On the Tunable Sparse Graph Solver for Pose Graph Optimization in Visual SLAM Problems

Chieh Chou, Di Wang, Dezhen Song, and Timothy A. Davis

***Abstract*— We report a tunable sparse optimization solver that can trade a slight decrease in accuracy for significant speed improvement in pose graph optimization in visual simultaneous localization and mapping (vSLAM). The solver is designed for devices with significant computation and power constraints such as mobile phones or tablets. Two approaches have been combined in our design. The first is a graph pruning strategy by exploiting objective function structure to reduce the optimization problem size which further sparsifies the optimization problem. The second step is to accelerate each optimization iteration in solving increments for the gradient-based search in Gauss-Newton type optimization solver. We apply a modified Cholesky factorization and reuse the decomposition result from last iteration by using Cholesky update/downdate to accelerate the computation. We have implemented our solver and tested it with open source data. The experimental results show that our solver can be twice as fast as the counterpart while maintaining a loss of less than 5% in accuracy.**

I. Introduction

Visual simultaneous localization and mapping (SLAM) algorithms allows mobile robots or devices to precisely estimate their location and establish visual scene understanding. The recent fast development of augmented reality (AR) applications on mobile devices is essentially enabled by visual SLAM (vSLAM) algorithms. Despite real time requirement, vSLAM algorithms face strict power and computation constraints when running on mobile devices.

A significant part of vSLAM computation is pose graph optimization. This step is also known as local bundle adjustment (LBA) because it performs the optimization to refine landmark locations and camera poses on a sliding window of adjacent frames to avoid a large global optimization problem. It has been widely accepted as an efficient approach because landmarks only continuously exist in a short sequence of adjacent frames. The short temporal dependence on landmarks means that the optimization problem is sparse. The optimization problem has a graph structure with landmarks and camera poses as vertices.

To speed up the computation, we propose a tunable sparse solver to solve the graph optimization problem with faster speed and similar accuracy than that of the traditional solver. Unlike the traditional solver, which usually solve the whole graph, we concentrate the computation on partial graph with high contribution to the cost function and the increments. We design two algorithms to accelerate the optimization process. First, we propose a graph pruning algorithm to reduce the original optimization problem size by exploiting the LBA structure. We only optimize vertices with large errors and fix the others to generate a pruned graph. Second, we apply a modified Cholesky factorization to speed up the computation. In each iteration, we only update a sub graph and then we reuse the decomposition result from last iteration by using Cholesky update/downdate to reduce the repeated computation (see the example in Fig. 1).

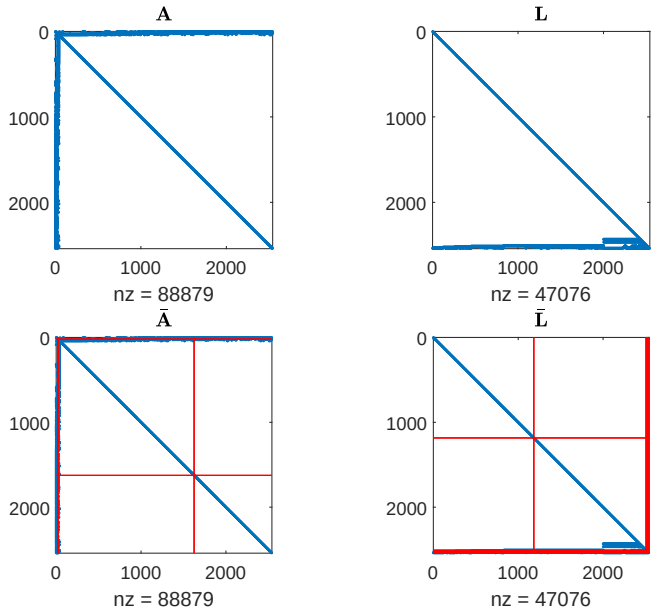


Fig. 1. Computation savings by updating system matrix $\mathbf{A}$ (top left) and its companion lower triangular matrix $\mathbf{L}$ (top right) in LBA problem from one iteration to next (i.e. the new counter part $\bar{\mathbf{A}}$ (bottom left) and $\bar{\mathbf{L}}$ (bottom right)) instead of regenerating the entire matrices. The example is generated using data from dataset KITTI00. The non-zero (nz) entries in four matrices are in blue. By using Cholesky multiple-rank update to update one edge, we are able to only update the part located at the intersection of the red columns except the upper triangular matrix in $\bar{\mathbf{L}}$.

We have implemented our solver by revising g2o [1] which is also used as our baseline. The results show that our solver can be twice as fast as the traditional solver for the LBA processes in vSLAM problems, when the loss in accuracy is no more than 5% on average.

II. Related Work

Our work relates to the areas of visual SLAM, optimization solver, and incremental solver.

As a variation of the SLAM problem [2], the visual SLAM problem is to simultaneously estimate robot pose

C. Chou, D. Wang, D. Song, and T. A. Davis are with CSE Department, Texas A&M University, College Station, TX 77843, USA, Email: dzsong@cs.tamu.edu.

This work was supported in part by National Science Foundation under NRI-1526200, 1748161, and 1925037.

and landmark positions using features from one or more cameras. Those features, with points as the most commonly employed feature type, are extracted from video frames using state-of-the-art feature detection methods such as SIFT [3], SURF [4], etc. The landmark positions are often represented as 3D positions of the feature points. The raw features have been through a process of selection, filtering, and identifying correspondence with features in adjacent frames using epipolar geometry in computer vision, and many statistics methods such as random sampling consensus (RANSAC) [5]. The visual SLAM process then estimates both camera poses and 3D landmarks from 2D features by minimizing reprojection error over the chosen set of the image frames. To solve the visual SLAM problem, there are two dominant approaches [6]: the filtering approaches (e.g., [7], [8]) and the bundle adjustment (BA) approaches (e.g., [9], [10]). The former comes from the traditional SLAM field of robotics research, while the latter builds on the optimization framework introduced by the structure from motion area in computer vision. We focus on improving the latter since it gains more popularity.

An optimization solver is indispensable to reduce uncertainty for SLAM problems. After obtaining the initial solutions, the solutions can be further improved by formulating an optimization problem in BA approaches and solving it using an optimization solver, such as ceres-solver [11] or g2o [1]. Moreover, the pose graph optimization can be applied to not only SLAM problems (e.g., [12], [13]), but also other applications such as map fusion (e.g., [14], [15]). However, solving pose graph optimization problems suffers from high computational cost. For efficiency purpose, it is common to apply LBA [16], [17] in the front end of SLAM problems instead of using global bundle adjustments.

Incremental solvers for SLAM problems become more and more prevalent recently since it can reduce repeated computation by taking advantage of previous results to update and solve the problems. Dellaert et al. propose $\sqrt{\text{SAM}}$ [18] to incrementally solve the problem by using a sparse direct solver. Kaess et al. further improve $\sqrt{\text{SAM}}$ by applying incremental QR factorization in iSAM [19] and Bayes tree in iSAM2 [20]. Polok et al. utilize incremental block Cholesky factorization [21] to speed up the solver. Ila et al. recover the estimation and covariance matrix in SLAM++ [22] and present an incremental BA to update Schur complement [23]. Liu et al. further propose an incremental BA solver for visual-inertial SLAM problems [24] Wang et al. improve iSAM and iSAM2 by incremental Cholesky factorization and min-heap based variable reordering [25].

Our work focus on improve LBA speed by utilizing its structure. From the algorithm aspect, we use Cholesky update/downdate algorithms in popular sparse matrix library CHOLMOD [26]–[31] to solve vSLAM problems. This approach can efficiently make use of previous results since there is no need to reconstruct the optimization iteration equations and recompute the Cholesky factorization.

III. Algorithms

A. Graph Optimization using the Classic Solver

First, we review the background of a graph optimization problem [1]. Let G denotes a graph, V_i denotes i-th vertex, and E_{ij} denotes the edge connecting V_i and V_j. Let $\mathbf{x}_{v_i}$ be the estimated parameter in V_i, and ε_{ij} be the error term in E_{ij}. Let $\mathbf{x} \in \mathbb{R}^M$ be the parameter vector obtained by stacking all $\mathbf{x}_{v_j}$, $\mathbf{z} \in \mathbb{R}^N$ be the measurement vector, and $f : \mathbb{R}^M \to \mathbb{R}^N$ be a differentiable mapping. The cost function is given by

$$F(\mathbf{x}) = \boldsymbol{\varepsilon}^\mathsf{T} \Sigma^{-1} \boldsymbol{\varepsilon}, \tag{1}$$

where $\boldsymbol{\varepsilon} = f(\mathbf{x}) - \mathbf{z}$ is the error term by stacking all ε_{ij} and Σ is the covariance matrix of gaussian noise in measurement. Finally, $\mathbf{x}$ can be solved by $\min_{\mathbf{x}} F(\mathbf{x})$.

To solve the optimization problem, the prevailing approach is to start from a good initial solution, and then further refine it iteratively by numerical approaches such as Gauss-Newton (GN) method or its variant such as Levenberg-Marquardt (LM) method [32]. Given an initial solution $\mathbf{x}$, the goal is to find a refined solution $\bar{\mathbf{x}} = \mathbf{x} + \Delta\mathbf{x}$, where $\Delta\mathbf{x}$ is the increment, such that $F(\bar{\mathbf{x}}) < F(\mathbf{x})$. Assuming f is locally linear so it can be approximated by first-order Taylor expansion at $\mathbf{x}$ as

$$f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}, \tag{2}$$

where $\mathbf{J} = \frac{\partial f}{\partial \mathbf{x}}$ is the Jacobian matrix. By substituting (2) into (1), the cost function $F(\bar{\mathbf{x}})$ cab be is approximated by

$$F(\mathbf{x} + \Delta\mathbf{x}) \simeq F(\mathbf{x}) + 2\Delta\mathbf{x}^\mathsf{T}\mathbf{J}^\mathsf{T}\Sigma^{-1}\boldsymbol{\varepsilon} + \Delta\mathbf{x}^\mathsf{T}\mathbf{J}^\mathsf{T}\Sigma^{-1}\mathbf{J}\Delta\mathbf{x}. \tag{3}$$

Then we take the derivative of (3) and derive the normal equation

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{b}, \tag{4}$$

where $\mathbf{A} = \mathbf{J}^\mathsf{T}\Sigma^{-1}\mathbf{J}$ is the system matrix and $\mathbf{b} = -\mathbf{J}^\mathsf{T}\Sigma^{-1}\boldsymbol{\varepsilon}$ is the right-hand side (RHS). Assume that the fill-reducing permutation is already applied [33], [34], we solve the normal equation by first performing Cholesky factorization on $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}$, where $\mathbf{L}$ is the lower triangular matrix, and $\mathbf{D}$ is the diagonal matrix. Thus (4) becomes $\mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}\Delta\mathbf{x} = \mathbf{b}$. Finally, we utilize forward solve to $\mathbf{L}\mathbf{y} = \mathbf{b}$, where $\mathbf{y} = \mathbf{D}\mathbf{L}^\mathsf{T}\Delta\mathbf{x}$, and apply backsolve to $\mathbf{D}\mathbf{L}^\mathsf{T}\Delta\mathbf{x} = \mathbf{y}$ to obtain $\Delta\mathbf{x}$. Alg. 1 summarize the approach as the classic solver. ε_0 is an empirical threshold applied when the increment is negligible.

Algorithm 1: $(\bar{\mathbf{x}}, \Delta\bar{\mathbf{x}}, \textbf{STOP})$=ClassicSolver$(\mathbf{x}, G)$

Input: $\mathbf{x}, G$
Output: $\bar{\mathbf{x}}, \Delta\bar{\mathbf{x}}, \textbf{STOP}$
Build $\{\mathbf{A}, \mathbf{b}\}$ from $\mathbf{x}$ and G ;
Decompose $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}$, and solve $\mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}\Delta\mathbf{x} = \mathbf{b}$;
if $\|\Delta\mathbf{x}\| < \varepsilon_0$ **then**
  $\bar{\mathbf{x}} = \mathbf{x}$, and **STOP** = TRUE ;
else
  $\bar{\mathbf{x}} = \mathbf{x} + \Delta\mathbf{x}$, and **STOP** = FALSE ;
end
$\Delta\bar{\mathbf{x}} = \Delta\mathbf{x}$;
return $\bar{\mathbf{x}}, \Delta\bar{\mathbf{x}}, \textbf{STOP}$

In Gauss-Newton class of optimization solvers [35], Alg. 1 is repeatedly called to generate next search starting point because it provides the new best solution in the form of $\Delta\mathbf{x}$. The whole iterative process stops when the stopping criteria is met. The process is summarized in Alg. 2. We name it as the original solver (ORI). We use the subindex k to indicate the frame index in algorithms. Here $\mathbf{x}_k$ denotes all the state variable in k-th frame before optimization, and $\bar{\mathbf{x}}_k$ denotes that after optimization. As a convention, we define the symbol bar ($\bar{}$) to indicate that the object is modified/updated in this paper. Because Alg. 2 builds the system $\{\mathbf{A}_k, \mathbf{b}_k\}$ from $\mathbf{x}_k$ and solves $\mathbf{A}_k\Delta\mathbf{x}_k = \mathbf{b}_k$ for each iteration, it is massive computational and time-consuming.

Algorithm 2: Original Solver

Input: $\{\mathbf{x}_k|k=1,2,\dots\}$, $\{G_k|k=1,2,\dots\}$
Output: $\{\bar{\mathbf{x}}_k|k=1,2,\dots\}$

```
for k = StartFrame to EndFrame do
    for Iteration = 1 to MaxIteration do
        (x̄_k, Δx̄_k, STOP)=ClassicSolver(x_k, G_k) ;
        if STOP then
            Break ;
        else
            x_k = x̄_k ;
        end
    end
end
return {x̄_k | k = 1,2,...}
```

B. Graph Pruning

Now, let us introduce our algorithm developments to accelerate the process. We propose two algorithms to speed up Alg. 1 and we begin with the first one, graph pruning.

To accelerate LBA process, we use graph pruning to reduce the original problem by spending computational efforts only on those vertices with high errors. For each pose vertex in the given graph, we first search for all connected landmark vertices, and then we keep those landmark vertices with high errors in the graph to be optimized and fix the others. A fixed vertex still contributes to the graph by providing constraints but a removed vertex does not.

Let $\mathcal{V}_p$ and $\mathcal{V}_l$ be the set containing camera pose vertices and landmark vertices, respectively. Let G and G_{pruned} be the graph before and after pruning. Recall that E_{ij} is the edge connecting V_j and V_i with ε_{ij} as its error. Here we denote $V_j \in \mathcal{V}_p$ and $V_i \in \mathcal{V}_l$ in our algorithm. ε_{χ^2} is a empirical threshold to determine if a vertex should be fixed or not. We summarize our graph pruning in Alg. 3.

The graph pruning can significantly sparsify the problem as shown in the example in Fig. 2. The original graph in Fig. 2(a) is reduced to the graph in Fig. 2(b).

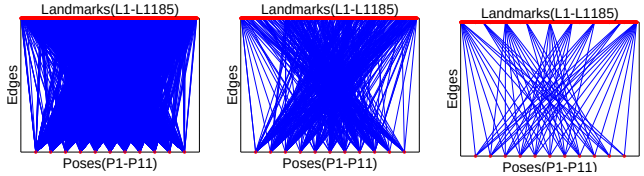
C. Update by Modified Cholesky Factorization

Next, we introduce the second algorithm to further accelerate Alg. 1 by utilizing modified Cholesky factorization.

Algorithm 3: G_{pruned}=GraphPruning(G)

Input: G
Output: G_{pruned}

```
G_pruned = G ;
for V_j ∈ V_p in G_pruned do
    for E_ij do
        if ε_ij < ε_χ² and V_i is unfixed then
            Fix V_i ;
        end
    end
end
return G_pruned
```



(a) (b) (c)

Fig. 2. These graphs represent one LBA problem obtained by running visual SLAM in KITTI00 dataset. The red dots on the top and bottom indicate landmark and pose vertices, respectively. The blue edges indicate the measurement between one pose vertex and one landmark vertex. (a) The original graph. The traditional solver often optimizes the whole original graph. (b) The pruned graph. We first design a graph pruning algorithm to shrink the original graph. (c) One sub graph of the pruned graph when applying modified Cholesky factorization.

We know more iterations in LBAs can minimize errors for better performance, but to build and solve (4) in multiple iterations within each LBA is often the most time consuming part. The iteration number depends on whether the increment is lower than a preset threshold, or the iteration number exceeds default value. Let $\Delta\mathbf{x}_{v_j}$ be the increment for vertex V_j, where $\Delta\mathbf{x}_{v_j}$ is a member of $\Delta\mathbf{x}$. By observing the increments of vertices $\Delta\mathbf{x}_{v_j}$ in each iteration, we find out that only few $\Delta\mathbf{x}_{v_j}$ have large norm, while the others are negligible. As a result, if a preset threshold is chosen prudently, then only a small portion of $\Delta\mathbf{x}$ needs to updated.

Therefore, when building (4), we only update the small portion of the increment. Besides, we reuse the decomposition solution from previous iteration and solve $\mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}\Delta\mathbf{x} = \mathbf{b}$ by using Cholesky update/downdate. For example, one sub graph (Fig. 2(c)) of the pruned graph (Fig. 2(b)) is updated in one iteration. We introduce the update strategies as follows.

1) Update $\mathbf{x}$ *:* We first check each vertex V_j to be updated or not. Given $\Delta\mathbf{x}$ from last iteration, we update V_j by

$$\bar{\mathbf{x}}_{v_j} = \begin{cases} \mathbf{x}_{v_j} + \Delta\mathbf{x}_{v_j}, & \text{IF } \|\Delta\mathbf{x}_{v_j}\| \geq \varepsilon_v, \\ \mathbf{x}_{v_j}, & \text{Otherwise,} \end{cases} \tag{5}$$

where $\bar{\mathbf{x}}_{v_j}$ is the estimation after update, and ε_v is the preset threshold. This differs from Alg. 1 which updates all V_j.

2) Update $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}$ *:* After obtaining $\bar{\mathbf{x}}$, we are able to construct $\bar{\mathbf{A}}$. Instead of constructing $\bar{\mathbf{A}}$ from scratch, and then computing $\bar{\mathbf{A}} = \bar{\mathbf{L}}\bar{\mathbf{D}}\bar{\mathbf{L}}^\mathsf{T}$, we make use of $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}$

by updating from last iteration. To update $\bar{\mathbf{x}}_{v_j}$ from $\mathbf{A}$, we have to update all the edges connected to V_j. The example in Fig. 1 shows that only a very few entries are changed when we update one edge. Since only a small portion of $\bar{\mathbf{x}}$ is updated, the updated edges are not too many so we can obtain $\{\bar{\mathbf{L}}, \bar{\mathbf{D}}\}$ from $\{\mathbf{L}, \mathbf{D}\}$ by using Cholesky multiple-rank update as follows

$$\bar{\mathbf{L}}\bar{\mathbf{D}}\bar{\mathbf{L}}^{\top} = \mathbf{L}\mathbf{D}\mathbf{L}^{\top} + \mathbf{J}_{\Delta}^{\top}\mathbf{J}_{\Delta}, \tag{6}$$

where $\mathbf{J}_{\Delta} = [\dots \mathbf{J}_{v_j} \dots]$ is the Jacobian matrices for those vertices V_j being updated, and $\mathbf{J}_{v_j} = \frac{\partial f}{\partial \mathbf{x}_{v_j}}$ is the Jacobian matrices of vertex V_j . This approach can save a lot of computation because we only compute the updated portion and update the system from last iteration rather than solve it from scratch. Besides, we don't need to factorize $\bar{\mathbf{A}} = \bar{\mathbf{L}}\bar{\mathbf{D}}\bar{\mathbf{L}}^{\top}$ since it is done after updating. More details regarding Cholesky multiple-rank update/downdate can be found in [26]–[31].

3) Update $\mathbf{b}$ *:* To update $\mathbf{b}$ is based on the same principle:

$$\bar{\mathbf{b}}_{v_j} = \begin{cases} -\mathbf{J}_{v_j}^{\top}\Sigma_{v_j}^{-1}\varepsilon_{v_j}, & \text{IF } \|\Delta\mathbf{x}_{v_j}\| \geq \varepsilon_v, \\ \mathbf{b}_{v_j}, & \text{Otherwise}, \end{cases} \tag{7}$$

where Σ_{v_j} is the covariance matrix for $\mathbf{x}_{v_j}$, and ε_{v_j} is the error term for $\mathbf{x}_{v_j}$, which is the summation of ε_{ij} with all the V_i connected to V_j.

4) Update Solver: We summarize the update strategies in Alg. 4 as the update solver. Let $\mathscr{V}_{mdf}$ be the set containing all vertices that need to be updated. The update solver takes advantage of previous results to update $\{\bar{\mathbf{L}}, \bar{\mathbf{D}}, \bar{\mathbf{b}}\}$ and solve $\bar{\mathbf{L}}\bar{\mathbf{D}}\bar{\mathbf{L}}^{\top}\Delta\mathbf{x} = \bar{\mathbf{b}}$, which is able to save computational time for rebuilding the system $\{\bar{\mathbf{A}}, \bar{\mathbf{b}}\}$ and re-decomposing $\bar{\mathbf{A}} = \bar{\mathbf{L}}\bar{\mathbf{D}}\bar{\mathbf{L}}^{\top}$.

Algorithm 4: $(\bar{\mathbf{x}}, \Delta\bar{\mathbf{x}})$=UpdateSolver($\mathbf{x}$,$\mathscr{V}_{mdf}$)

Input: $\mathbf{x}$,$\mathscr{V}_{mdf}$
Output: $\bar{\mathbf{x}}, \Delta\bar{\mathbf{x}}$
Update $\bar{\mathbf{x}}$ by (5) ;
Update $\{\bar{\mathbf{L}}, \bar{\mathbf{D}}\}$ by (6) ;
Update $\bar{\mathbf{b}}$ by (7) ;
Solve $\bar{\mathbf{L}}\bar{\mathbf{D}}\bar{\mathbf{L}}^{\top}\Delta\mathbf{x} = \bar{\mathbf{b}}$;
$\Delta\bar{\mathbf{x}} = \Delta\mathbf{x}$;
return $\bar{\mathbf{x}}, \Delta\bar{\mathbf{x}}$

IV. Tunable Sparse Graph Optimization Solver

So far we only introduce algorithms to accelerate Alg. 1. Now let us expand it to the entire graph solver. We propose our tunable sparse solver (TSS) to replace ORI in Alg. 2.

We propose our tunable sparse solver by Alg. 5. Recall that we use ε_v as the threshold for all vertices in Section III-C. Here we extend it to ε_p and ε_l, where ε_p is the threshold for camera pose vertices and ε_l is the threshold for landmark vertices. And let ε_{up} be the threshold satisfying $0 \leq \varepsilon_{up} \leq 1$ to determine to call Alg. 1 or Alg. 4. All the thresholds are obtained through experiments.

Algorithm 5: Tunable Sparse Solver

Input: $\{\mathbf{x}_k | k = 1, 2, \dots\}$, $\{G_k | k = 1, 2, \dots\}$
Output: $\{\bar{\mathbf{x}}_k | k = 1, 2, \dots\}$
for *i =StartFrame to EndFrame* **do**
  OriginalGraph = TRUE ;
  for *Iteration* = 1 *to MaxIteration* **do**
    if *Iteration* == 1 **then**
      $(\bar{\mathbf{x}}_k, \Delta\bar{\mathbf{x}}_k,$ **STOP**$)$=ClassicSolver$(\mathbf{x}_k, G_k)$;
      if STOP then
        Break ;
      else
        $\mathbf{x}_k = \bar{\mathbf{x}}_k$;
      end
    else
      if OriginalGraph then
        $\bar{G}_k$=GraphPruning(G) ;
        OriginalGraph = FALSE ;
      end
      $\mathscr{V}_{mdf} = \emptyset$;
      for $V_j \in \mathscr{V}_p$ **do**
        if $\|\Delta\bar{\mathbf{x}}_{v_j}\| > \varepsilon_p$ **then**
          Save $\{V_j, \Delta\bar{\mathbf{x}}_{v_j}\}$ to $\mathscr{V}_{mdf}$;
        end
      end
      if $|\mathscr{V}_{mdf}| \neq \emptyset$ **then**
        $(\bar{\mathbf{x}}_k, \Delta\bar{\mathbf{x}}_k,$ **STOP**$)$=ClassicSolver$(\mathbf{x}_k, \bar{G}_k)$;
        if STOP then
          Break ;
        else
          $\mathbf{x}_k = \bar{\mathbf{x}}_k$;
        end
      else
        for $V_j \in \mathscr{V}_l$ **do**
          if $\|\Delta\bar{\mathbf{x}}_{v_j}\| > \varepsilon_l$ **then**
            Save $\{V_j, \Delta\bar{\mathbf{x}}_{v_j}\}$ to $\mathscr{V}_{mdf}$;
          end
        end
        if $\frac{|\mathscr{V}_{mdf}|}{|\mathscr{V}_l|} > \varepsilon_{up}$ **then**
          $(\bar{\mathbf{x}}_k, \Delta\bar{\mathbf{x}}_k,$ **STOP**$)$=ClassicSolver$(\mathbf{x}_k, \bar{G}_k)$;
          if STOP then
            Break ;
          else
            $\mathbf{x}_k = \bar{\mathbf{x}}_k$;
          end
        else if $|\mathscr{V}_{mdf}| = \emptyset$ **then**
          Break ;
        else
          $(\bar{\mathbf{x}}_k, \Delta\bar{\mathbf{x}}_k)$=UpdateSolver$(\mathbf{x}_k, \mathscr{V}_{mdf})$;
        end
      end
    end
  end
end
return $\{\bar{\mathbf{x}}_k | k = 1, 2, \dots\}$

Alg. 5 consists of four main steps: 1) initialization, 2) graph pruning, 3) pose update, and 4) landmark update. In initialization, we call Alg. 1 at first iteration for each frame to obtain $\Delta\bar{\mathbf{x}}_k$ for future use. In graph pruning, we check if the graph has been pruned or not. If not, we run Alg. 3 to prune the original graph.

In pose update, we check if any pose vertex $V_j \in \mathcal{V}_p$ needs to be updated by comparing $\|\Delta\mathbf{x}_{v_j}\|$ with ε_p. If there is at least one pose updated, we call Alg. 1. The reason why we do not call Alg. 4 for pose update is due to the special graph structure for visual SLAM problems. When we update a vertex in the graph, we have to further update all the edges connected to it. For a pose vertex, there are typically hundreds of connected edges that also have to be updated. But for a landmark vertex, there are typically ten or fewer edges that need to be updated. Multiple-rank update becomes less efficient when the number of updates is near or larger than the number of variables, therefore, Alg. 1 is used in pose update instead of Alg. 4.

In landmark update, we first determine how many landmark vertices $V_j \in \mathcal{V}_l$ need to be updated by comparing $\|\Delta\mathbf{x}_{v_j}\|$ with ε_l. We then decide to call Alg. 1, Alg. 4, or exit the iteration according to the ratio $\frac{|\mathcal{V}_{mdf}|}{|\mathcal{V}_l|}$, where $|\cdot|$ represents the cardinality of the set. If the ratio is above a high threshold ε_{up}, it means that most landmark vertices need to be updated so we call Alg. 1 instead of updating them individually. On the other hand, if there is no need to update (ratio=0), we directly exit the iteration and go to next frame. Otherwise, we call Alg. 4 to update those landmark vertices with higher increment. The benefit of this strategy is that we depend on the current condition to efficiently process each iteration rather than blindly reconstruction or update.

It is worth noting that we design our algorithm to sometimes switch back to Alg. 1 because of two reasons. First, Cholesky multiple-rank update/downdate will be slower comparing to solving the problem from scratch if there are too many updates. Second, to partially update the system will cause accumulated error due to approximation. Hence, using Alg. 1 can alleviate this problem.

V. Experiments

The proposed algorithms have been validated in physical experiments by using benchmark datasets.

A. Experiment Setup

For the optimization framework, we utilize g2o [1] as our baseline, and extend it by adding Cholesky update and graph pruning. For the complete visual SLAM system, we exploit ORB-SLAM [13] to run all the datasets. For testing data, we use KITTI datasets [36] to validate the results. We process all the data by using C++ on a desktop PC with an Intel Core i7-4790 CPU at 3.6GHz with 16GB RAM.

We compare our TSS with ORI. To validate our algorithms, we first extract the initial values of estimations (camera poses and landmark positions) with their measurements (landmark positions in pixel coordinates) for each key frame, and then save them as g2o files with vertices and edges before optimization. Finally, we process the g2o files by using ORI and TSS, and compare their performance. Since we focus on comparing the LBA part in visual SLAM problems, the g2o files we only save the LBA part. GBA and loop closure features are turned off because they are not part of the comparison.

It is worth noting that, due to the structure and the sparsity of visual SLAM problems, it is sometimes more efficient to solve $\mathbf{A}\Delta\mathbf{x} = \mathbf{b}$ using the Schur complement method rather than applying a direct solver. According to [1], using the Schur complement method outperforms the direct solver when the landmark number exceeds the pose number, which is usually the case for visual SLAM problem. Therefore, in order to have a fair comparison, we compare our TSS with ORI using Schur complement for runtime in this Section.

B. Accuracy Comparison

We compare the accuracy by using the cost function $F(\mathbf{x})$ values since $F(\mathbf{x})$ values can describe how good the optimization approaches reduce the error. Besides, we provide the initial cost value before performing optimization to show that our TSS can achieve the similar performance to ORI but with less time.

Moreover, we define the cost gain for more detail comparison as follows. Let C_{INI} be the cost function per frame for initial cost, C_{ORI} be the cost function per frame for ORI, and C_{TSS} be the cost function per frame for TSS. We define the cost gain C by $C = (C_{ORI} - C_{TSS})/C_{INI}$. We show the results for cost gain C in Tab. I. According to Tab. I, the accuracy of using TSS slightly decreases, but no more than 6% compared with that of ORI.

C. Runtime Comparison

We compare the runtime for both ORI and TSS. The unit of runtime is millisecond (ms). We measure the average time processed for each frame, and define the time speedup factor for comparison. Let T_{ORI} be the runtime per frame for ORI, and T_{TSS} be the runtime per frame for TSS. We define the time speedup factor S by $S = T_{ORI}/T_{TSS}$.

The time complexity of decomposing a sparse matrix $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T}$ highly depends on the reordering result produced by fill-reducing pivoting methods [33], [34]. These methods mostly use heuristic to solve the NP-hard minimum fill-in problem, which makes complexity analysis difficult without looking into particular problem instances. Therefore, we use experiments to compare solver performance statistically.

The results of runtime comparison are shown in Tab. I. Tab. I indicates that our TSS can cut the runtime in half for the LBA processes in visual SLAM problems, with comparable accuracy.

D. Graph Pruning vs Cholesky Update

We also show the contribution of graph pruning (Alg. 3) and Cholesky update (Alg. 4). We use KITTI00 and KITTI02 to run the experiments. Tab. II illustrates the comparison of speedup factor (S) and cost gain (C) by using graph pruning, Cholesky update, and their combination.

TABLE I

COMPARISON: ORIGINAL SOLVER VS TUNABLE SPARSE SOLVER

		Initial Cost	ORI		TSS		Gain(+/-)	
Dataset	Frames #	Cost (C_{INI})	Time (T_{ORI})	Cost (C_{ORI})	Time (T_{TSS})	Cost (C_{TSS})	Speedup (S)	Cost (C)
KITTI00	473(2-474)	10782.0	66.4	1628.1	33.1	2247.6	2.006x	-5.74%
KITTI00	258(475-732)	11071.7	63.2	1472.7	30.4	1799.9	2.078x	-2.95%
KITTI00	262(733-994)	10514.3	61.7	1477.8	29.9	1858.3	2.063x	-3.61%
KITTI00	247(995-1241)	11413.9	71.8	1830.7	35.0	2404.7	2.051x	-5.02%
KITTI02	1411(2-1412)	9774.8	63.5	1434.9	30.1	1853.6	2.109x	-4.28%
KITTI02	88(1413-1500)	14025.9	62.4	1658.5	30.5	2423.9	2.045x	-5.45%
Avg.	2739	10426	64.5	1518.8	31.0	1985.1	2.075x	-4.47%

We are able to see that both graph pruning and Cholesky update can achieve at around 1.3x speedup in time with approximately 3% loss in accuracy, respectively. And the combination of both techniques can achieve 2.0x speedup in time with less than 5% loss in accuracy on average, which indicates that both algorithms are complementary to each other.

E. Tunable Analysis

We illustrate the tunable capability for our proposed solver in Fig. 3 by using KITTI00 dataset. The tunable capability enables us to sacrifice a little accuracy but gain much speed. Fig. 3 shows the results of speedup versus cost for TSS (red) in different setting and ORI (blue) as the baseline. The different setting depends on how we tune the thresholds: ε_{χ^2}, ε_p, and ε_l.

As ε_{χ^2} becomes high, TSS gains the speed but loss accuracy since the pruned graph is small. On the other hand, when ε_{χ^2} is small, the performance gets close to that of ORI. The same principle applied to ε_p and ε_l as well. The higher ε_p and ε_l, the fewer vertices need to be updated, which means it is faster but less accurate. However, it is worth noting that if we set a low ε_l, then TSS would be slower than ORI because there are too many vertices needing to be updated. This is why we need to design ε_{up} to switch to Alg. 1 instead of Alg. 4. ε_{up} usually depends on the problems when using Cholesky update and downdate. In our experiments, $\varepsilon_{up} = 0.1$.

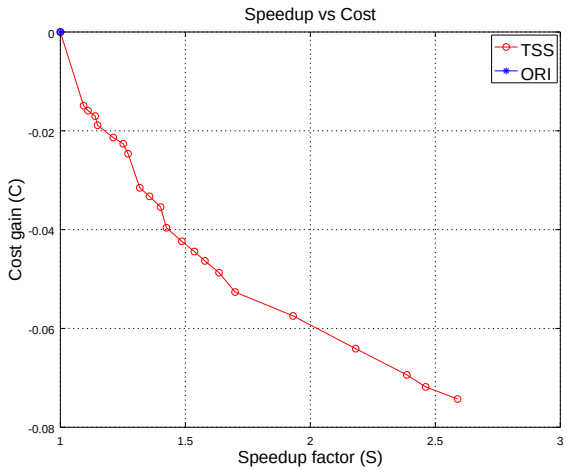


Fig. 3. The illustration of time vs cost for TSS (red) in different setting and ORI (blue) as the baseline. The experiment is done by using KITTI00 from frame 2 to frame 474.

VI. CONCLUSIONS AND FUTURE WORK

We proposed a tunable sparse solver that is able to solve graph optimization problems in visual SLAM problems with faster speed and comparable accuracy comparing with the traditional solver. We designed a graph pruning algorithm to reduce the original optimization problem, so we can focus on those vertices with large errors, and ignore the others with small errors to reduce redundant computation. We also applied modified Cholesky factorization to update the system of linear equations and solve the increments from one iteration of the optimization problem to another in order to reduce the repeated computation. We validated our algorithm in physical experiments and the results show that our tunable sparse solver can cut the runtime in half, compared to the original solver, with a loss of less than 5% in accuracy for the LBA processes in visual SLAM problems.

In the future, we will first improve the tunable sparse solver to by developing better data structure and multi-threaded implementation. Improving parallelization and incorporating graphics process unit (GPU) will help improve algorithm speed.

ACKNOWLEDGMENT

We are grateful to H. Cheng, B. Li, S. Yeh, A. Kingery, and A. Angert for their inputs and contributions to the Networked Robots Laboratory at Texas A&M University.

REFERENCES

[1] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g 2 o: A general framework for graph optimization," in *IEEE International Conference on obotics and Automation (ICRA), Shanghai, China*, 2011, pp. 3607–3613.

[2] S. Thrun and Y. Liu, "Multi-robot SLAM with sparse extended information filers," in *Robotics Research*, ser. Springer Tracts in Advanced Robotics, P. Dario and R. Chatila, Eds. Springer Berlin / Heidelberg, 2005, vol. 15, pp. 254–266.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (SURF)," *Computer vision and image understanding*, vol. 110, no. 3, pp. 346–359, 2008.

TABLE II
COMPARISON: GRAPH PRUNING VS CHOLESKY UPDATE

		Graph Pruning		Cholesky Update		Cholesky Update+Graph Pruning	
Dataset	Frames #	Speedup (S)	Cost (C)	Speedup (S)	Cost (C)	Speedup (S)	Cost (C)
KITTI00	473(2-474)	1.328x	-4.20%	1.320x	-3.44%	2.006x	-5.74%
KITTI00	258(475-732)	1.276x	-2.21%	1.264x	-1.70%	2.078x	-2.95%
KITTI00	262(733-994)	1.285x	-2.77%	1.272x	-1.89%	2.063x	-3.61%
KITTI00	247(995-1241)	1.394x	-3.76%	1.305x	-4.48%	2.051x	-5.02%
KITTI02	1411(2-1412)	1.314x	-3.10%	1.264x	-3.10%	2.109x	-4.28%
KITTI02	88(1413-1500)	1.258x	-3.53%	1.263x	-3.59%	2.045x	-5.45%
Avg.	2739	1.316x	-3.26%	1.278x	-3.06%	2.075x	-4.47%

[5] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[6] H. Strasdat, J. Montiel, and A. J. Davison, "Real-time monocular SLAM: Why filter?" in *IEEE International Conference on obotics and Automation (ICRA), Anchorage, AK*, 2010, pp. 2657–2664.

[7] J. Civera, O. G. Grasa, A. J. Davison, and J. Montiel, "1-point ransac for EKF-based structure from motion," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), St. Louis, MO*, 2009, pp. 3498–3504.

[8] J. Sola, T. Vidal-Calleja, J. Civera, and J. M. M. Montiel, "Impact of landmark parametrization on monocular EKF-SLAM with points and lines," *International journal of computer vision*, vol. 97, no. 3, pp. 339–368, 2012.

[9] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—a modern synthesis," in *International workshop on vision algorithms*. Springer, 1999, pp. 298–372.

[10] H. Strasdat, J. Montiel, and A. J. Davison, "Scale drift-aware large scale monocular SLAM," *Robotics: Science and Systems VI*, vol. 2, 2010.

[11] S. Agarwal, K. Mierle, and Others, "Ceres solver," http://ceres-solver.org.

[12] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM," in *European conference on computer vision*. Springer, 2014, pp. 834–849.

[13] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras," *IEEE Transactions on Robotics*, 2017.

[14] Y. Lu, J. Lee, S.-H. Yeh, H.-M. Cheng, B. Chen, and D. Song, "Sharing heterogeneous spatial knowledge: Map fusion between asynchronous monocular vision and lidar or other prior inputs," in *The International Symposium on Robotics Research (ISRR), Puerto Varas, Chile*, vol. 158, 2017.

[15] C. Chou, A. Kingery, D. Wang, H. Li, and D. Song, "Encoder-camera-ground penetrating radar tri-sensor mapping for surface and subsurface transportation infrastructure inspection," in *IEEE International Conference on obotics and Automation (ICRA), Brisbane, Australia*, May 2018.

[16] E. Mouragnon, M. Lhuillier, M. Dhome, F. Dekeyser, and P. Sayd, "Real time localization and 3D reconstruction," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1, 2006, pp. 363–370.

[17] Y. Lu and D. Song, "Visual navigation using heterogeneous landmarks and unsupervised geometric constraints," in *IEEE Transactions on Robotics (T-RO)*, vol. 31, no. 3, June 2015, pp. 736 —- 749.

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

[19] M. Kaess, A. Ranganathan, and F. Dellaert, "iSAM: Incremental smoothing and mapping," *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1365–1378, 2008.

[20] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "iSAM2: Incremental smoothing and mapping using the bayes tree," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 216–235, 2012.

[21] L. Polok, V. Ila, M. Solony, P. Smrz, and P. Zemcik, "Incremental block Cholesky factorization for nonlinear least squares in robotics." in *Robotics: Science and Systems*, 2013.

[22] V. Ila, L. Polok, M. Solony, and P. Svoboda, "SLAM++ - A highly efficient and temporally scalable incremental SLAM framework," *The International Journal of Robotics Research*, vol. 36, no. 2, pp. 210–230, 2017.

[23] V. Ila, L. Polok, M. Solony, and K. Istenic, "Fast incremental bundle adjustment with covariance recovery," in *IEEE International Conference on 3D Vision (3DV)*, 2017, pp. 175–184.

[24] H. Liu, M. Chen, G. Zhang, H. Bao, and Y. Bao, "ICE-BA: Incremental, consistent and efficient bundle adjustment for visual-inertial SLAM," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1974–1982.

[25] X. Wang, R. Marcotte, G. Ferrer, and E. Olson, "AprilSAM: Real-time smoothing and mapping," in *2018 IEEE International Conference on Robotics and Automation (ICRA), Brisbane, Australia*, 2018, pp. 2486–2493.

[26] T. A. Davis and W. W. Hager, "Modifying a sparse Cholesky factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 606–627, 1999.

[27] ——, "Multiple-rank modifications of a sparse Cholesky factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 4, pp. 997–1013, 2001.

[28] ——, "Row modifications of a sparse Cholesky factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 3, pp. 621–639, 2005.

[29] ——, "Dynamic supernodes in sparse Cholesky update/downdate and triangular solves," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 4, p. 27, 2009.

[30] T. A. Davis, *Direct methods for sparse linear systems*. SIAM, 2006.

[31] Y. Chen, T. A. Davis, W. 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.

[32] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[33] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[34] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, "Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 377–380, 2004. [Online]. Available: https://doi.org/10.1145/1024074.1024080

[35] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms, 3rd Edition*. Wiley, 2006.

[36] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.