Closing a Million-Landmarks Loop

Udo Frese
Universität Bremen, Fachbereich Mathematik und Informatik
SFB/TR 8 Spatial Cognition
email: ufrese@informatik.uni-bremen.de

Lutz Schröder
Universität Bremen, Fachbereich Mathematik und Informatik
email: lschrode@informatik.uni-bremen.de

***Abstract*— We present an improved version of the treemap SLAM algorithm which uses Cholesky factors for representing Gaussians and a Hierarchical Tree Partitioning algorithm derived from the established Kernighan-Lin heuristic for graph bisection. We demonstrate the algorithm's efficiency by mapping a simulated building with 1032271 landmarks. In the end, we close a million-landmarks loop in 21ms, providing an estimate for ≈10000 selected landmarks close to the robot, or in 442ms for computing a full estimate.**

I. Introduction

Simultaneous Localization and Mapping (SLAM) has been a topic of research for almost two decades by now after Smith, Self and Cheeseman first formulated it as an estimation problem [1] for a state vector of landmark positions. While Smith et al. used a full $n \times n$ covariance matrix for n landmarks, many approaches tried to avoid the resulting $O(n^2)$ update time by maintaining only individual covariances for each landmark. Julier and Uhlmann followed this idea with a particularly impressive result. Their algorithm maintains statistically consistent bounds and is so efficient they were able to estimate a 'million beacon map' in real time [2].

Later it has been realized that correlations, i.e. uncertainty information not only for each landmark alone but also for their relative position, are crucial [3]. We have earlier paraphrased this phenomenon as 'certainty of relations despite uncertainty of positions' [4], [5]. It becomes most visible in closing a loop, because the precisely known relative position of adjacent landmarks forces the SLAM algorithm to distribute the error along the loop without introducing a break anywhere.

In the last five years, many researchers have aimed at devising efficient approaches that maintain correlations. Several successful algorithms emerged, among them Relaxation [6], CEKF [7], SEIF [8], FastSLAM [9], Atlas [10], MLR [11], TJTF [12], Olson's stochastic descent algorithm [13], Dellaert's multifrontal-QR approach [14], and treemap [15], [16]. Our contribution was the latter; we feel that it is now time to pick up where Julier and Uhlmann left off and — with the help of algorithmic progress and Moore's law — close a million-landmarks loop.

The material is organized as follows. To make the paper self-contained, we give a brief description of the overall approach of the treemap algorithm, i.e. the geometric and probabilistic meaning of a node and how distributions are passed along the tree, in Sect. II (a broader discussion can be found in [15], [16]). Section III introduces a new Cholesky factor based representation for Gaussians which greatly simplifies the matrix-computation part of treemap and makes it numerically more stable. Section IV presents the new Hierarchical Tree Partitioning (HTP) subalgorithm which optimizes the tree while the robot is moving, in order to reduce future computation time. Compared to the original approach [15], it is simpler and more rigorous. We believe that it is also more dependable, because we derived it from the established Kernighan-Lin (KL) graph partitioning heuristic [17], [18].

We conclude in Sect. VI with the experiment that motivated this article. A simulated robot moves through four 100 story buildings with altogether $n = 1032271$ landmarks, $m = 14463587$ measurements, and $p = 3708301$ robot poses. Treemap processes this data and in the end closes a loop over all four buildings in 442ms (or 21ms for a one-story estimate).

II. Treemap's overall Concept

A. Geometric View

Imagine the robot is in a building that is virtually divided into two parts A and B. Now consider: *If the robot is in part A, what is the information needed about B?* Only few of B's features are involved in observations while the robot is in A. All other features of B are not needed to integrate these observations. So probabilistically speaking, the information needed about B is only the marginal distribution of features of B also observed from A conditioned on observations in B.

The idea can be applied recursively by dividing the building into a binary tree of regions. The marginal distribution for a region can be computed recursively. The marginals for the two subregions are multiplied and features are marginalized out that are not observed from the outside of that larger region anymore. This core computation is the same as employed by TJTF [12]. The key benefit of this approach is that for integrating a measurement, only the region containing the robot and its super-regions need to be updated. All other regions remain unaffected.

B. Bayesian View

The input to treemap are observations assigned to leaves of the tree. They are modeled as distributions $p(X|z_i)$ of the state vector of features X, i.e. of landmark positions and robot poses, given some measurement z_i. At the moment, let us take an abstract probabilistic perspective as to how treemap computes an estimate $\hat{x} = E(X|z)$ from these observations. We will subsequently describe the Gaussian implementation.

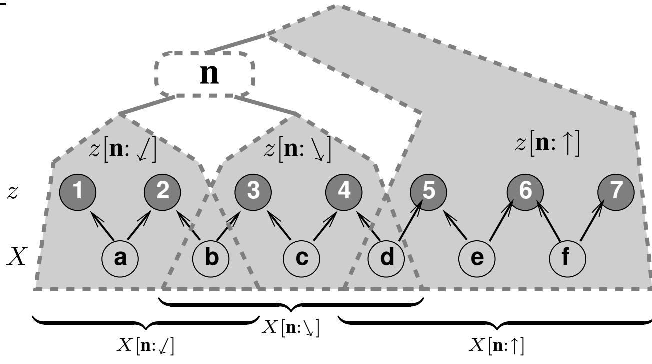

Fig. 1. **Bayesian View.** In this example, observations $z_{1\ldots7}$ provide information about the features $X_{\mathsf{a}\ldots\mathsf{f}}$. The arrows and circles show this probabilistic input as a Bayes net with observed nodes in gray. The dashed outlines illustrate the view of a single node **n**. It divides the tree into three parts, *left-below* $\swarrow$, *right-below* $\searrow$ and *above* $\uparrow$. Hence the observations z are disjointly divided into $z[\mathbf{n}\colon\swarrow] = z_{1\ldots2}$, $z[\mathbf{n}\colon\searrow] = z_{3\ldots4}$ and $z[\mathbf{n}\colon\uparrow] = z_{5\ldots7}$. The corresponding features $x[\mathbf{n}\colon\swarrow] = X_{\mathsf{a}\ldots\mathsf{b}}$, $x[\mathbf{n}\colon\searrow] = X_{\mathsf{b}\ldots\mathsf{d}}$ and $x[\mathbf{n}\colon\uparrow] = X_{\mathsf{d}\ldots\mathsf{f}}$ however overlap ($X[\mathbf{n}\colon\swarrow\searrow] = X_{\mathsf{b}}$, $X[\mathbf{n}\colon\uparrow\searrow] = X_{\mathsf{d}}$). The key insight is that $X[\mathbf{n}\colon\downarrow\uparrow] = X[\mathbf{n}\colon\swarrow\uparrow \vee \searrow\uparrow] = X_{\mathsf{d}}$ separates the observations $z[\mathbf{n}\colon\downarrow]$ and features $X[\mathbf{n}\colon\downarrow\not\uparrow]$ below **n** from the observations $z[\mathbf{n}\colon\uparrow]$ and features $X[\mathbf{n}\colon\not\downarrow\uparrow]$ above **n**, so both are conditionally independent given $X[\mathbf{n}\colon\downarrow\uparrow]$.

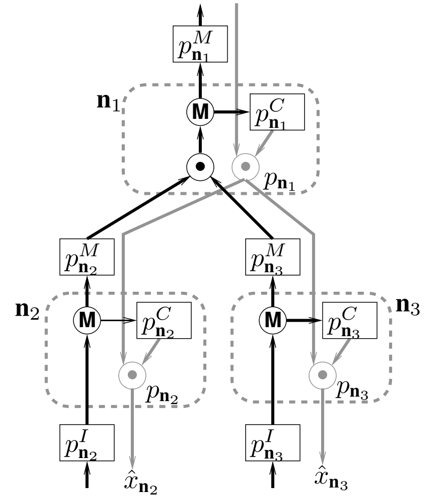

Fig. 2. **Data flow view** of the probabilistic computations performed. The leaves store the input $p^I_{\mathbf{n}}$. During updates (black arrows), a node **n** integrates ($\odot$) the distributions $p^M_{\mathbf{n}_\swarrow}$ and $p^M_{\mathbf{n}_\searrow}$ passed by its children. The result is factorized (Ⓜ) into a marginal $p^M_{\mathbf{n}}$ passed up and a conditional $p^C_{\mathbf{n}}$ stored at **n**. To compute an estimate (gray arrows), each node **n** receives a distribution $p_{\mathbf{n}_\uparrow}$ from its parent, integrates ($\odot$) it with the conditional $p^C_{\mathbf{n}}$, and passes the result $p_{\mathbf{n}}$ down. In the end, estimates $\hat{x}_{\mathbf{n}}$ are available at the leaves.

With respect to the motivating idea, nodes define local regions and super-regions. Formally, however, a node **n** just represents the set of observations assigned to leaves below **n** without any explicit geometric definition.

For a node **n**, the left and right child and the parent are denoted by $\mathbf{n}_\swarrow$, $\mathbf{n}_\searrow$ and $\mathbf{n}_\uparrow$, respectively. We often have to deal with subsets of observations or features according to where they are represented within the tree relative to the node **n** (Fig. 1). Thus, let $z[\mathbf{n}\colon\downarrow]$, $z[\mathbf{n}\colon\swarrow]$, $z[\mathbf{n}\colon\searrow]$, and $z[\mathbf{n}\colon\uparrow]$ denote the observations assigned to leaves *below* ($\downarrow$), *left-below* ($\swarrow$), *right-below* ($\searrow$), and *above* ($\uparrow$) node **n**, respectively. The term *above* **n** refers to *all regions outside the subtree below* ***n*** (!). Analogous expressions $X[\mathbf{n}\colon\ldots]$ denote the features involved in the corresponding observations $z[\mathbf{n}\colon\ldots]$. Note that, while observation sets for different directions $\{\swarrow, \searrow, \uparrow\}$ are disjoint, the corresponding feature sets may overlap because different observations may share a feature. In particular, $X[\mathbf{n}\colon\swarrow\searrow\not\uparrow]$ denotes all features for which **n** is the least common ancestor. They play an important role for **n** because they are marginalized out and finally stored there. As a special case, for a leaf **n**, let $X[\mathbf{n}\colon\swarrow\searrow\not\uparrow]$ denote the features only involved at **n**.

As input, treemap receives the distributions $p^I_{\mathbf{n}} = p\big(X[\mathbf{n}\colon\downarrow]\big|z[\mathbf{n}\colon\downarrow]\big)$ where **n** ranges over all leaves. It is computed from the probabilistic model for the observations $z[\mathbf{n}\colon\downarrow]$ assigned to **n**. The output are the distributions $p_{\mathbf{n}} = p\big(X[\mathbf{n}\colon\downarrow]\big|z\big)$. During the computation, intermediate distributions $p^M_{\mathbf{n}}$ and $p^C_{\mathbf{n}}$ are passed through the tree and stored at the nodes, respectively. In general, $p^I_{\mathbf{n}}$, $p^M_{\mathbf{n}}$, $p^C_{\mathbf{n}}$, and $p_{\mathbf{n}}$ refer to distributions actually computed by treemap, whereas distributions $p\big(X[\ldots]\big|z[\ldots]\big)$ refer to the *abstract* probabilistic input model shown in Fig. 1.

C. Data Flow View (Upwards: Integration)

Figure 2 depicts the data flow that consists of integration ($\odot$) and marginalization (Ⓜ), i.e. multiplying and factorizing probability distributions. Let us follow the probability distributions on their way up through the treemap (Fig. 2, upwards arrows). The data flow starts at a leaf **n** with an input distribution

$$p^I_{\mathbf{n}} = p\big(X[\mathbf{n}\colon\downarrow]\big|z[\mathbf{n}\colon\downarrow]\big), \tag{1}$$

i.e. the distribution of the involved features conditioned on the observations assigned to the leaf **n**. Then features not involved above **n** are marginalized out and the result

$$p^M_{\mathbf{n}} = p\big(X[\mathbf{n}\colon\downarrow\uparrow]\big|z[\mathbf{n}\colon\downarrow]\big) \tag{2}$$

is passed to the parent node. This marginalization is performed in every node, so we will proceed to the discussion of an inner node and explain the details there.

A parent node **n** receives the marginals from both children and multiplies them, resulting in

$$\begin{aligned} p^M_{\mathbf{n}_\swarrow} p^M_{\mathbf{n}_\searrow} &= p\big(X[\mathbf{n}_\swarrow\colon\downarrow\uparrow]\big|z[\mathbf{n}_\swarrow\colon\downarrow]\big)\, p\big(X[\mathbf{n}_\searrow\colon\downarrow\uparrow]\big|z[\mathbf{n}_\searrow\colon\downarrow]\big) \quad (3)\\ &= p\big(X[\mathbf{n}\colon\swarrow\uparrow \vee \swarrow\searrow\not\uparrow]\big|z[\mathbf{n}\colon\swarrow]\big)\, p\big(X[\mathbf{n}\colon\searrow\uparrow \vee \swarrow\searrow\not\uparrow]\big|z[\mathbf{n}\colon\searrow]\big)\\ &= p\big(X[\mathbf{n}\colon\downarrow\uparrow \vee \swarrow\searrow\not\uparrow]\big|z[\mathbf{n}\colon\downarrow]\big). \quad (4)\end{aligned}$$

Let $Y = X[\mathbf{n}\colon\downarrow\uparrow \vee \swarrow\searrow\not\uparrow]$ be the vector of features involved. We divide $Y = \binom{U}{V}$ into the features $U = X[\mathbf{n}\colon\swarrow\searrow\not\uparrow]$ only involved below **n**, for which **n** is the least common ancestor, and those $V = X[\mathbf{n}\colon\downarrow\uparrow]$ also involved above **n**. Next U is marginalized out (Ⓜ) by factorizing $p^M_{\mathbf{n}_\swarrow} p^M_{\mathbf{n}_\searrow}$ into the marginal $p^M_{\mathbf{n}}$ and the corresponding conditional $p^C_{\mathbf{n}}$, with

$$p^M_{\mathbf{n}_\swarrow}(y) \cdot p^M_{\mathbf{n}_\searrow}(y) = p^M_{\mathbf{n}}(v) \cdot p^C_{\mathbf{n}}(u|v), \quad y = \binom{u}{v} \tag{5}$$

$$p^M_{\mathbf{n}} = p\big(X[\mathbf{n}\colon\downarrow\uparrow]\big|z[\mathbf{n}\downarrow]\big) \tag{6}$$

$$p^C_{\mathbf{n}} = p\big(X[\mathbf{n}\colon\swarrow\searrow\not\uparrow]\big|x[\mathbf{n}\colon\downarrow\uparrow], z[\mathbf{n}\downarrow]\big) \tag{7}$$

$$= p\big(X[\mathbf{n}\colon\swarrow\searrow\not\uparrow]\big|x[\mathbf{n}\colon\downarrow\uparrow], z\big). \tag{8}$$

Equation (8) is the formal key point of the overall approach. In Bayes net terminology, $X[\mathbf{n}\colon\downarrow\uparrow]$ separates the observations $Z[\mathbf{n}\colon\downarrow]$ and landmarks $X[\mathbf{n}\colon\downarrow\not\uparrow]$ below $\mathbf{n}$ from the observations $Z[\mathbf{n}\colon\uparrow]$ and landmarks $X[\mathbf{n}\colon\not\downarrow\uparrow]$ above $\mathbf{n}$, as shown in Fig. 1. So $X[\mathbf{n}\colon\not\swarrow\searrow\not\uparrow]$, which is part of $X[\mathbf{n}\colon\downarrow\not\uparrow]$, is conditionally independent from the other observations $Z[\mathbf{n}\colon\uparrow]$ given $X[\mathbf{n}\colon\downarrow\uparrow]$.

The conditional $p_{\mathbf{n}}^C$ is not needed above $\mathbf{n}$ and thus stored at $\mathbf{n}$. The marginal in turn is passed to the parent $\mathbf{n}_\uparrow$ and further processed there. Overall a feature is passed in $p_{\mathbf{n}}^M$ from the leaves where it is involved in $p_{\mathbf{n}}^I$ up to the least common ancestor of all these leaves, where it is part of $X[\mathbf{n}\colon\not\swarrow\searrow\not\uparrow]$. There it is marginalized out and finally stored in $p_{\mathbf{n}}^C$.

For the efficiency of upward integration, the key point is that the $p_{\mathbf{n}}^M$ and $p_{\mathbf{n}}^C$ depend only on the input distributions $p_{\mathbf{n}}^I$ below $\mathbf{n}$. So when adding a new leaf, one essentially needs to update only the nodes from that leaf up.

D. Data Flow View (Downwards: State Recovery)

After the information is integrated into the $p_{\mathbf{n}}^C$, an estimate $\hat{x} = E\big(X\big|z\big)$ is computed by recursively passing distributions downwards (Fig. 2, downward arrows). A node $\mathbf{n}$ receives $p_{\mathbf{n}_\uparrow}$ from its parent $\mathbf{n}_\uparrow$. It computes the marginal $p\big(X[\mathbf{n}\colon\downarrow\uparrow]\big|z\big)$, an implicit step in our Gaussian implementation. The result is then multiplied with the conditional $p_{\mathbf{n}}^C$ stored at $\mathbf{n}$:

$$p_{\mathbf{n}} = p_{\mathbf{n}}^C p_{\mathbf{n}_\uparrow} = p\big(X[\mathbf{n}\colon\not\swarrow\searrow\not\uparrow]\big|x[\mathbf{n}\colon\downarrow\uparrow], z\big)p\big(X[\mathbf{n}\colon\downarrow\uparrow]\big|z\big) \quad (9)$$

$$= p\big(X[\mathbf{n}\colon\downarrow\uparrow \vee \not\swarrow\searrow\not\uparrow]\big|z\big). \quad (10)$$

The product $p_{\mathbf{n}}$ is passed to both children. At the leaves it could be used to compute the estimate as $E\big(x[\mathbf{n}\colon\downarrow]\big|z\big)$. For Gaussians however we can even pass the mean $E\big(X[\mathbf{n}\colon\downarrow\uparrow]\big|z\big)$ directly instead of the whole distribution $p\big(X[\mathbf{n}\colon\downarrow\uparrow]\big|z\big)$.

III. Cholesky Factor Representation of Gaussians

Treemap represents probability distributions as Gaussians, being essentially a least square estimation algorithm. In the original version [15], [16] we used the information form

$$p\big(y\big|...\big) \propto \exp -\tfrac{1}{2}\left(y^T A y + y^T b + \gamma\right) \quad (11)$$

$$= \exp -\tfrac{1}{2}\left(\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)^T \left(\begin{smallmatrix} A & b/2 \\ b^T/2 & \gamma \end{smallmatrix}\right)\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right), \quad (12)$$

where we have combined A, b, and γ in a single matrix using homogeneous coordinates.

As a further improvement, we maintain the Cholesky factor R, with $R^T R = \left(\begin{smallmatrix} A & b/2 \\ b^T/2 & \gamma \end{smallmatrix}\right)$, instead of the matrix itself:

$$\ldots = \exp -\tfrac{1}{2}\left(\left(R\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right)^T\left(R\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right)\right) = \exp -\tfrac{1}{2}\left|R\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right|^2. \quad (13)$$

This representation is easier to implement and numerically more stable since $\mathrm{cond}(RR^T){=}\mathrm{cond}(R)^2{>}\mathrm{cond}(R)$. Traditionally, Cholesky decomposition is defined as LL^T. We write $R^T R$ instead, since we compute $R{=}L^T$ by QR-decomposition.

A. Linearization

In each step incoming observations are linearized in the usual way and stacked. Each component z_i of the observation contributes one row of R. The result is added as a new leaf.

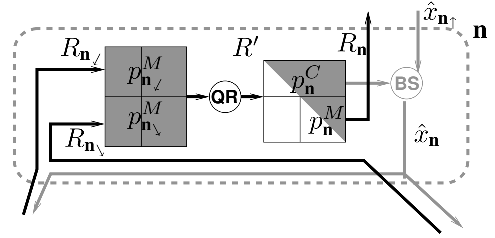

Fig. 3. **Gaussian view.** In the Gaussian implementation, Cholesky factors are passed upwards and means downwards. For upward integration, the passed factors $R_{\mathbf{n}_\swarrow}$ and $R_{\mathbf{n}_\searrow}$ are stacked, implementing $\odot$, and then QR-decomposed (QR) as R', implementing Ⓜ. For downward state recovery, back-substitution (BS) solves the triangular equations defined by $R'_{1\bullet}$, implementing $\odot$.

B. Upwards: Integration

Figure 3 shows how the probabilistic operations at a node $\mathbf{n}$ are realized. The Cholesky factors $R_{\mathbf{n}_\swarrow}$ and $R_{\mathbf{n}_\searrow}$ passed by the children are stacked, implementing the multiplication ($\odot$):

$$\exp -\tfrac{1}{2}\left|R_{\mathbf{n}_\swarrow}\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right|^2 \exp -\tfrac{1}{2}\left|R_{\mathbf{n}_\searrow}\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right|^2$$

$$= \exp -\tfrac{1}{2}\left|\left(\begin{smallmatrix} R_{\mathbf{n}_\swarrow} \\ R_{\mathbf{n}_\searrow} \end{smallmatrix}\right)\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right|^2.$$

While stacking $R_\swarrow$ and $R_\searrow$, their columns are permuted according to the features they represent and grouped such that $X[\mathbf{n}\colon\not\swarrow\searrow\not\uparrow]$ corresponds to the first block of columns and $X[\mathbf{n}\colon\downarrow\uparrow]$ to the second. Next, the result is QR-decomposed as $\left(\begin{smallmatrix} R_\swarrow \\ R_\searrow \end{smallmatrix}\right) = QR'$, replacing it by an equivalent triangular (or trapezoidal) matrix R'. The orthonormal Q is discarded, since

$$\ldots = \exp -\tfrac{1}{2}\left|QR'\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right|^2 = \exp -\tfrac{1}{2}\left|R'\left(\begin{smallmatrix} y \\ 1 \end{smallmatrix}\right)\right|^2. \quad (14)$$

Covariance matrices allow marginalization by taking a submatrix whereas information matrices allow conditioning thereby. Notably Cholesky factors allow both at the same time but not for arbitrary submatrices. With y decomposed as $\left(\begin{smallmatrix} u \\ v \end{smallmatrix}\right)$ and R' accordingly as $\left(\begin{smallmatrix} R'_{11} & R'_{12} \\ 0 & R'_{22} \end{smallmatrix}\right)$, the marginal is easily calculated as

$$p(v) = \int_u \exp -\tfrac{1}{2}\left|\left(\begin{smallmatrix} R'_{11} & R'_{12} \\ 0 & R'_{22} \end{smallmatrix}\right)\left(\begin{smallmatrix} u \\ \left(\begin{smallmatrix} v \\ 1 \end{smallmatrix}\right) \end{smallmatrix}\right)\right|^2 \propto \exp -\tfrac{1}{2}\left|R'_{22}\left(\begin{smallmatrix} v \\ 1 \end{smallmatrix}\right)\right|^2$$

by substituting $u'{=}u + R'^{-1}_{11} R'_{12} v$. The related conditional is

$$p(u|v) = \tfrac{p(u,v)}{p(v)} = \exp -\tfrac{1}{2}\left|\left(\begin{smallmatrix} R'_{11} & R'_{12} \end{smallmatrix}\right)\left(\begin{smallmatrix} u \\ \left(\begin{smallmatrix} v \\ 1 \end{smallmatrix}\right) \end{smallmatrix}\right)\right|^2. \quad (15)$$

So for upward integration, we simply store R at $\mathbf{n}$ and pass its right-lower block R'_{22} to the parent as the marginal $R_{\mathbf{n}}$, implementing Ⓜ.

C. Downwards: State Recovery

For state recovery, we pass the estimate $\hat{x}_{\mathbf{n}} = E\big(X[\mathbf{n}\colon\downarrow\uparrow \vee \not\swarrow\searrow\not\uparrow\big|z\big)$ recursively downwards through the tree. For an estimate, no covariance is required. A node receives $\hat{v} = E(V|z)$ from its parent, where the easiest implementation is to pick it out of a global array of estimates. It then computes

$$\hat{u} = E(U|V = E(V|z), z) \quad (16)$$

$$= \arg\min_u \left|R\left(\begin{smallmatrix} u \\ v \\ 1 \end{smallmatrix}\right)\right|^2 = \arg\min_u \sum_i \left(R_{i\bullet}\left(\begin{smallmatrix} u \\ v \\ 1 \end{smallmatrix}\right)\right)^2. \quad (17)$$

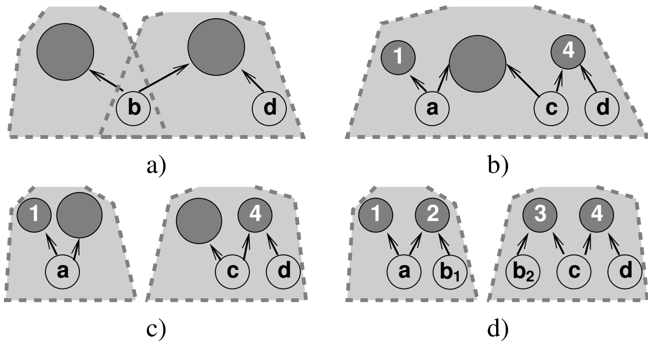

Fig. 4. **Bayesian View.** a) In the example in Fig. 1, feature a and c could be marginalized out without sacrificing further information. b) If b should be marginalized out exactly, both leaves would have to be integrated before. c) Instead, b is marginalized out of both leaves separately. d) This is equivalent to sacrificing the information that the b's in both leaves are the same feature before marginalizing them out. Thus it is a consistent sparsification.

This minimum is obtained one row at a time by back-substitution, since R is triangular (or trapezoidal), thereby implementing $\odot$. We initialize y with $\left(\begin{smallmatrix}0\\v\\1\end{smallmatrix}\right)$ and apply

$$y_i = -\frac{1}{R_{ii}} \sum_{j=i+1}^{\dim y} R_{ij} y_j \quad \text{for } i = \dim u \text{ down to } 1. \tag{18}$$

This is the minimum, since $Ry = R\left(\begin{smallmatrix}u\\v\\1\end{smallmatrix}\right) = \left(\begin{smallmatrix}0\\R_{22}\left(\begin{smallmatrix}v\\1\end{smallmatrix}\right)\end{smallmatrix}\right)$ and the second block-row does not depend on u anyway. The result y is passed to the children, or in practice stored in the global estimate array. To compute a full estimate $\hat{x}$, this is done recursively down the tree. Equation (18) is the key to treemap's efficiency in the downward state recovery phase. For every feature estimated, only a small scalar product is computed.

D. Integration, Marginalization and Sparsification

Initially for each step, one leaf is added to the tree involving the landmarks observed and the current and previous robot pose. This leads to many leaves and old robot poses we are not interested in. Treemap addresses this issue with an operation where two leaves are integrated into one leaf. Again, the input distributions of both leaves are stacked and QR-decomposed. Then features can be marginalized out, and the marginal is the input distribution of the new leaf. The conditional is discarded.

Figure 4 discusses the effect. If a feature is only involved in one leaf, the result is exact marginalization; otherwise information is sacrificed for the sake of keeping the tree sparse. This sparsification is the same as used by TJTF [12] and related to 'cutting the odometry sequence' [4] and 'relocation' [19]. Remarkably it does not introduce overconfidence.

Surprisingly, the matrix arithmetic part of treemap is limited to three simple operations: permuting matrices, QR decomposition (`LAPACK`'s `GEQR2` routine [20]), and back-substitution. The propagation of Gaussians along the tree is exact. The only approximations are linearization, i.e. computing input Gaussians from nonlinear observations, and sparsification.

IV. Hierarchical Tree Partitioning (HTP)

The treemap is built while the robot moves, inserting a new leaf into the tree at every step. The bookkeeping part updates nodes as necessary and computes the estimate using the operations described so far. The efficiency crucially depends on the tree being well balanced and no leaf involving too many features. Thus, a hierarchical tree optimization algorithm runs in parallel and tries to reduce treemap's update cost by moving parts of the tree and by integrating leaves. As discussed in [5], [15], there is no formal guarantee on the tree quality achieved. Outdoor environments often even do not have an efficient tree representation. While in our experience most indoor environments do have one, it is a heuristic assumption that the algorithm finds it. For the discussion here, we formally assume that the HTP algorithm maintains a tree of $O(\log n)$ depth, where a node shares features only with $O(1)$ leaves, and where each node involves $O(k)$ features, where k is the number of features observable from one robot pose. Under this assumption, computation time for upward integration is $O(k^3 \log n)$, and for downward state recovery, $O(kn)$. HTP optimization will then turn out to need time $O(k \log^2 n)$.

A. Bookkeeping

The Gaussians $p_{\mathbf{n}}$ passed along the treemap are accompanied by the information which column of their Cholesky factor corresponds to which feature. It is stored in a sorted array of feature indices and counters. Counters are initialized with 1 at the leaves. When two Gaussians are multiplied ($\odot$), their arrays are merged, adding corresponding counters ($O(k)$). Treemap maintains the total count for each feature in a global array. When at a node **n** a feature's counter reaches its total count, **n** is the least common ancestor of all leaves involving that feature, so it is marginalized out at **n**. A pointer to the node where this happens is also stored in the global array.

A node has two flags indicating whether the Gaussian and the feature array, respectively, are valid. Whenever we change something somewhere in the tree, we reset these flags from there up to the root. Whenever the HTP algorithm needs information on which feature is involved at some node, it recursively updates the invalidated feature arrays. The invalidated Gaussians, however, are recomputed only after the HTP algorithm is finished.

B. Cost function

HTP aims at maintaining a tree that can be updated efficiently. So the most natural cost function to be minimized is the maximal time it can take to recompute the Gaussians from one leaf up to the root. The main computation at node **n** is QR-decomposing a $k \times k$ matrix, where $k = |\mathbf{n}\colon \downarrow\uparrow \vee \swarrow\searrow\Uparrow|$ is the number of features involved at **n**. QR-decomposition is $O(k^3)$, so we calibrated the computation time using a third

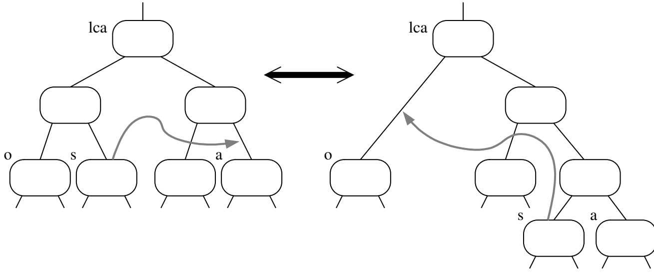

Fig. 5. **Moving a subtree**. In each move, the Kernighan-Lin heuristic greedily moves the subtree below **s** from one side of **lca** to above **a** on the other side of **lca**. The goal is to reduce the worst case cost of **lca**. The sibling of **s** is stored as **o** so the move can be undone by moving **s** back to above **o**.

order polynomial cost(k). Hence a node's cost function is

$$\mathbf{n}.\text{cost} = \text{cost}(|\mathbf{n}\colon \downarrow\uparrow \vee \swarrow\searrow \not\uparrow|), \tag{19}$$

$$\mathbf{m}.\text{worstCost} = \max_{\text{leaf } \mathbf{n} \text{ below } \mathbf{m}} \sum_{\mathbf{n}'=\mathbf{n}}^{\mathbf{m}} \mathbf{n}.\text{cost}, \tag{20}$$

and the overall cost function is **root**.worstCost. We feel that this criterion is more elegant than the one in [15], because it directly reflects the goal of reducing computation time, and provides a sound trade-off between balancing, partitioning, and leaf integration.

Since **root**.worstCost is defined by the worst path to the root, moves in the tree that do not involve this path often do not change **root**.worstCost. Thus for optimization steps in a specific subtree below **lca**, the least common ancestor **lca**.worstCost allows a finer comparison of different moves than **root**.worstCost.

Both **n**.cost and **n**.worstCost are updated together with the feature array of **n**.

C. The Kernighan-Lin Heuristic

HTP is a critical part of the treemap algorithm, actually the only one that could fail. It is also NP-complete, so we decided to adapt the established Kernighan-Lin (KL) heuristic [17] for graph bisection. It is reported to work especially well when applied in a multilevel scheme [18], which we can easily do since we start every step with an existing tree to be improved. Here are the key ideas of KL in treemap terminology (Fig. 5).

1) Consider a single node **lca** to minimize **lca**.worstCost (one bisection problem at a time).
2) Greedily move that subtree **s** from one side of **lca** to the other that minimizes **lca**.worstCost in the next step. (Do this even if **lca**.worstCost increases, since later steps in the run may succeed in reducing the cost.)
3) After a number of unsuccessful moves undo the run.
4) Consider moving **s** to above **a** only if **s** and **a** share a feature.
5) Move every subtree at most once in a run.

D. Overall optimization

Figure 6 shows treemap's activity in each step. A new leaf is added, moved to the optimal place (see below), some

Fig. 6. treemapOneStep ()

Insert new leaf **s** (odometry / landmark observ.) below **root**.	
Invalidate **s** and marginalization nodes of involved features.	
move $=$ optDescend$(\mathbf{s}, \text{true})$	
Execute move permanently; add ancestors to optimiz. queue.	
WHILE Update cost for Gaussians < 3 **root**.worstCost.	
	optimizeByKLRun ()
Recursively update Gaussians (Sec. III-B).	
Recursively compute estimate (Sec. III-C).	

Fig. 7. optimizeByKLRun ()

Fetch node **n** from optimization queue.				
lca = **n**; startCost = **lca**.worstCost; moves $=$ ()				
WHILE $\lvert$moves$\rvert$ $<$ maxMoves				
	move$_1$ $=$ optKLMove$(\mathbf{lca}, \mathbf{lca}_{\swarrow}, \swarrow)$			
	move$_2$ $=$ optKLMove$(\mathbf{lca}, \mathbf{lca}_{\searrow}, \searrow)$			
	move $=$ best of $\{\text{move}_1, \text{move}_2\}$			
	IF	move.$\mathbf{s}_{\uparrow} = \mathbf{lca} \neq$ move.$\mathbf{a}_{\uparrow}$	THEN	**lca** $=$ move.**o**
	IF	move.cost $\geq$ startCost		
	THEN	Execute move preliminarily.		
		Add move to moves.		
		Reset *movable* flag at move.**s**.		
	ELSE	Undo preliminary moves and set *movable* flag.		
		Do moves permanently.		
		Add nodes on path to optimization queue.		
		Add **n** to end of optimization queue.		
		return.		
Undo preliminary moves and set *movable* flag.				
Set *optimal* flag for **n**.				
Try to sparsify out robot poses in $X[\mathbf{n}\colon \swarrow\searrow\uparrow]$.				

optimization runs are performed, and the estimate is computed. HTP can have rather irregular computation time. To reduce this effect, we watch the total update cost for Gaussians that became invalid after moving subtrees. When it exceeds 3 times the worst case update cost, we stop HTP.

For a single run (Fig. 7), we fetch a node **lca** from a queue of nodes to optimize. When a new leaf is inserted, all ancestors are added to this queue, and when a node is moved, all nodes on the unique path are also added. We maintain an *optimal* flag that is reset to avoid adding a node twice.

Then we repeatedly find and execute the best move following 2) until **lca**.worstCost is reduced or maxMoves have been reached. All moves are preliminary at first, i.e. we do not invalidate the Gaussians, to avoid recomputing them if everything is undone later. A special case is moving a child of **lca** (Fig. 5). Then **lca** is moved along with the child, and the child's sibling takes **lca**'s role as least common ancestor.

E. Search for the optimal move

The optimal move for a given **lca** is found by two nested recursions. The outer one recursively traverses all possible

Fig. 8. optKLMove $(\mathbf{lca}, \mathbf{s}, \text{side})$

IF	$\mathbf{s}$.features $\cap$ $\mathbf{lca}_{\neg\text{side}}$.features $= \emptyset$	THEN	return.
IF	$\mathbf{s}$ is flagged *movable*		
THEN	move.$\mathbf{s} = \mathbf{s}$; move.$\mathbf{o}$ = sibling of $\mathbf{s}$		
	IF	$\mathbf{s}_{\uparrow} \neq \mathbf{lca}$	
	THEN	Move $\mathbf{s}$ preliminarily to above $\mathbf{lca}_{\neg\text{side}}$.	
		move$_1$ = optDescend$(\mathbf{s}, \text{true})$	
		move$_1$.cost = max(move$_1$.cost, $\mathbf{lca}_{\text{side}}$.worstCost) + $\mathbf{lca}$.cost	
		Move $\mathbf{s}$ back to above move$_1$.$\mathbf{o}$.	
	ELSE	move$_1$ = optDescend$(\mathbf{s}, \text{false})$	
Recursively update feature array for $\mathbf{lca}$.			
IF	$\mathbf{s}$ is no leaf		
THEN	move$_2$ = optKLMove$(\mathbf{lca}, \mathbf{s}_{\swarrow}, \text{side})$		
	move$_3$ = optKLMove$(\mathbf{lca}, \mathbf{s}_{\searrow}, \text{side})$		
Return best of $\{\text{move}_1, \text{move}_2, \text{move}_3\}$.			

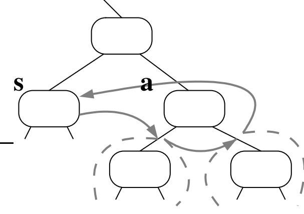

Fig. 9. **Optimal Descend**. The subalgorithm in Fig. 10 searches for the optimal place to move **s**. It considers the initial place directly above **a**, moves **s** to above $\mathbf{a}_{\swarrow}$ (gray arrow) and performs recursion (dashed), moves **s** to above $\mathbf{a}_{\searrow}$ and performs recursion, and finally moves **s** back to where it started.

candidates for **s**. This subalgorithm optKLMove (Fig. 8) considers only one side of **lca** (side=$\swarrow$ or $\searrow$). According to 4), the search is restricted to nodes that share a feature with the other side. Such a node also shares a feature with $\mathbf{lca}_{\neg\text{side}}$, i.e. **lca**'s child on the other side. So we stop the recursion if this is not the case. This greatly reduces the search space, because under the assumptions discussed, only $O(1)$ leaves and $O(\log n)$ nodes share a feature with a fixed node.

As a strategy to simplify the implementation, we do not compute what would happen if we moved a subtree — instead we move it, see what happens, and move it back. Hence every **s** considered in the outer recursion is moved to above $\mathbf{lca}_{\neg\text{side}}$ on the other side, and the inner recursion finds the best place to put it by recursively moving **s** through the tree.

Figure 9 illustrates the inner recursion performed by the subalgorithm in Fig. 10. It starts in the situation where **s** is already moved to directly above a node **a**, and returns the best place to put **s** — there or below — minimizing the worst case cost of $\mathbf{s}_{\uparrow} = \mathbf{a}_{\uparrow}$. It considers three options: Let **s** stay where it is, move it to somewhere below $\mathbf{a}_{\swarrow}$, or move it to somewhere below $\mathbf{a}_{\searrow}$. The last two are checked by moving **s** to above $\mathbf{a}_{\swarrow}$ (or $\mathbf{a}_{\searrow}$, resp.), recursion, and adding **a**.cost, the cost of **a** itself.

The algorithm shown in Fig. 10 solves two subtle problems: First, when **s** is moved to below **a**, **a** takes over $\mathbf{s}_{\uparrow} = \mathbf{a}_{\uparrow}$'s role as minimization target. Second, if the place where **s** is finally moved to does not affect **lca**'s worst case path, it may actually not change **s**.worstCost. For two different options where to move **s**, the algorithm implicitly considers the worst case cost at the least common ancestor of both. So even if both lead to the same **lca**.worstCost because both do not affect **lca**'s worst case path, the algorithm chooses the one that leads to a smaller cost somewhere below **lca**.

Additionally, we use a branch-and-bound optimization not shown in the pseudocode: Throughout the recursion, we maintain the largest $\mathbf{s}_{\uparrow}$.worstCost that would lead to a better move than the best found so far. We stop the recursion if this quantity

Fig. 10. optDescend $(\mathbf{s}, \text{mayStayHere})$

$\mathbf{a}$ = sibling of $\mathbf{s}$; move$_{1,2,3,4}$.$\mathbf{s} = \mathbf{s}$			
Update feature array of $\mathbf{s}_{\uparrow}$.			
IF	$\mathbf{s}$.features $\cap$ $\mathbf{a}$.features $= \emptyset$	THEN	return
IF	mayStayHere		
THEN	move$_1$.$\mathbf{a} = \mathbf{a}$; move$_1$.cost = $\mathbf{s}_{\uparrow}$.worstCost		
IF	$\mathbf{s} \neq \mathbf{root} \wedge \mathbf{s}$ is leaf $\wedge\, \mathbf{a}$ is leaf		
THEN	move$_2$.cost = $\frac{3}{2}$ cost after joining $\mathbf{s}$ and $\mathbf{a}$		
	IF	move$_2$.cost leads to $<$ startCost at $\mathbf{lca}$	
	THEN	move$_2$.$\mathbf{a} = \mathbf{a}$; move$_2$.integrate = true.	
IF	$\mathbf{a}$ is leaf	THEN	return best of $\{\text{move}_1, \text{move}_2\}$
Move $\mathbf{s}$ to above $\mathbf{a}_{\swarrow}$ preliminarily; update $\mathbf{a}$'s feature array.			
move$_3$ = optDescend$(\mathbf{s}, \text{true})$			
move$_3$.cost = $\max\{\text{move}_3.\text{cost}, \mathbf{a}_{\searrow}.\text{worstCost}\} + \mathbf{a}.\text{cost}$			
Move $\mathbf{s}$ to above $\mathbf{a}_{\searrow}$ preliminarily; update $\mathbf{a}$'s feature array.			
move$_4$ = optDescend$(\mathbf{s}, \text{true})$			
move$_4$.cost = $\max\{\text{move}_4.\text{cost}, \mathbf{a}_{\swarrow}.\text{worstCost}\} + \mathbf{a}.\text{cost}$			
Move $\mathbf{s}$ to above $\mathbf{a}$ preliminarily; update $\mathbf{a}$'s feature array.			
return best of $\{\text{move}_1, \text{move}_2, \text{move}_3, \text{move}_4\}$			

exceeds $\mathbf{s}.\text{worstCost} + \text{cost}(|\mathbf{s}: \downarrow\uparrow|)$.

Under the assumptions discussed, there are $O(\log n)$ candidates for **s** and $O(\log n)$ for **a**, and the inner recursion recomputes the feature array of one node ($O(k)$) in each step. So finding the optimal move for **lca** takes time $O(k \log^2 n)$.

F. Integration, Marginalization, and Sparsification

Apart from moving subtrees, leaves must be integrated in order to make the tree more compact and to marginalize out old robot poses. This task is part of the inner recursion (Fig. 10, line 5). When considering moving **s** to above **a** and both are leaves, it is also considered to integrate them into one leaf. Moving **s** and integrating it with **a** are treated as one step together. Otherwise, a higher cost before joining **s** and **a** may prevent **s** from moving there. Integration cannot be undone, so we only integrate if this leads to a successful KL-run and, as a heuristic, only if the cost is reduced by $\frac{1}{3}$.

Whenever two leaves are joined, old robot poses that are only involved in the resulting leaf are marginalized out.

Sparsification is performed in the same way, by marginalizing an old robot pose out of a leaf even if it is still involved in another leaf. However since information is lost thereby, we

treat sparsification as a 'last resort'. Hence we sparsify out an old robot pose only if the least common ancestor of all occurrences of that pose is flagged *optimal* (Fig. 7, last line), because then we do not expect any more to marginalize it out without information loss. In any case, a pose is only sparsified out if the involved leaves share at least two landmarks, because otherwise the map may disintegrate into unconnected pieces.

V. Comparison with Related Algorithms

We now briefly compare treemap to some related algorithms; for a general overview and discussion, cf. [5], [21]. Treemap is closely related to the Thin Junction Tree Filter (TJTF) by Paskin [12], as both pass distributions along a tree. However, treemap stores the input distributions at leaves only and thus can optimize how the leaves are combined into a tree, whereas TJTF stores input distributions at all nodes. Further, treemap uses different representations for passing up and down, thus reducing computation time in the recursive downward pass, whereas TJTF uses the information form for both.

The Sparse Extended Information Filter (SEIF) [8] and treemap both exploit sparsity in the information form, since the product of all $p_{\mathbf{n}}^I$ is a sparse Gaussian. However, the tree on top allows treemap to exactly recover the mean, where SEIF must rely on slowly converging relaxation. On the other hand, treemap's topological restrictions are more severe than SEIF's.

Dellaert et al. propose using a sparse-QR factorization [14]. This is related to treemap, since all $R_{\mathbf{n}}$ together form a sparse QR factorization. However, treemap can incrementally update the factorization, whereas Dellaert's approach always recomputes from scratch. Further, treemap's tree is more than a sparse matrix. Since each leaf is a probability distribution on its own, treemap can consistently sparsify.

VI. Experiments

A. Simulated Setting

We simulate the robot moving through four 100-story skyscrapers (Fig. 11a,b), where the identical stories are taken from our university building (Universität Bremen, MZH, level 3). The robot uses elevators, modeled as pure vertical motion, to go from story to story. Still the mapping is 2D, the stories are simply counted. The robot starts in the middle of the map and consecutively maps the left-upper, right-upper, right-lower and left-lower building, connected on the ground level by corridors. Then it closes a huge loop through all four buildings.

The simulator provides a landmarks' identity, so we do not address data-association in this work. The simulated robot has an odometry error of $5\frac{\text{mm}}{\sqrt{\text{m}}}$, with 0.3m radius, a 180° field of view, 3m sensor range, 1° angular error and 1% distance error. All experiments use an AMD Athlon 64 2.2Ghz.

B. Optimized Implementation

A million landmarks make an extremely large map, so we optimized the implementation to an extent not necessary for normal-sized maps. After computing the QR-decomposition in double-precision, we store the upper triangle bottom to top, right to left in single-precision. This improves storage space and cache performance for back-substitution.

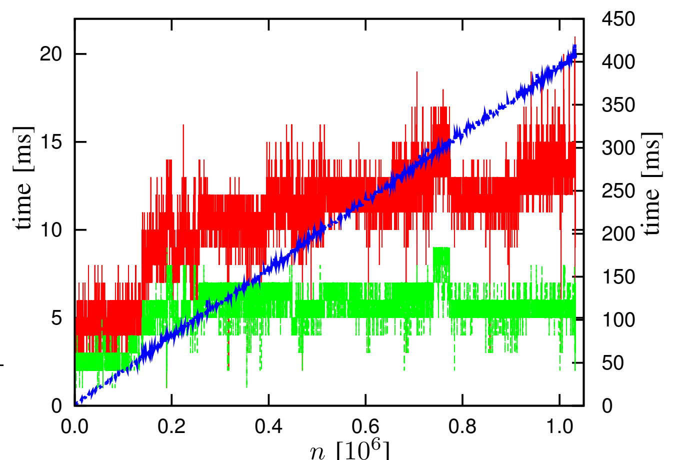

Fig. 12. **Computation time.** Time per step (top plot, red) for a 1-story estimate, bookkeeping time (bottom plot, green), and additional time for a full estimate of all stories (straight plot, blue, right y-axis).

Still, computing an estimate for a million landmarks needs 442ms time, and with 3.5 million steps the whole experiment would take $\approx$ 9 days. To reduce time we normally compute only an estimate for the landmarks in the current story, so each step needs $<$ 21ms, and the whole experiment takes $\approx$ 10h. With the given map scale this corresponds to real time mapping with 10m/s and observations every 25cm.

C. Results

Overall, the map encompasses $n = 1032271$ landmarks, $m = 14463587$ measurements, and $p = 3708301$ robot poses. This highlights the importance of marginalizing out old robot poses, since otherwise the estimation problem would have $2n + 3p = 13189445$ dimensions instead of $2n = 2064542$. Treemap succeeded in marginalizing out 3373643 poses without loss of information and sparsified out another 285968 poses. It kept 48690 poses, which is a negligible overhead compared to the landmarks. The worst case cost of the root node grew from 1.415ms for $n=10000$ to 5.55ms in the end. This shows that the HTP subalgorithm succeeded in maintaining a suitable tree.

Figure 11c shows the ground floor of the estimated map immediately before closing the huge loop over all four buildings. When the loop is closed all four buildings move in the map and most landmarks significantly change their estimate. Thereafter, the robot closes two further loops connecting the center to the top and bottom corridor (Fig. 11d).

The figure shows the ground-level estimate. Note that the result is the same regardless whether all estimates are computed or just one story. This is in contrast to SEIF, relaxation, and multilevel relaxation, where updating only one story would implicitly condition on all other stories. This would fix the ground story and hence obstruct the closing of the loop.

Figure 12 shows the computation time needed. Computation time for a one-story estimate is always below 21ms, and

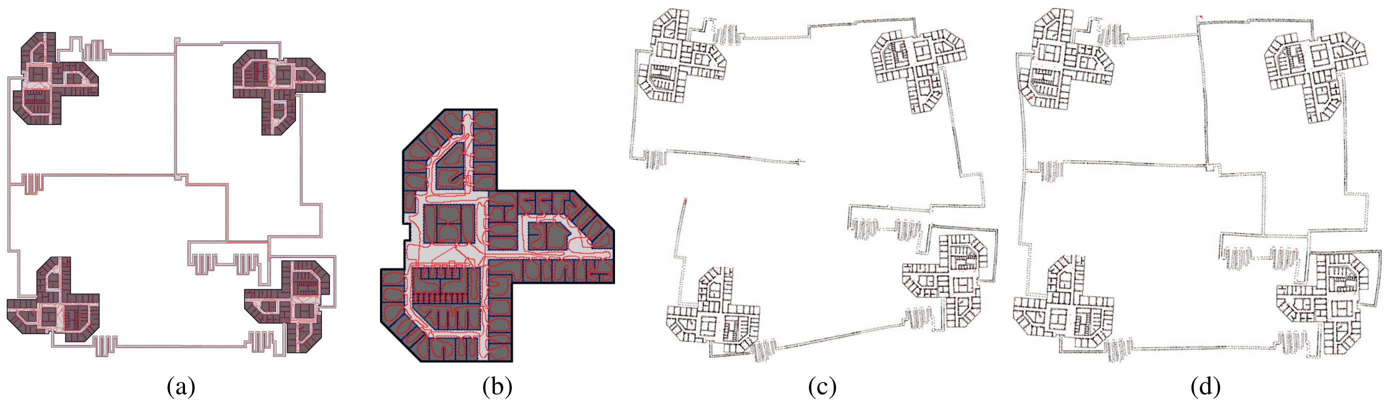

Fig. 11. **Maps.** a) True map (350m × 350m) used by the simulation with 99 stories on top, which are identical except for the connecting corridors. b) Closeup. The red line shows the robot's trajectory, small blue dots are landmarks. c) Map estimate before finally closing the loop. d) Map estimate after closing the loop. A 3D animation of the growing map can be downloaded from the authors' web site `www.informatik.uni-bremen.de/~ufrese/`.

usually even smaller. About half of the time is spent in HTP bookkeeping, the rest is for updating Gaussians and state recovery. The additional time for an estimate of all landmarks rises linearly to 421ms, which is still close to realtime for practical experiments. Since the tree needed 1.04GB storage space the limiting factor was memory bandwidth (2.1GB/s).

VII. Conclusion

We have presented an improved treemap algorithm that can update a map with $n{=}1032271$ landmarks in real time. It closes a million-landmarks loop in 21ms, providing an estimate for 10000 landmarks, or in 442ms for a full estimate.

In a scenario like in our experiment, this means that the SLAM algorithm is no longer a limiting factor for map size — likely, a physical robot would fail before actually mapping the entire path. We thus have an algorithm that not only performs well with map sizes typical for present-day applications, but is also well-prepared to cope with future increases in map size.

Treemap is very general since it manipulates plain Gaussians. So our next goal is an open source implementation where the application programmer can choose the model (2D or 3D; landmarks and / or poses) and define the linearization and sparsification policy. We also want to compare the map precision depending on this policy with other algorithms.

References

[1] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," in *Autonomous Robot Vehicles*, I. Cox and G. Wilfong, Eds. Springer Verlag, New York, 1988, pp. 167 – 193.

[2] S. Julier and J. Uhlmann, "Building a million beacon map," *Proceedings of SPIE: Sensor Fusion and Decentralized Control in Robotic Systems IV*, vol. 4571, 2001.

[3] J. Castellanos, J. Tardós, and G. Schmidt, "Building a global map of the environment of a mobile robot: The importance of correlation," in *Proceedings of the IEEE International Conference on Robotics and Automation, Albuquerque*, 1997, pp. 1053 – 1059.

[4] U. Frese and G. Hirzinger, "Simultaneous localization and mapping - a discussion," in *Proceedings of the IJCAI Workshop on Reasoning with Uncertainty in Robotics, Seattle*, Aug. 2001, pp. 17 – 26.

[5] U. Frese, "A discussion of simultaneous localization and mapping," *Autonomous Robots*, vol. 20, no. 1, pp. 25–42, 2006.

[6] T. Duckett, S. Marsland, and J. Shapiro, "Learning globally consistent maps by relaxation," in *Proceedings of the IEEE International Conference on Robotics and Automation, San Francisco*, 2000, pp. 3841–3846.

[7] J. Guivant and E. Nebot, "Solving computational and memory requirements of feature-based simultaneous localization and mapping algorithms," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 4, pp. 749–755, 2003.

[8] S. Thrun, Y. Liu, D. Koller, A. Ng, Z. Ghahramani, and H. Durrant-Whyte, "Simultaneous localization and mapping with sparse extended information filters," *International Journal of Robotics Research*, vol. 23, no. 7–8, pp. 613–716, 2004.

[9] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," in *Proceedings of the AAAI National Conference on Artificial Intelligence, Edmonton*, 2002, pp. 593–598.

[10] M. Bosse, P. Newman, J. Leonard, and S. Teller, "SLAM in large-scale cyclic environments using the Atlas framework," *International Journal on Robotics Research*, vol. 23, no. 12, pp. 1113–1140, 2004.

[11] U. Frese, P. Larsson, and T. Duckett, "A multigrid algorithm for simultaneous localization and mapping," *IEEE Transactions on Robotics*, vol. 21, no. 2, pp. 1–12, 2004.

[12] M. Paskin, "Thin junction tree filters for simultaneous localization and mapping," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, San Francisco, 2003, pp. 1157–1164.

[13] E. Olson, J. Leonard, and S. Teller, "Fast iterative alignment of pose graphs with poor initial estimates," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2006, to appear.

[14] F. Dellaert, A. Kipp, and P. Krauthausen, "A multifrontal qr factorization approach to distributed inference applied to multi-robot localization and mapping," in *Proceedings of the American Association for Artificial Intelligence*, 2005.

[15] U. Frese, "An $O(\log n)$ algorithm for simulateneous localization and mapping of mobile robots in indoor environments," Ph.D. dissertation, University of Erlangen-Nürnberg, 2004.

[16] ——, "Treemap: An $O(\log n)$ algorithm for indoor simultaneous localization and mapping," *Autonomous Robots*, 2006, to appear.

[17] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th ACM/IEEE Design Automation Conference, Las Vegas*, June 1982, pp. 175–181.

[18] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the ACM International Conference on Supercomputing, Sorrento*, 1995, pp. 626–657.

[19] M. Walter, R. Eustice, and J. Leonard, "A provably consistent method for imposing exact sparsity in feature-based slam information filters," in *Proceedings of the 12th International Symposium of Robotics Research*, 2005.

[20] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Society for Industrial and Applied Mathematics, 1999.

[21] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2005.