Support-Theoretic Subgraph Preconditioners for Large-Scale SLAM

Yong-Dian Jian, Doru Balcan, Ioannis Panageas, Prasad Tetali and Frank Dellaert

***Abstract*— Efficiently solving large-scale sparse linear systems is important for robot mapping and navigation. Recently, the subgraph-preconditioned conjugate gradient method has been proposed to combine the advantages of two reigning paradigms, direct and iterative methods, to improve the efficiency of the solver. Yet the question of how to pick a good subgraph is still open. In this paper, we propose a new metric to measure the quality of a spanning tree preconditioner based on support theory. We use this metric to develop an algorithm to find good subgraph preconditioners and apply them to solve the SLAM problem. The results show that although the proposed algorithm is not fast enough, the new metric is effective and resulting subgraph preconditioners significantly improve the efficiency of the state-of-the-art solver.**

I. Introduction

Simultaneous localization and mapping (SLAM) refers to the problem of localizing a robot in an unknown environment while simultaneously building a consistent map. Being able to efficiently conduct SLAM in large and complex environments is important for autonomous mobile robots [1], [2].

The *smoothing* approach had been successfully applied to solve the SLAM problem [3]. Central to the efficiency of the smoothing approach is the ability to solve sparse linear systems efficiently. There are two ways to solve linear systems: direct methods and iterative methods. Direct methods [4] are efficient if a good elimination ordering is available, but they may not scale well and lead to high computational cost. Iterative methods [5] have better scalability but they suffer from slow convergence if the problem is ill-conditioned.

Recently, Dellaert et al. [6] proposed the subgraph-preconditioned conjugate gradients (SPCG) method, which aims to combine the advantages of direct and iterative methods to efficiently solve the SLAM problem. The main idea is to identify a sparse sub-problem (subgraph) that can be efficiently factorized by direct methods, and use it to build a preconditioner for the conjugate gradient (CG) method. They showed that SPCG is superior to using either direct or iterative methods alone. Yet the question of how to pick a good subgraph is still open.

In this paper, we propose a new metric to measure the quality of a spanning tree preconditioner based on the recently-developed support theory [7]. Then we use this metric to develop an algorithm based on Markov Chain Monte Carlo (MCMC) methods [8] to find a good spanning tree, and then augment it with additional edges to build a good subgraph preconditioner for SLAM. We use the resulting subgraph preconditioner with the least-squares preconditioned conjugate gradient method to solve synthetic and real SLAM problems. The results show that although the proposed algorithm is not fast enough, our new metric is effective and the proposed algorithm is able to produce significantly better subgraph preconditioners.

The authors are affiliated with the College of Computing, Georgia Institute of Technology, USA. Prasad Tetali is also affiliated with the School of Mathematics, Georgia Institute of Technology, USA.

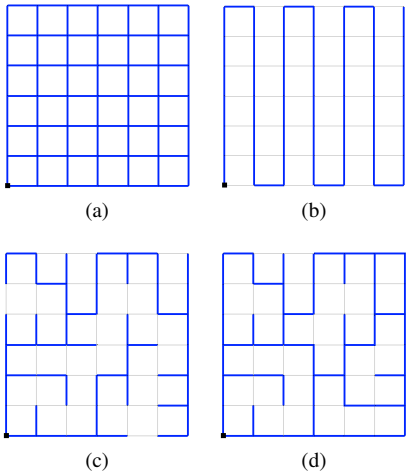


Fig. 1: Illustration of the proposed algorithm with a simple grid graph. (a) The original graph. (b) The robot's trajectory as an initial spanning tree. (c) The spanning tree after 30 iterations of our algorithm. (d) A subgraph is built by inserting additional high-stretch edges to the spanning tree.

This paper has three contributions: (1) We present a new metric based on support theory to measure the quality of a spanning tree preconditioner for SLAM. (2) On the basis of this new metric, we propose an algorithm to construct good subgraph preconditioners. (3) Finally we apply these subgraph preconditioners to improve the efficiency of the state-of-the-art SLAM solver. This paper focuses on the theoretical contributions above, and the MCMC-based algorithm we use to find the high-quality preconditioners is not practical in its current form. However, recent developments in finding low-stretch spanning trees [9], ultra-sparsifiers [10], and simple combinatorial solvers [11] make us believe that this obstacle will be removed in future work.

II. Review

A. SLAM

Here we review SLAM formulation to facilitate the exposition. We define $\boldsymbol{\theta} = \{\theta_i\}_{i=1}^{n}$ as the state variables (e.g., robot poses), and $\mathbf{Z} = \{z_j\}_{j=1}^{m}$ as the measurements (e.g., odometry and loop-closure). The goal is to obtain the

maximum a posteriori (MAP) estimation

$$\boldsymbol{\theta}_{\text{MAP}}(\mathbf{Z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, P(\boldsymbol{\theta})P(\mathbf{Z} \mid \boldsymbol{\theta}). \tag{1}$$

Assuming the variables are independent, and the measurements are conditionally independent, we can factorize the right-hand side of (1) into

$$P(\boldsymbol{\theta})P(\mathbf{Z}|\boldsymbol{\theta}) \propto \prod_{i=1}^{n} P(\theta_i) \prod_{j=1}^{m} P(z_j \mid \theta_j) \tag{2}$$

where θ_j denotes the set of variables associated with the jth measurement.

The SLAM problem can also be formulated with the factor graph representation [12] where each vertex denotes a state variable, and each factor (edge) is represented by the squared error term associated with a probability density function in (2). More specifically, we assume prior and measurement models are Gaussian, defined by

$$P(\theta_i) \propto \exp(-\|g_i(\theta_i)\|^2_{\Gamma_i}) \tag{3}$$

$$P(z_j \mid \theta_j) \propto \exp(-\|h_j(\theta_j)\|^2_{\Psi_j}). \tag{4}$$

where $g_i(\cdot)$ denotes the prior model over the ith variable and $h_j(\cdot)$ denotes the model of the jth measurement. In both models, we assume zero-mean and normally distributed noise with covariance matrices Γ_i and Ψ_j respectively. Here $\|e\|_\Sigma = \sqrt{e^T\Sigma^{-1}e}$ denotes the Mahalanobis distance. By substituting the probability densities in (2) with the functions in (3) and (4), and taking negative logarithm, we obtain the following factor graph representation for the SLAM problem

$$\boldsymbol{\theta}_{\text{MAP}}(\mathbf{Z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{n} \|g_i(\theta_i)\|^2_{\Gamma_i} + \sum_{j=1}^{m} \|h_j(\theta_j)\|^2_{\Psi_j}. \tag{5}$$

More generally, we can combine the two terms in (5) as

$$\boldsymbol{\theta}_{\text{MAP}}(\mathbf{Z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{k=1}^{m+n} \|e_k(\theta_k)\|^2_{\Sigma_k} \tag{6}$$

where $e_k(\cdot)$ denotes a function on a set of variables θ_k with a covariance matrix Σ_k.

B. A Smoothing Approach to SLAM

Here we show how to solve (6) via smoothing. In general, the function in (6) is not convex and has no closed-form expression to compute the optimum, but assuming we have some initial estimates of the variables, we can find a local minimum by using any nonlinear least-squares optimization algorithm (e.g., the Gauss-Newton or the Levenberg-Marquardt algorithm) [13]. The key is to apply the first-order Taylor expansion to linearize the function as

$$e_k(\theta_k) \approx e_k(\theta_k^0) + \mathbf{J}_k \Delta\theta_k \tag{7}$$

where $\mathbf{J}_k$ is the Jacobian matrix of $e_k(\cdot)$ with respect to θ_k at the linearization point θ_k^0:

$$\mathbf{J}_k = \left.\frac{\partial e_k(\theta_k)}{\partial \theta_k}\right|_{\theta_k^0}. \tag{8}$$

If we set (7) to zero, then we obtain $\mathbf{J}_k\Delta\theta_k = -e_k(\theta_k^0)$ which is linear in $\Delta\theta_k$. Repeating this procedure for all of the $e_k(\cdot)$ functions, we can derive a linear system

$$\mathbf{A}\Delta\boldsymbol{\theta} = \mathbf{b} \tag{9}$$

where $\mathbf{A}$ is a rectangular matrix whose kth (block) row contains the Jacobian matrix $\mathbf{J}_k$ in (8), and $\mathbf{b}$ is a vector whose kth (block) row equals $-e_k(\theta_k^0)$. Equation (9) can be considered as a linearized version of the SLAM problem whose graph structure is represented by the sparsity pattern of $\mathbf{A}$. Hereafter we will refer to (9) as the *linear system* or the *Gaussian factor graph* of the SLAM problem, and refer to $\mathbf{A}$ as the *Jacobian matrix*. We then iteratively solve (9) to update the current estimates until convergence.

We can see that solving SLAM is equivalent to solving a sequence of sparse linear systems. Direct and iterative methods are the two reigning paradigms to solve sparse linear systems [4], [5], but each of these methods presents its own advantages and limitations when applied to solving large SLAM problems [6].

C. Subgraph-Preconditioned Conjugate Gradient Method

Recently, Dellaert et al. [6] proposed the subgraph-preconditioned conjugate gradient (SPCG) method to solve the linearized SLAM problem efficiently. The main idea is to combine the advantages of direct and iterative methods by identifying a *sparse* sub-problem (subgraph) and then solve it with direct methods to build a prior probability density to precondition the original problem. Choosing a sparse subgraph has the advantage that solving the induced sub-problem and applying the preconditioner can both be performed efficiently.

More specifically, for any linear least-squares problem or Gaussian factor graph $f(\mathbf{x}) = \|\mathbf{A}_G\mathbf{x} - \mathbf{b}_G\|^2$, SPCG first identifies a sparse sub-problem (subgraph) $\|\mathbf{A}_S\mathbf{x} - \mathbf{b}_S\|^2$ that can be efficiently solved by direct methods. Note that subscript G denotes the original graph while the subscript S denotes a subgraph of G. The tuple $(\mathbf{A}_S, \mathbf{b}_S)$ corresponds to a subset of rows in $(\mathbf{A}_G, \mathbf{b}_G)$. Hence we can split the function into two terms: $f(\mathbf{x}) = \|\mathbf{A}_S\mathbf{x} - \mathbf{b}_S\|^2 + \|\mathbf{A}_{\bar{S}}x - \mathbf{b}_{\bar{S}}\|^2$ where the subscript $\bar{S} = G\backslash S$ denotes the complement of S. Hereafter we will refer to $(\mathbf{A}_S, \mathbf{b}_S)$ as the *subgraph* part and $(\mathbf{A}_{\bar{S}}, \mathbf{b}_{\bar{S}})$ as the *constraint* part of the problem.

The easiest way to understand preconditioning is through variable re-parametrization. Consider applying QR factorization to $\mathbf{A}_S = \mathbf{Q}_S\mathbf{R}_S$ to obtain the solution $\bar{\mathbf{x}} = \mathbf{R}_S^{-1}\mathbf{Q}_S^T\mathbf{b}_S$ of the subgraph part with the corresponding Gaussian log-likelihood $\|\mathbf{R}_S\mathbf{x} - \mathbf{c}_S\|^2$ where $\mathbf{c}_S = \mathbf{Q}_S^T\mathbf{b}_S$. Therefore, the original objective function becomes

$$f(\mathbf{x}) = \|\mathbf{R}_S\mathbf{x} - \mathbf{c}_S\|^2 + \|\mathbf{A}_{\bar{S}}\mathbf{x} - \mathbf{b}_{\bar{S}}\|^2 \tag{10}$$

Now we re-parametrize the problem in terms of the whitened deviation from the prior $\mathbf{y} = \mathbf{R}_S\mathbf{x} - \mathbf{c}_S = \mathbf{R}_S(\mathbf{x} - \bar{\mathbf{x}})$. By substituting $\mathbf{x} = \bar{\mathbf{x}} + \mathbf{R}_S^{-1}\mathbf{y}$ in (10), we obtain

$$\bar{f}(\mathbf{y}) = \|\mathbf{y}\|^2 + \|\mathbf{A}_{\bar{S}}\mathbf{R}_S^{-1}\mathbf{y} - \mathbf{d}\|^2 \tag{11}$$

where $\mathbf{d} = \mathbf{b}_{\bar{S}} - \mathbf{A}_{\bar{S}}\bar{\mathbf{x}}$. Then we can solve (11) using the least-squares conjugate gradient (LSCG) method. [14].

III. The Quality of Subgraph Preconditioners

Although SPCG demonstrates promising performance [6], the question of how to pick a good subgraph is still open. To this end, we introduce support theory [7] to measure the quality of subgraph preconditioners.

A. Generalized Condition Number

The generalized condition number is a well-known measure for the convergence speed of the preconditioned conjugate gradient (PCG) method [5]. Namely, the generalized condition number for a pair of positive and definite matrices $\mathbf{M}_G$ and $\mathbf{M}_H$ is defined as

$$\kappa(\mathbf{M}_G, \mathbf{M}_H) = \frac{\lambda_{\max}(\mathbf{M}_G, \mathbf{M}_H)}{\lambda_{\min}(\mathbf{M}_G, \mathbf{M}_H)} \tag{12}$$

where $\lambda_{\max}(\mathbf{M}_G, \mathbf{M}_H)$ and $\lambda_{\min}(\mathbf{M}_G, \mathbf{M}_H)$ denote the largest and smallest generalized eigenvalue respectively. The generalized condition number is known to be inversely proportional to the worst-case convergence speed of PCG [5]. In the SLAM context, the roles of $\mathbf{M}_G$ and $\mathbf{M}_H$ are played by the outer product of the Jacobian matrix, i.e. $\mathbf{M}_G = \mathbf{A}_G^T\mathbf{A}_G$ and $\mathbf{M}_H = \mathbf{A}_H^T\mathbf{A}_H$. Hereafter we will refer to $\mathbf{M}_G$ as the original system matrix and $\mathbf{M}_H$ as the preconditioner system matrix.

B. Support Theory

The generalized condition number measures the quality of a preconditioner in terms of the ratio of extreme eigenvalues; however, directly optimizing this measure is not trivial. Recently, support theory [7] has been proposed to assess the quality of preconditioners for symmetric and positive definite linear systems. Here we provide a brief introduction to support theory. The readers may refer to [7] for details.

Central to support theory is the notion of *support number*:

Definition 1. *The support number of a pair of square matrices $\mathbf{M}_G \in \mathbb{R}^{n\times n}$, and $\mathbf{M}_H \in \mathbb{R}^{n\times n}$ is defined as*

$$\sigma(\mathbf{M}_G, \mathbf{M}_H) = \min\{t \in \mathbb{R} \mid \tau\mathbf{M}_H \succeq \mathbf{M}_G, \forall \tau \geq t\}. \tag{13}$$

In other words, the support number is the smallest number of "copies" that we need for $\mathbf{M}_H$ in order to dominate $\mathbf{M}_G$ in a Loewner sense, that is, for $\tau\mathbf{M}_H - \mathbf{M}_G$ to be positive semidefinite [15]. Another interpretation of the support number is that the shape of the quadratic function associated with $\tau\mathbf{M}_H - \mathbf{M}_G$ is convex.

In particular, the generalized condition number and the support number are connected via the following property:

Proposition 2. *Suppose $\mathbf{M}_G \in \mathbb{R}^{n\times n}$ and $\mathbf{M}_H \in \mathbb{R}^{n\times n}$ are symmetric and positive definite, then*

$$\kappa\left(\mathbf{M}_G, \mathbf{M}_H\right) = \sigma\left(\mathbf{M}_G, \mathbf{M}_H\right)\sigma\left(\mathbf{M}_H, \mathbf{M}_G\right). \tag{14}$$

This proposition suggests that $\mathbf{M}_H$ is a good preconditioner for $\mathbf{M}_G$ if both matrices can *support* each other with as little additional help as possible. Therefore we can instead focus on finding a preconditioner that minimizes the product of the two support numbers in (14).

Now let us turn our discussion back to the Jacobian matrices ($\mathbf{A}_G$ and $\mathbf{A}_H$) and explain another important notion in support theory: the *embedding matrix*. An embedding matrix $\mathbf{W}$ contains the coefficients to linearly synthesize each row in matrix $\mathbf{A}_G$ by using the rows in matrix $\mathbf{A}_H$. This notion is useful to characterize the support number via the following theorem:

Theorem 3. *Suppose $\mathbf{A}_G \in \mathbb{R}^{m\times n}$ is in the range of $\mathbf{A}_H \in \mathbb{R}^{p\times n}$, $\mathbf{M}_G = \mathbf{A}_G^T\mathbf{A}_G$, and $\mathbf{M}_H = \mathbf{A}_H^T\mathbf{A}_H$, then*

$$\sigma\left(\mathbf{M}_G, \mathbf{M}_H\right) = \min_{\mathbf{W}} \|\mathbf{W}\|_2^2 \text{ subject to } \mathbf{W}\mathbf{A}_H = \mathbf{A}_G. \tag{15}$$

Theorem 3 shows that the square of the spectral norm (largest singular value) of any embedding matrix provides an upper bound for the support number. The better embedding matrix we identify, the lower upper bound for the support number we obtain. However, directly working with this metric could be inefficient because there is no closed-form expression to compute the spectral norm of a matrix.

Fortunately, there are simpler matrix functions that yield upper bounds for the spectral norm, and consequently for the support number. One of them is the Frobenius norm $\|\mathbf{W}\|_F$, which is defined as the square root of the sum of squared elements in the matrix. The consequence of this fact is a well-known result in numerical linear algebra, namely $\|\mathbf{W}\|_2^2 \leq \|\mathbf{W}\|_F^2$ [16]. The Frobenius norm is easier to work with as it decouples the embedding matrix so that each of its rows can be considered independently. In the next section, we will use this inequality to develop a metric to evaluate the quality of any spanning tree preconditioner in SLAM.

C. Subgraph Preconditioners

Here we use support theory to analyze the subgraph preconditioners. By definition, if S is a subgraph of G, i.e. $\mathbf{A}_S$ consists of a subset of the rows of $\mathbf{A}_G$, then by using Theorem 3 we know $\sigma\left(\mathbf{M}_S, \mathbf{M}_G\right) \leq 1$ because there exists an embedding matrix which is a proper subset of an identity matrix. This statement is true for all well-posed linear systems. Therefore we only need to analyze the other support number $\sigma\left(\mathbf{M}_G, \mathbf{M}_S\right)$.

The support number $\sigma\left(\mathbf{M}_G, \mathbf{M}_S\right)$ bears a graphical interpretation when the Jacobian matrix is an *oriented incidence matrix*, where each row has only two nonzeros with the same magnitude but opposite signs. In this setting, the Jacobian matrices $\mathbf{A}_G$ and $\mathbf{A}_S$ can transformed into weighted graphs G and S respectively. Boman and Hendrickson showed that the *stretch* between G and S is equivalent to the Frobenius norm of the embedding matrix, which is an upper bound of the support number $\sigma\left(\mathbf{M}_G, \mathbf{M}_S\right)$ [17].

However, the stretch cannot be used to evaluate the subgraph preconditioners for SLAM because the Jacobian matrices in SLAM are more general than oriented incidence matrices: they are typically block-structured and each nonzero block could have arbitrary values. This limitation motivates us to develop a new metric to measure the quality of subgraph preconditioners for SLAM.

IV. Generalized Stretch (GST)

In this section, we will define the notion of *stretch* for the Jacobian matrices with the following properties: (1) the matrices are block-structured, (2) every nonzero block is invertible, (3) there is exactly one block-row with exactly one nonzero block, (4) the other block-rows have exactly two nonzero blocks, and (5) the matrix has full column rank. In SLAM, this setting resembles a scenario in which the robot knows its initial pose (a unary prior factor) in the world coordinate and has sensors (e.g., odometry, loop-closure) to induce pose constraints (binary factors). Hereafter we will refer to a matrix satisfying these properties as an A-matrix. Since we exclusively work with block-structured matrices, the word "block" will sometimes be omitted for simplicity.

A. Canonical Form of an A-Matrix

To facilitate the exposition, we define the canonical form of an A-matrix as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \end{bmatrix} \tag{16}$$

where

$$\mathbf{A}_0 = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{17}$$

is the row with one nonzero block, and

$$\mathbf{A}_i = \begin{bmatrix} \cdots & \mathbf{A}_{i,a_i} & \cdots & \mathbf{A}_{i,b_i} & \cdots \end{bmatrix} \tag{18}$$

is the ith row vector with two nonzero blocks (indexed by a_i and b_i), m is the number of binary factors, and n is the number of block variables. Every A-matrix can be transformed to this canonical form by permuting the rows and columns. An A-matrix is indexed by the block variables, and therefore we define

$$\mathbf{A}(i,j) = \begin{cases} \mathbf{A}_{0,0} & \text{if } i = 0 \text{ and } j = 0, \\ \mathbf{A}_{i,j} & \text{if } 1 \le i \le m \text{ and } j \in \{a_i, b_i\}, \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{19}$$

B. Transformation to an A-Graph

Here we show how to transform an A-matrix into an A-graph. We define a graph $G = (V, E)$ where $V = \{v_1, v_2, \cdots, v_n\}$ denotes a set of vertices, each of which corresponds to a column in $\mathbf{A}$, and $E = \{e_0, e_1, \cdots, e_m\}$ are the edges of G, each of which corresponds to a row in $\mathbf{A}$. With slight abuse of notation, we define $\mathbf{A}(e_i) = \mathbf{A}_i$ that associates an edge to a block row, and $\mathbf{A}(e_i, v_j) = \mathbf{A}(i,j)$ that associates a pair of vertex v_j and edge e_i to a square block matrix.

We say that edge e_i is incident to the vertex v_j if $\mathbf{A}(e_i, v_j) \neq \mathbf{0}$. Moreover, two edges e_i and e_j are adjacent if they share a vertex, denoted as $e_i \cap e_j$. For a pair of adjacent edges e_i and e_j, we define a function

$$r_{\mathbf{A}}(e_i, e_j) = \mathbf{A}(e_i, e_i \cap e_j) \cdot \mathbf{A}(e_j, e_i \cap e_j)^{-1}, \tag{20}$$

which is a square matrix that represents the *ratio* between two edges with respect to the shared vertex.

C. Path Embedding in a Spanning Tree

Central to our derivation is the notion of *path embedding*. Here we show how to compute the *path embedding* for any edge with respect to a spanning tree. We choose to investigate this case because the path embedding in a spanning tree is unique and can be derived analytically.

More specifically, suppose $T = (V, E_T)$ is a spanning tree of G, the path embedding for an edge $e_s \in E$ with respect to T consists of a set of weights

$$\mathbf{w}_T(e_s) = \{\mathbf{w}_T(e_s, e) \mid \forall e \in E_T\} \tag{21}$$

so that e_s can be perfectly reconstructed, that is,

$$\sum_{e \in E_T} \mathbf{w}_T(e_s, e)\mathbf{A}(e) = \mathbf{A}(e_s). \tag{22}$$

Since T is a spanning tree, the weights are unique and can be derived analytically. Suppose $e_s = (v_a, v_b)$ is an edge to be embedded, there are two cases: (1) If $e_s \in E_T$, then the weights are all zeros except that $\mathbf{w}_T(e_s, e_s)$ is an identity matrix. (2) If $e_s \notin E_T$, then the weights can be derived by performing Gaussian elimination from the end vertices, v_a and v_b, to the root vertex v_r of T, which is defined as the vertex with the unary prior factor. Note that $v_r = v_1$ in our canonical representation.

After a series of algebraic calculations, we can derive the weights with respect to the ratio function defined in (20):

$$\mathbf{w}_T(e_s, e) = \begin{cases} \mathbf{0} & \text{if } e \notin P_T(v_a) \cup P_T(v_b) \\ r_{\mathbf{A}}(e_s, e) & \text{if } e \cap e_s = v \\ -\sum_{e' \in \mathbf{D}_T(e)} r_{\mathbf{A}}(e', e) \cdot \mathbf{w}_T(e_s, e') & \text{o/w,} \end{cases} \tag{23}$$

where $P_T(v)$ is defined as e_0 plus the edges on the unique path between v and the v_r in T, $\mathbf{D}_T(e)$ denotes a set of edges incident to e in T leading to the vertices of greater depth. The depth of a vertex is defined as its distance to the root vertex.

D. Generalized Stretch

In support theory, the *stretch* of an edge is defined as the squared Frobenius norm of the path embedding for the oriented incidence matrices [18]. Here we use (23) to define the notion of *generalized stretch* for the A-matrices:

$$\mathbf{gst}_T(e_s) = \sum_{e \in E_T} \|\mathbf{w}_T(e_s, e)\|_{\mathrm{F}}^2. \tag{24}$$

It has been shown in [17] that the sum of stretches over all edges in G, namely

$$\mathbf{gst}_T(G) = \sum_{e_s \in E_G} \mathbf{gst}_T(e_s), \tag{25}$$

measures how well a spanning tree T serves as a preconditioner for a graph G because it corresponds to an upper bound of the generalized condition number. We will use this property to derive good subgraph preconditioners.

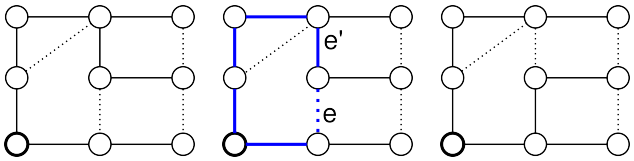


Fig. 2: Illustration of one iteration of our algorithm. (a) The current spanning tree T (solid edges). (b) Suppose the off-tree edge e is sampled. Inserting e into T would induce the blue cycle. Suppose the edge e' is sampled from the cycle. (c) Swapping e and e' leads to a new tree T'.

V. Finding Good Subgraph Preconditioners

To find a good subgraph preconditioner, a common practice is to find a low-stretch spanning tree as the skeleton, and then augment it with additional edges to further reduce the total stretch. Therefore, we first focus on solving the following problem:

$$\min_{T} \ \mathbf{gst}_T(G). \tag{26}$$

Solving (26) is an NP-hard problem. Instead of solving it directly, we give an algorithm based on MCMC techniques to find a low-stretch spanning tree. The algorithm assumes an initial spanning tree T is available. For each iteration, we sample an edge $e \notin E_T$ with a probability proportional to $\mathbf{gst}_T(e)$. Inserting e into the spanning tree leads to a new subgraph $T^+ = (V, E_T \bigcup e)$, which contains an induced cycle $C_T(e)$. To obtain a spanning tree again, we pick an edge $e' \in C_T(e)$ uniformly at random, and swap e and e' to build a new spanning tree T'. If the new total stretch $\mathbf{gst}_{T'}(G)$ is smaller than the original total stretch $\mathbf{gst}_T(G)$, then we accept T' unconditionally. Otherwise, we accept T' with a probability following an exponential distribution of the logarithm of the ratio between two stretches. Thus the algorithm can be thought of as a Markov Chain based on Metropolis updates. The above procedure is illustrated in Figure 2. We repeat this procedure until convergence. In the end, we output the best spanning tree during the course.

Given the best spanning tree T_* computed in the previous step, we construct a subgraph by inserting the edges with high stretch into the spanning tree. The rationale behind picking these edges is that they are likely to reduce the generalized condition numbers the most. We have examined two edge selection strategies. The first is to greedily pick the edges with the largest stretch. The second is to sample the edges with a probability according to the logarithm of their stretch. Please refer to Section VI-C for results. The key steps of our algorithm are summarized in Algorithm 1.

Algorithm 1: The proposed algorithm

Input: G is the graph, T_0 is a spanning tree of G
Initialization: $s_0 = \mathbf{gst}_{T_0}(G)$
for i = 0 **to** maximum iterations **do**
  if *convergent* **then** break
  1. sample an edge $e \in G$ with probability $\propto \mathbf{gst}_{T_i}(e)$
  2. let $C_{T_i}(e)$ be the unique cycle in $T^+ = (V, E_T \bigcup e)$
  3. uniformly at random sample an edge e' from $C_{T_i}(e)$
  4. swap e and e' so that $T_i' = (V, E_T \bigcup e \backslash e')$
  5. compute $s_i' = \mathbf{gst}_{T_i'}(G)$
  if $s_i' < s_i$ **then**
    $T_{i+1} = T_i'$; $s_{i+1} = s_i'$
  else
    $x = \log(\frac{s_i'}{s_i})$
    $\alpha = \min(1, \lambda \exp(-\lambda x))$
    generate a random number $q \sim U[0, 1]$
    if $q \leq \alpha$ **then** $T_{i+1} = T_i'$; $s_{i+1} = s_i'$
    else $T_{i+1} = T_i$; $s_{i+1} = s_i$
  end
end
let $T_* = \text{argmin}_{T_i} \mathbf{gst}_{T_i}(G)$
augment T_* with edges (see text), and output the subgraph

VI. Results

We conducted five experiments to evaluate the proposed algorithm: (1) We evaluated the efficiency of our MCMC algorithm. (2) We evaluated the quality of different spanning tree preconditioners. (3) Given a low-stretch spanning tree, we evaluated two different edge selection strategies to construct a subgraph. (4) We evaluated the quality of different subgraph preconditioners. (5) We used these subgraph preconditioners in the least-squares preconditioned conjugate gradient (LSPCG) method to solve both synthetic and real SLAM problems, and compare the running time against the state-of-the-art sparse direct solver [19].

To facilitate the comparison, we generated a synthetic Blockworld SLAM problem, simulating a robot traversing a block world. The bird's-eye view of this problem is illustrated in Figure 3. For each instance of the Blockworld problem, we attached a prior factor to the first robot pose to make the SLAM problems well-posed. In addition, for each robot pose we added twenty relative constraints to its closest neighbor poses, and these measurements are contaminated by zero-mean and normally distributed noise. The initial values of the robot poses are computed by composing the measurements along the robot's trajectory.

For Algorithm 1, we set $\lambda = 10^3$ and considered the algorithm is convergent if the average decrease of total stretch in the past 100 iterations is smaller than 10^{-3}.

We ran all of the experiments on a PC with an Intel Core i7 CPU, and reported the tenth percentile, the median and the ninetieth percentile over fifty trials.

A. Efficiency of Our MCMC Algorithm

We evaluated the efficiency of our MCMC algorithm by measuring the required time and iterations to converge for the Blockworld problem. For each instance of the Blockworld problem, starting from a random spanning tree, we applied our algorithm to find a low-stretch spanning tree and reported the results in Figure 4. We can see that as the problem size increases, the number of required iterations stays almost constant, which indicates that a good tree can be found in a constant number of edge swaps. However, the required time increases linearly with the problem size, which negatively

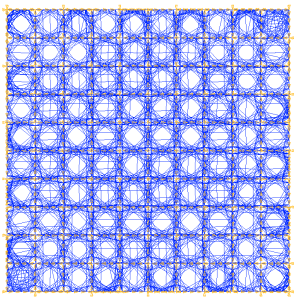


Fig. 3: The bird's-eye view of a Blockworld problem with 1,000 robot poses (yellow) and 10,000 constraints (blue).

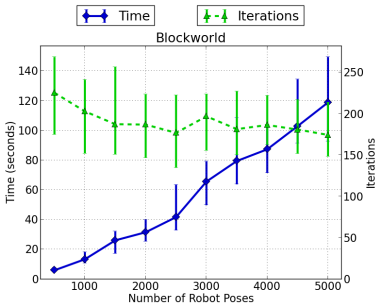


Fig. 4: The efficiency of our MCMC algorithm.

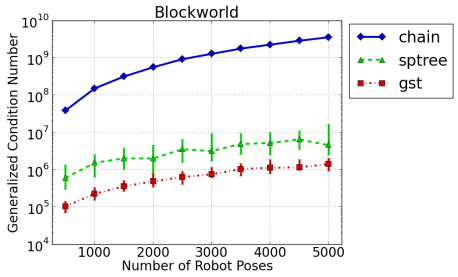


Fig. 5: The generalized condition numbers of different spanning tree preconditioners.

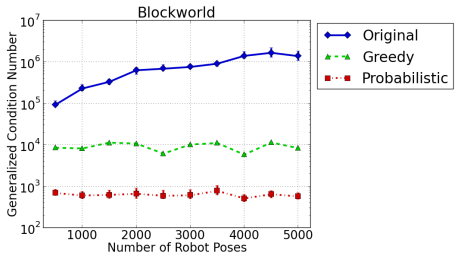


Fig. 6: The generalized condition numbers resulted by using different edge selection strategies to build a subgraph.

affects the performance of our algorithm for large-scale problems. We plan to resolve this problem in future work.

B. Generalized Condition Numbers of Spanning Trees

We compared three spanning tree preconditioners for the Blockworld problem: (1) the odometry chain of the robot poses (chain), (2) a random spanning tree (sptree), and (3) the spanning tree computed by the proposed algorithm, but without additional edges (gst). The first setting characterizes the performance of an ad-hoc deterministic spanning tree. The second setting characterizes the empirical performance of a random spanning tree. To build a random spanning tree, we assigned a random weight from 1 to 100 to each edge of the graph, and computed the maximum-weighted spanning tree with Kruskal's algorithm [20].

Once the spanning tree is determined, we used CHOLMOD [19], an efficient sparse direct solver, to compute the preconditioner, and used ARPACK [21], a popular solver for the sparse eigenvalue problem, to compute the generalized condition numbers.

Figure 5 shows the generalized condition numbers of the preconditioned systems. We can see that a random spanning tree (sptree) is better than the odometry chain (chain) by one to two orders of magnitude. Moreover, we can see that our low-stretch spanning tree (gst) is better than a random spanning tree (sptree) by another order of magnitude. This result confirms that our algorithm indeed produces good spanning trees.

C. Subgraph Construction

Given the low-stretch spanning tree computed in the previous step, we evaluated the performance of the two edge selection strategies to construct a subgraph. More specifically, we examined the following two strategies: (1) greedily pick the edges with the largest stretch, (2) probabilistic sample the edges according to the logarithm of their stretch.

We conducted experiments on the Blockworld problem. For each instance of the Blockworld problem, we applied our algorithm to computed a low-stretch spanning tree which serves as the baseline (Original). Then we used the two strategies to insert n edges to each of the spanning trees to build subgraph preconditioners, where n is the number of robot poses. For the Greedy setting, we sorted edges according to their stretch, and pick the top-n edges. For the Probabilistic setting, we sampled n additional edges with probability proportional to the logarithm of their stretch, and inserted them into the spanning tree to build a subgraph. Once the subgraph is determined, we used the same procedure described in Sec. VI-B to compute the generalized condition numbers.

From the results in Figure 6, we can see that the subgraphs generated by both strategies can significantly improve the generalized condition numbers of the spanning trees by at least one order of magnitude. It implies that inserting additional edges to a spanning tree indeed leads to a better preconditioner. Comparing these two strategies, we can see that Probabilistic is better than Greedy by one to two orders of magnitude. We conjecture that it is because the edges chosen by Greedy may concentrate on a certain part of the graph, and therefore fail to reduce the stretch for the other parts of the graph. On the other hand, the edges chosen by Probabilistic have a higher chance to spread over the graph, and therefore could reduce the total stretch even further.

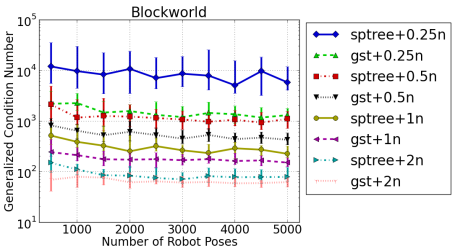


Fig. 7: The generalized condition numbers of different subgraph preconditioners.

D. Generalized Condition Numbers of Subgraphs

We compared the generalized condition numbers of two subgraph preconditioners: (1) a random spanning tree augmented with $c \cdot n$ additional random edges (sptree+cn), and (2) a low-stretch spanning tree augmented with $c \cdot n$ additional edges sampled by using the Probabilistic strategy (gst+cn), where c is a scalar and n is the number of robot poses.

From the results in Figure 7, we can see that as the amount of additional edges increases, the generalized condition numbers of both subgraph preconditioners decrease consistently. Moreover, the results also show that gst+cn is consistently better than sptree+cn, but the difference is inversely proportional to c. This phenomenon is expected because as c becomes larger, the subgraphs produced by gst+cn and sptree+cn will become more similar, eventually both of them will produce the original graph.

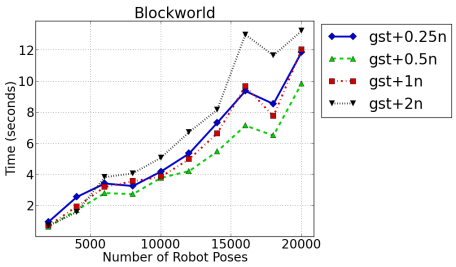
E. Timing Results on Synthetic Datasets

We evaluated the running time of using different subgraph preconditioners in the LSPCG method [14] to solve the Blockworld problem and compared the performance against the state-of-the-art sparse direct solver (CHOLMOD [19]).

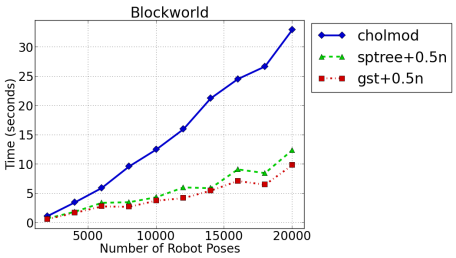
We generated Blockworld datasets up to twenty thousand robot poses, and ran the Gauss-Newton algorithm to solve the nonlinear SLAM problem for ten iterations. In each iteration, we used either LSPCG or CHOLMOD to solve the linear systems. For the LSPCG method, we used either sptree+cn or gst+cn as the preconditioners. The stopping criteria for LSPCG are (1) the norm of the current gradient is smaller than 10^{-2} times of the initial gradient, or (2) the number of LSPCG iterations exceeds one thousand. For CHOLMOD, we use the implementation in SuiteSparse compiled with GotoBlas2. All of the solvers run with single thread. Since different settings achieve different errors in the end, we reported the time to achieve the error that is ϵ-close to the optimum, i.e.,

$$\frac{|e - e_*|}{|e_0 - e_*|} \leq \epsilon \tag{27}$$

where e_0 is the initial error, e is the current error, e_* is the minimum achieved error of all solvers, and ϵ is a threshold. We set $\epsilon = 10^{-6}$ in our experiments. Notice that since our algorithm to find a good subgraph is still inefficient, we *excluded* the time spent in Algorithm 1, and focused on comparing the efficiency of different solvers.



(a)



(b)

Fig. 8: The timing results of different linear solvers.

We evaluated the performance of LSPCG solvers with the gst+cn preconditioners, and show the results in Figure 8a. We can see that gst+0.5n is the most efficient among all of the settings. These results suggest that the most efficient subgraph preconditioner is not necessarily the one with the most edges, and establishing the number of additional edges to augment a spanning tree involves a trade-off: Inserting too few edges into the subgraph may not lead to an effective preconditioner, while inserting too many edges into the subgraph may slow down the overall performance. We will further investigate this issue in future work.

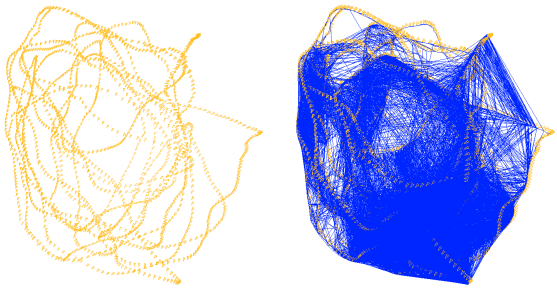
Then we compared the LSPCG solvers with CHOLMOD and showed the results in Figure 8b. Note that for the LSPCG solvers, we only showed the results for gst+0.5n and sptree+0.5n for clarity. The performance of the other settings are inferior in this experiment. We can see that CHOLMOD is two to three times slower than the LSPCG solvers, and it suggests that direct solvers are not suitable for large-scale problems. Comparing the two preconditioners, we can see that sptree+0.5n is 5-30% slower than gst+0.5n. This result suggests our subgraph preconditioner is more effective than a random subgraph preconditioner.

F. Timing Results on Real Dataset

We also evaluated the performance of these solvers on a real dataset. Existing public SLAM datasets mostly have highly sparse graphs which cannot demonstrate the advantages of iterative solvers. To this end, we created a real dataset of 2,000 images with a Videre STOC camera in an office environment, where the camera constantly visits the same place to create many loop-closure constraints. Figure 9 shows the sample images and the bird's-eye view of the camera trajectory and the pose constraints of this dataset.

We used the visual odometry pipeline presented in [22] to initialize the robot poses by composing the relative pose

(a)



(b) (c)

Fig. 9: The real dataset. (a) Four sample images. (b) The bird's-eye view of the camera poses. (c) The bird's-eye view of the camera poses overlayed with the pose constraints.

TABLE I: The timing result on the real dataset in seconds.

gst+1n		sptree+1n	CHOLMOD
Algo. 1	others		
15.4	3.7	5.0	8.8

constraints along the image sequence, and then used the vocabulary tree technique [23] to generate 33,234 loop-closure constraints.

We applied gst+1n, sptree+1n with LSPCG and CHOLMOD to solve this dataset and reported the running time in Table I. We set the scalar $c = 1$ as it gave the best performance in this experiment. We can see that although our MCMC algorithm took 15.4 seconds to find the subgraph, our subgraph preconditioner gst+1n is 25% and 58% faster than sptree+1n and CHOLMOD respectively in terms of solving the problem.

VII. Conclusions and Future Work

In this paper, we propose a new metric based on support theory to evaluate the quality of a spanning tree preconditioner for SLAM. We use this metric to develop an MCMC-based algorithm to find good subgraph preconditioners. To the best of our knowledge, this is the first attempt to derive theoretically good subgraph preconditioners for SLAM. We apply them to solve synthetic and real SLAM problems, and the results show that the proposed metric is effective and the resulting subgraph preconditioners display significant improved efficiency over the state-of-the-art solver.

There are several directions for future work. The first is to improve the efficiency of the proposed MCMC algorithm. The second is to generalize other combinatorial notions such as *congestion* and *dilation* [18], and apply them to find good subgraphs for SLAM. So far we have focused on the batch version of the SLAM problem; another interesting direction is to adapt the proposed algorithm to online scenarios.

References

[1] H. Durrant-Whyte and T. Bailey, "Simultaneous localisation and mapping (SLAM): Part I the essential algorithms," *Robotics & Automation Magazine*, Jun 2006.

[2] T. Bailey and H. Durrant-Whyte, "Simultaneous localisation and mapping (SLAM): Part II state of the art," *Robotics & Automation Magazine*, Sep 2006.

[3] F. Dellaert and M. Kaess, "Square root SAM: Simultaneous localization and mapping via square root information smoothing," *International Journal of Robotics Research*, vol. 25, no. 12, pp. 1181–1203, 2006.

[4] T. Davis, *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[6] F. Dellaert, J. Carlson, V. Ila, K. Ni, and C. E. Thorpe, "Subgraph-preconditioned conjugate gradient for large scale SLAM," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

[7] E. Boman and B. Hendrickson, "Support theory for preconditioning," *SIAM Journal on Matrix Analysis and Applications*, vol. 25, no. 3, pp. 694–717, 2003.

[8] W. R. Gilks, S. Richardson, and D. Spiegelhalter, *Markov Chain Monte Carlo in Practice: Interdisciplinary Statistics*. Chapman & Hall/CRC, 1995, vol. 2.

[9] I. Abraham and O. Neiman, "Using petal-decompositions to build a low stretch spanning tree," in *ACM Symp. on Theory of Computing (STOC)*. ACM, 2012, pp. 395–406.

[10] I. Koutis, G. Miller, and R. Peng, "A nearly-m*logn solver for SDD linear systems," in *Symp. on Foundations of Computer Science (FOCS)*, 2011.

[11] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, "A simple, combinatorial algorithm for solving sdd systems in nearly-linear time," *arXiv preprint arXiv:1301.6628*, 2013.

[12] F. Kschischang, B. Frey, and H. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[13] J. Nocedal and S. Wright, *Numerical Optimization*. Springer Verlag, 1999.

[14] A. Björck, *Numerical Methods for Least Squares Problems*. SIAM, 1996.

[15] F. Zhang, *Matrix Theory: Basic Results and Techniques*. Springer, 2011.

[16] G. Golub and C. Van Loan, *Matrix Computations*. Johns Hopkins University Press, 1996, vol. 3.

[17] E. Boman and B. Hendrickson, "On spanning tree preconditioners," *Manuscript, Sandia National Laboratories*, 2001.

[18] S. Toledo and H. Avron, *Combinatorial Scientific Computing*, ser. Chapman & Hall/CRC Computational Science. CRC Press, 2011, ch. 11.

[19] Y. Chen, T. Davis, W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 22, 2008.

[20] J. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[21] R. Lehoucq, D. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[22] C. Beall, B. Lawrence, V. Ila, and F. Dellaert, "3d reconstruction of underwater structures," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010, pp. 4418–4423.

[23] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 2. IEEE, 2006, pp. 2161–2168.