

Acknowledgment

This research was supported by the Israel Science Foundation, grant No. 1639/19.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Alber, D. M. and Olson, L. N. Parallel coarse-grid selection. *Numerical Linear Algebra with Applications*, 14(8):611–643, 2007.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Brandt, A. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31:333–390, 1977.

Brandt, A., McCormick, S. F., and Ruge, J. Algebraic multigrid (AMG) for sparse matrix equations. In Evans, D. J. (ed.), *Sparsity and its applications*, pp. 257–284. Cambridge University Press, Cambridge, 1984.

Brezina, M., Cleary, A. J., Falgout, R. D., Henson, V. E., Jones, J. E., Manteuffel, T. A., McCormick, S. F., and Ruge, J. W. Algebraic multigrid based on element interpolation AMGe. *SIAM J. Sci. Comput.*, 22(5):1570–1592, 2000. ISSN 1064-8275.

Briggs, W. L., Henson, V. E., and McCormick, S. F. *A multigrid tutorial*. SIAM, second edition, 2000.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral Networks and Locally Connected Networks on Graphs. pp. 1–14, 2013. URL http://arxiv.org/abs/1312.6203.

Chen, Z., Villar, S., Chen, L., and Bruna, J. On the equivalence between graph isomorphism testing and function approximation with gnns, 2019.

Cleary, A. J., Falgout, R. D., Jones, J. E., et al. Coarse-grid selection for parallel algebraic multigrid. In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pp. 104–115. Springer, 1998.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pp. 3844–3852, 2016.

Falgout, R. D. An introduction to algebraic multigrid. *IEEE: Computing in Science and Engineering*, 8:24–33, 2006.

Fox, A. and Manteuffel, T. Algebraic multigrid for directed graph laplacian linear systems (ns-lamg). *Numerical Linear Algebra with Applications*, 25(3):e2152, 2018. doi: 10.1002/nla.2152.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR. org, 2017.

Gori, M., Monfardini, G., and Scarselli, F. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pp. 729–734. IEEE, 2005.

Greenfeld, D., Galun, M., Basri, R., Yavneh, I., and Kimmel, R. Learning to optimize multigrid pde solvers. *arXiv preprint arXiv:1902.10248*, 2019.

H. De Sterck, Manteuffel, T. A., McCormick, S. F., Nguyen, Q., and Ruge, J. Multilevel adaptive aggregation for Markov chains, with application to web ranking. *SIAM J. Sci. Comput*, 30:2235–2262, 2008.

H. De Sterck, Manteuffel, T. A., McCormick, S. F., Miller, K., Pearson, J., Ruge, J., and Sanders, G. Smoothed aggregation multigrid for Markov chains. *SIAM J. Sci. Comput.*, 32:40–61, 2010.

Henaff, M., Bruna, J., and LeCun, Y. Deep Convolutional Networks on Graph-Structured Data. (June), 2015. ISSN 1506.05163. URL http://arxiv.org/abs/1506.05163.

Henson, V. E. and Vassilevski, P. S. Element-free AMGe: General algorithms for computing interpolation weights in AMG. *SIAM J. Sci. Comput.*, 23(2):629–650, 2001. ISSN 1064-8275.

Heys, J. J., Manteuffel, T. A., McCormick, S. F., and Olson, L. N. Algebraic multigrid for higher-order finite elements. *J. Comput. Phys.*, 204(2):520–532, 2005. ISSN 0021-9991.

Horton, G. and Leutenegger, S. T. A multi-level solution algorithm for steady-state Markov chains. *Perform. Eval. Rev.*, 22:191–200, 1994.

Hsieh, J.-T., Zhao, S., Eismann, S., Mirabella, L., and Ermon, S. Learning neural pde solvers with convergence guarantees. *arXiv preprint arXiv:1906.01200*, 2019.

Katrutsa, A., Daulbaev, T., and Oseledets, I. Deep multigrid: learning prolongation and restriction matrices. *arXiv preprint arXiv:1711.03825*, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Knyazev, A. V. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

Li, Z., Chen, Q., and Koltun, V. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pp. 539–548, 2018.

Livne, O. E. and Brandt, A. Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver. *SIAM. J. Stat. Sci. Comput.*, 34(12):B499–B522, 2013.

Maron, H., Ben-Hamu, H., Serviansky, H., and Lipman, Y. Provably powerful graph networks, 2019.

Napov, A. and Notay, Y. An efficient multigrid method for graph Laplacian systems. *Electronic Trans. Numer. Anal.*, 45:201–218, 2016.

Olson, L. N. and Schroder, J. B. PyAMG: Algebraic multigrid solvers in Python v4.0, 2018. URL https://github.com/pyamg/pyamg. Release 4.0.

Olson, L. N., Schroder, J. B., and Tuminaro, R. S. A general interpolation strategy for algebraic multigrid using energy minimization. *SIAM Journal on Scientific Computing*, 33 (2):966–991, 2011.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Ruge, J. Algebraic multigrid (AMG) for geodetic survey problems. In *in Proceedings of the International Multigrid Conference*. Copper Mountain, CO, 1983.

Ruge, J. and Stüben, K. Algebraic multigrid (AMG). In McCormick, S. F. (ed.), *Multigrid Methods, frontiers in applied mathematics*, pp. 73–130. SIAM, Philadelphia, 1987.

Ruge, J. W. and Stüben, K. Algebraic multigrid. In *Multigrid methods*, pp. 73–130. SIAM, 1987.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *Neural Networks, IEEE Transactions on*, 20(1):61–80, 2009. ISSN 1045-9227. doi: 10.1109/TNN.2008.2005605.

Schmitt, J., Kuckuk, S., and Köstler, H. Optimizing geometric multigrid methods with evolutionary computation. *arXiv preprint arXiv:1910.02749*, 2019.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

Shewchuk, J. R. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Workshop on Applied Computational Geometry*, pp. 203–222. Springer, 1996.

Sterck, H. D., Yang, U. M., and Heys, J. J. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM J. Matrix Anal. Appl*, 27:1019–1039, 2006.

Stüben, K. Algebraic multigrid (AMG): an introduction with applications. In Trottenberg, U., Oosterlee, C., and Schüller, A. (eds.), *Multigrid*. Academic Press, 2001.

Sun, M., Yan, X., and Sclabassi, R. J. Solving partial differential equations in real-time using artificial neural network signal processing as an alternative to finite-element analysis. In *Neural Networks and Signal Processing, 2003. Proceedings of the 2003 International Conference on*, volume 1, pp. 381–384. IEEE, 2003.

Treister, E. and Yavneh, I. Square and stretch multigrid for stochastic matrix eigenproblems. *Numerical Linear Algebra with Application*, 17:229–251, 2010.

Trottenberg, U., Oosterlee, C., and Schüller, A. *Multigrid*. Academic Press, London and San Diego, 2001.

Vanek, P., Mandel, J., and Brezina, M. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56:179–196, 1996.

Virnik, E. An algebraic multigrid preconditioner for a class of singular M-matrices. *SIAM J. Sci. Comput.*, 29(5): 1982–1991, 2007.

Von Luxburg, U. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

Wang, W., Dang, Z., Hu, Y., Fua, P., and Salzmann, M. Backpropagation-friendly eigendecomposition. *arXiv preprint arXiv:1906.09023*, 2019a.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., and Solomon, J. M. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 38 (5):1–12, 2019b.