2012 International Conference on Indoor Positioning and Indoor Navigation, 13-15th November 2012

Complexity-reduced FootSLAM for Indoor Pedestrian Navigation

Achieving t log t Computational Complexity for Efficient Real-Time and Mass Market Usage

Maria Garcia Puyol*
Email: Maria.GarciaPuyol@dlr.de

Patrick Robertson*
Email: Patrick.Robertson@dlr.de

Oliver Heirich*
Email: Oliver.Heirich@dlr.de

*Institute of Communications and Navigation
German Aerospace Center (DLR)
Wessling, Germany

***Abstract*—FootSLAM or simultaneous localization and mapping (SLAM) for pedestrians is a technique that addresses the indoor positioning and mapping problem based on human odometry (aka pedestrian dead reckoning), e.g. with a foot-mounted inertial sensor. FootSLAM follows the FastSLAM factorization, using a Rao-Blackwellized particle filter to simultaneously estimate the building layout and the pedestrian's pose - his position and orientation. To that end, FootSLAM divides the 2D space into a grid of uniform and adjacent hexagons and counts the number of times each particle crosses the edges of the hexagons it visits. As we shall show, the complexity of FootSLAM grows quadratically with time, preventing the mapping of large areas.**

In this paper we present a new geographic tree-based data structure, called H-tree, to reduce the quadratic-in-time computational growth rate of naive FootSLAM to t times log t. In addition, we introduce a compact representation (alphabet) for the set of six counters that are used to map the transitions of the particles across the edges of each hexagon. This alphabet is particularly effective during the exploration phases of FootSLAM that requires much particle diversity.

In this contribution, the computational savings of the H-tree are presented both theoretically and with real-world data. In practice, we believe that FootSLAM can be applied in quasi real-time applications that require rapid mapping of unknown areas. Additionally, the mass market offline mapping process can be undertaken much more efficiently.

***Keywords*—Indoor navigation, real-time pedestrian localization and mapping, FootSLAM, FastSLAM.**

I. Introduction

Pedestrian navigation has attracted significant research interest over the past few years. Pedestrian navigation refers to the process by which the position of a pedestrian can be determined over time. Whereas outdoor pedestrian navigation relies on the use of satellite signals and GNSS-aided automated mapping techniques for the determination of the pedestrian's position (e.g. GPS receiver), indoor environments forbid the use of satellite signals due to distorting effects such as multipath propagation.

To meet the challenges of accurately determining the position of a pedestrian in indoor environments, one of the addressed topics has been the use of low-cost sensors that the pedestrian wears or carries and that measures his steps while walking. One of the possibilities is the use of MEMS-based inertial sensors (IMUs). Nevertheless, it is well-known that the performance of these IMUs is prone to unbounded drifting errors over time despite the use of zero velocity updates at every step [1]. However, when the constraints that limit the pedestrian's motion are known - i.e. a map, it has been shown that these errors can be limited and that IMUs can be used to obtain stable positioning in two and three dimensions [2][3][4]. Nonetheless, maps are often unavailable, proprietary, obsolete, incomplete, outdated or do not represent other non-wall features of the environment - e.g. pieces of furniture, stalls, etc. - that conduct the motion of the pedestrian as much as walls do.

FootSLAM or simultaneous localization and mapping (SLAM) for pedestrians was recently presented to address this [5]. Human measurements are used by a Rao-Blackwellized Particle Filter (RBPF) to generate a map of the environment while estimating the pose of the pedestrian - his position and orientation - within the map. To map the environment, the 2D space has been divided into a grid of hexagons where the particles map their transitions across the edges of the hexagons. FootSLAM has already shown its SLAM capabilities at different building environments [5][6][7]. However, as we shall show, the computational requirements of FootSLAM increase strongly with the covered area, showing a linear growth **per time step**.

In this paper we present a new geographic tree-based data structure, called H-tree, to reduce this linear growth rate per time step to logarithmic, thus allowing quasi real-time processing. Furthermore, an alphabet of symbols has been introduced to compactly represent the six transition counters that store the number of times that each particle has crossed the edges of a hexagon.

This paper is organized as follows: Section II introduces the current particle filter implementation for FootSLAM. Section III analyzes the complexity of a naive FootSLAM implementation. Section IV presents the H-tree and its impact on the complexity of FootSLAM and Section V explains the alphabet of symbols. Next, Section VI describes a possible approach to store the particles' diversity of hypotheses when mapping large areas. Finally, experimental results and conclusions are presented in Sections VII and VIII respectively.

II. FootSLAM's Particle Filter Implementation

FootSLAM uses a particle filter to estimate the pedestrian's poses and the map layout. The basic idea of particle filters is to represent the desired posterior density of the state by a set of N_p weighted samples, called particles [8]. FootSLAM builds on the FastSLAM algorithm [9] that follows the Rao-Blackwellized factorization. Rao-Blackwellized particle filters (RBPF) use particles to represent the posterior over some of the state variables along with some parametric posterior density function (in visual SLAM usually Gaussians) to represent the other variables [9] .

FootSLAM represents human motion as a first order Markov process: The next step of the pedestrian depends only on his current location. In order to obtain the probability distributions of human motion as a function of location, FootSLAM partitions the 2D space into a grid of N_h adjacent regular hexagons. The particles register their transitions across the edges of the hexagons they visit in the form of transition counts, C.

The main goal of FootSLAM is the computation of the following full posterior [5]:

$$p(\{\mathbf{PUE}\}_{0:k}, \mathbf{M}|\mathbf{Z}^U_{1:k}), \tag{1}$$

where, at time step k, $\mathbf{P}_k$ refers to the pose of the pedestrian, $\mathbf{U}_k$ is the true step vector, $\mathbf{Z}^U_k$ is the step measurement subject to correlated errors $\mathbf{E}_k$ as well as white noise, and $\mathbf{M}$ represents the time invariant map of the environment where the pedestrian walks.

Following the FastSLAM factorization [9] the FootSLAM problem can be decomposed into a pedestrian localization problem and a mapping problem conditioned on the pedestrian's poses [5]. As a consequence, the full posterior can be simplified:

$$p(\{\mathbf{PUE}\}_{0:k}, \mathbf{M}|\mathbf{Z}^U_{1:k}) = \underbrace{p(\mathbf{M}|\mathbf{P}_{0:k})}_{\text{map estimation}} \cdot \underbrace{p(\{\mathbf{PUE}\}_{0:k}|\mathbf{Z}^U_{1:k})}_{\text{pose estimation}}. \tag{2}$$

Furthermore, the division of the space into independent hexagons makes possible the decomposition of the mapping problem into map estimation subproblems [5], one for each hexagon, resulting in:

$$p(\mathbf{M}|\mathbf{P}_{0:k}) = \prod_{h \in N_h} p(\mathbf{M}_h|\mathbf{P}_{0:k}), \tag{3}$$

where $\mathbf{M}_h$ represents the local map of hexagon h, that is, a vector of 6 random variables denoting the probability of transitions across each edge $e \in \{0 \cdots 5\}$ of the hexagon h:

$$\mathbf{M}^{e(\mathbf{U}_k)}_{h(\mathbf{P}_{k-1})} = p(\mathbf{P}_k \in h'|\mathbf{P}_{k-1} \in h). \tag{4}$$

Thereupon, $\mathbf{M}^{e(\mathbf{U}_k)}_{h(\mathbf{P}_{k-1})}$ represents the probability of crossing edge e when leaving hexagon h to go into hexagon h' by adding step vector $\mathbf{U}_k$ to $\mathbf{P}_{k-1}$.

Therefore, the full posterior can be divided into $(1 + N_h)$ estimators, one for the computation over the pedestrian's poses, step vectors and correlated errors and N_h estimators for the computation of the posterior over the map.

A. Particle's Map and Cumulative Map

In FootSLAM a RBPF is applied to estimate the pedestrian's history of poses, step vectors and errors. Each particle i in the RBPF represents $\{\{\mathbf{PUE}\}^i_k, p(\mathbf{M}|\mathbf{P}^i_{0:k})\}$.

But for the particles to effectively represent $p(\mathbf{M}|\mathbf{P}^i_{0:k})$, each particle must possess its own data structure to store the associated map estimates corresponding to the pedestrian's history of poses $\mathbf{P}^i_{0:k}$. This structure is referred to as *particle's map* or, in short, map.

The local map $\mathbf{M}_h$ corresponding to hexagon h is estimated using a data structure that we have called *HTC* (hexagon transition counters). An *HTC* consists of a vector of length six. Each vector element is used to register the transitions C^e_h across an edge. The associated *HTC* to hexagon h that has been visited by particle i is:

$$\{M_h\}^i = \{C^0_h, C^1_h, C^2_h, C^3_h, C^4_h, C^5_h\}^i.$$

Note that when written in bold face, $\mathbf{M}_h$ denotes the unknown random variable that we wish to estimate: The transition probabilities across the edges of a hexagon. On the other hand, M_h represents an *HTC*, i.e. the vector of 6 hexagon transition counters from which the probabilities can be inferred [5].

Each particle's map, $\{M\}^i$, is composed of the set of visited hexagons N^i_{vis} with their corresponding *HTC*s, i.e.:

$$\{M\}^i = \{h, \{M_h\}^i\}; \, \forall h \in N^i_{vis}.$$

Our naive FootSLAM implementation - from now on referred to as naive FootSLAM - builds this map using a lookup table where each entry corresponds to a visited hexagon and its *HTC*. As we shall demonstrate, the choice for the map structure will have a significant impact on the performance of FootSLAM.

In addition, the hexagon transition estimates contained in the N_p particles' maps can be compressed in the form of a *cumulative map*. This *cumulative map* $\{M\}^{cum}$ contains the hexagons visited by *all* particles with their corresponding *cumulative HTC*s, $\{M_h\}^{cum}$:

$$\{M\}^{cum} = \{h, \{M_h\}^{cum}\}; \, \forall h \in N^{cum}_{vis},$$

where $N^{cum}_{vis} = \cup N^i_{vis}$; $\forall i \in N_p$.

The *cumulative HTC* can be computed using the weights of each particle w^i:

$$\{M_h\}^{cum} = \{C^0_h, C^1_h, C^2_h, C^3_h, C^4_h, C^5_h\}^{cum}, \tag{5}$$

where

$$\{C^e_h\}^{cum} = \sum_{i \in N_p} \{C^e_h\}^i \cdot w^i. \tag{6}$$

B. RBPF Algorithm in FootSLAM

The goal of this paper is to analyze the complexity of naive FootSLAM and to reduce it. Thus, a summary of the RBPF algorithm is recalled here (see Section III.A in [5] for a more detailed version):

1) Initialize all N_p particles pose and draw initial error states from a proposed initial distribution.

2) For each time stamp k:
   a) Draw particles from a proposal density.
   b) For all particles, compute the new pose of the pedestrian $\mathbf{P}_k^i$.
   c) Update the weight w_k^i of each particle i based on the existing transition counters $\{C_h^e\}^i$ corresponding to those edges crossed by i when moving from $\mathbf{P}_{k-1}^i$ to $\mathbf{P}_k^i$: $w_k^i \propto w_{k-1}^i \cdot \{\frac{C_h^e + \alpha_h^e}{C_h + \alpha_h}\}^i$, where $C_h = \sum_e C_h^e$ and α_h^e and $\alpha_h = \sum_e \alpha_h^e$ are the prior counts.
   d) Normalize particle's weights to sum to unity.
   e) Update the transition counters $\{C_h^e\}^i$ of the crossed edges corresponding to step starting in $\mathbf{P}_{k-1}^i$ and finishing in $\mathbf{P}_k^i$.
   f) Resample if required.

In a nutshell, particles that revisit similar transitions are rewarded thus allowing the generation of a reliable map of the walked areas.

Steps 1 and 2a, 2b, 2c and 2d are straightforward. We will focus our attention on the two last steps of the RBPF in naive FootSLAM (steps 2e, 2f), which will be explained in the next subsections.

C. Update of Hexagon Transition Counters

The update consists of the addition of one count to the counters of those edges that were crossed when the pedestrian modeled by particle i moved from $\mathbf{P}_{k-1}^i$ to $\mathbf{P}_k^i$. For example, if edge $e = 2$ of hexagon h was crossed, such a transition is mapped as follows:

$$\{M_{h\to e=2}\}^i = \{C_h^0, C_h^1, \mathbf{C_h^2+1}, C_h^3, C_h^4, C_h^5\}^i,$$

where $M_{h\to e=2}$ indicates that the *HTC* has been updated after crossing edge 2.

The update of the *HTC* is the most fundamental step in the RBPF because it represents the basis for the particle weight computation (step 2c). If particle i revisits hexagon h crossing an edge that it had already crossed, it will be rewarded with a greater weight update than the other particles that are not revisiting edge transitions. As a result its hypotheses for the history of pedestrian's poses, step vectors and errors $\{\mathbf{PUE}\}_{0:k}^i$ will be considered to be more likely the one that the pedestrian and the system actually followed.

D. Resampling Step

Resampling means drawing N_p' new particles from the existing set of N_p particles according to their importance weight w_k^i [8]. Drawing a particle (called "resampling" particle) from a generating or "parent" particle usually means copying all its state information. As a consequence, the entire map $\{M\}^i = \{h, \{M_h\}^i\}$ needs to be copied (with replacement) from the resampling particle.

Resampling allows the particle filter to concentrate on particles with large weight and eventually eliminate those that have a low weight.

III. FootSLAM Complexity Analysis

An important aspect of analyzing an algorithm refers to the determination of the needed amount of resources for its execution (time steps or memory requirements) as a function of the algorithm's input size [10].

Algorithms are usually analyzed in the asymptotic sense, i.e. for an arbitrary large input size. Big O notation, omega notation and theta notation are used to this purpose. In this contribution, we focus on big O notation, which shall be explained next.

A. Big O Notation

Big notation, usually denoted as $\mathcal{O}$, is used to express the worst-case scenario for a given algorithm by providing an upper bound for its growth rate.

An algorithm exhibits a growth rate on the order of a mathematical function $\mathcal{O}(f(n))$ if [10]:

$$0 \leq T(n) \leq C \cdot f(n); \forall n \geq n_0,$$

where C and n_0 are positive constants.

An algorithm is said to be constant growth rate if its complexity growth rate is bounded by a value that does not depend on the size of the input n. It is written as $\mathcal{O}(1)$.

B. Naive FootSLAM Complexity Analysis

As discussed above, FootSLAM's implementation of FastSLAM uses, in the limit, N_h separate estimators over the map and one estimator over the pedestrian's poses. Thus, with N_p particles in the particle filter, FootSLAM maintains a total of $N_p \cdot (1 + N_h)$ estimators.

Our implementation of FootSLAM fixes the number of particles N_p and the maximum number of hexagons in the grid N_h at the beginning of the estimation process. However, in the worst-case scenario, a person will visit a constant number of new hexagons H per time step. To cope with this situation, we envision a scalable implementation for FootSLAM in which N_h can be dynamically incremented over time to allow the mapping of large areas. Thus, the number of hexagons in the grid will be considered a linear function of time: $N_h = \mathcal{O}(t)$.

Our goal is to analyze the complexity in time and per time step k of the different steps of the RBPF algorithm given the input parameter N_h: $T(N_h) = T(t)$. Since N_p remains constant during the FootSLAM estimation process, the analysis with one particle is valid for any N_p. The complexity of the FootSLAM algorithm after time t, when the walk is over, can be obtained simply by summing the resulting complexity order per time step over all time steps.

It is clear that steps 2a, 2b, 2c and 2d require constant time at every time step k:

$$T_k^{\text{2a, 2b, 2c, 2d}}(t) = \mathcal{O}(1).$$

We will focus our attention on the last two steps of the RBPF algorithm (update and resampling steps) and analyze their dependency with the number of hexagons N_h and their impact on the complexity growth rate over time.

1) Complexity of the Update Step: The update step in naive FootSLAM is related to the search and insertion costs of a lookup table, $\{M\}^i = \{h, \{M_h\}^i\}$. This lookup table has been implemented using a Hash table whereby collisions are prevented by means of a biunivocal correspondence between the hexagons and their Hash code. Therefore, the time requirements of search and insertion are constant:

$$T_k^{upd}(t) = \mathcal{O}(1).$$

2) Complexity of the Resampling Step: At the resampling step, at each time step, each resampling particle needs to copy (with replacement) the map of the parent particle. Two cases can be easily differentiated when analyzing FootSLAM's complexity:

a) Exploration of White Space: By exploration of white space we refer to the stage in which particles explore new areas, visiting new hexagons. Thus, the size of the Hash table grows linearly with time, making the resampling step require linear time in the number of visited hexagons. As a consequence, the resampling step during exploration of white space (ws) depends on the time step k itself:

$$T_k^{ws}(t) = \mathcal{O}(N_{vis}(k)) = \mathcal{O}(t).$$

b) Revisiting Areas: When the particles revisit hexagons, the size of the lookup table remains constant, e.g. with N_{vis} entries. Therefore, the run-time of the resampling step will not depend on the time elapsed: Each particle needs to copy N_{vis} entries. In that case, the resampling step requires constant time:

$$T_k^{rev}(t) = \mathcal{O}(N_{vis}) = \mathcal{O}(1).$$

As it can be observed, the complexity of naive FootSLAM per time step is dominated by the resampling step during exploration phases, that is, FootSLAM requires time linear in the covered area for each time step k.

Hence, after integrating over all steps k we obtain the requirements in time per FootSLAM estimation process:

$$T^{FS}(t) = \mathcal{O}(t^2). \tag{7}$$

As a conclusion, we can state that the bottleneck of naive FootSLAM is located at the resampling step while new areas are being discovered, showing a **quadratic-in-time** complexity growth rate.

FootSLAM's main goal is the mapping of large areas where N_h tends to infinity in quasi real-time applications [7]. The complexity analysis reveals that the bottleneck in naive FootSLAM is caused by the map copying process during the resampling step. Thus, a per-particle data structure is not a fit approach to build the maps of the particles. A new data structure has been developed to reduce the complexity growth rate of FootSLAM from quadratic to $t \log t$. This data structure is called *H-tree* and it is presented in the next section.

IV. The H-tree: A New Data Structure for Complexity-reduced FootSLAM

The H-tree or Hexagon Tree is a tree-based data structure specifically designed for FootSLAM to reduce its time requirements from quadratic to linear times logarithmic, as suggested in [9]. Before further comments on the H-tree, a brief introduction to basic concepts of a tree-based data structure is summarized for clarity [10]:

1) A tree is an acyclic connected graph in which all nodes have one parent node except for the top node (called root). Trees are usually drawn growing downwards.
2) All nodes have zero or more children nodes. Nodes that have no children are called leaves.
3) The degree of a node refers to the number of children of a node.
4) Any node can be reached following a unique path from the root node. The path is composed of edges or links.
5) The depth of a node is the length of the path that goes from the root to the node. The height of the tree refers to depth of the deepest node in the tree.
6) A balanced tree is a tree whose leaves are all located at the same depth.

As we shall show, the H-tree builds on the R-tree [11], that is, the H-tree is a geographic tree-based data structure. The R-tree is a balanced tree that uses rectangular shapes to quickly index geographical coordinates. Objects that are close to each other are grouped and represented by means of a bounding rectangle. Each node in the tree refers to a rectangle. A leaf node's rectangle refers uniquely to an object. The higher levels contain the aggregation of the lower levels.

The H-tree in FootSLAM is actually composed of two substructures: The Global H-tree (one per FootSLAM estimation process) and the Dynamic H-tree (one per particle). As we shall explain, the choice of such decomposition avoids the computational cost of rebalancing the Dynamic H-trees during the mapping process - as opposed to [9], where only the dynamic structures are present.

Both global and dynamic trees have in common the following characteristics:

- The maximum degree of all internal nodes is s.
- The tree is balanced: all leaves are located at the same depth.
- The height of the tree is $height = \log_s N_h$, being N_h the number of hexagons in the grid (N_h is adjusted to be a power of s or s is adjusted to be a root of N_h).
- Generating and accessing a partial tree (e.g., the path to a leaf) can be done in time $\mathcal{O}(\log_s N_h) = \mathcal{O}(\lg N_h)$.
- Each leaf refers unambiguously to a hexagon in the grid.

In the following subsections, the Global and the Dynamic H-tree are presented in more detail.

A. The Global H-tree

The Global H-tree is a tree-based data structure that exists once and only once per FootSLAM estimation process. It is in charge of the storage and representation of *all* hexagons

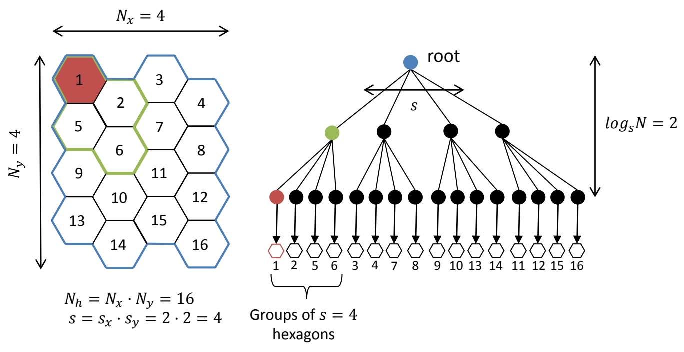

$N_x = 4$
$N_y = 4$
root
s
$log_s N = 2$
1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16
Groups of $s = 4$ hexagons
$N_h = N_x \cdot N_y = 16$
$s = s_x \cdot s_y = 2 \cdot 2 = 4$

Fig. 1. Construction of the Global H-tree: Given the hexagon grid depicted on the left - composed of $N_x = 4$ columns and $N_y = 4$ rows, the resulting Global H-tree using a degree of $s = s_x \cdot s_y = 2 \cdot 2 = 4$ is shown on the right. Whereas the root of the Global H-tree refers to all $N_h = 16$ hexagons in the grid (marked in blue), each node one level below refers to a group of hexagons $s = 4$ times smaller (e.g. the green node refers to hexagons number 1, 2, 5 and 6, and the red node refers only to hexagon 1). The hexagons are stored at the leaf level.

in the grid: The Global H-tree stores only geographic-related information (hexagons coordinates). Similarly to the R-tree, objects (the hexagons) are stored at the leaf level of the tree. The path to each hexagon is stored in a Hash table:

$$paths = \{h, path\}; \forall h \in N_h,$$

allowing quick indexing of hexagons and paths to each hexagon in the tree when a particle accesses its map.

The grid of N_h hexagons is composed of N_x columns and N_y rows. All internal nodes in the Global H-tree have the same degree s, i.e. all nodes are full. At the leaf level there exist N_h nodes, each one holding one hexagon. Higher level nodes refer to a greater number of hexagons (Figure 1). In contrast to [9], where the tree-based data structure has no geographical meaning, the Global H-tree groups hexagons that are nearby. This will have a significant impact when building the Dynamic H-tree for each particle, as we shall see.

The degree of the nodes can be decomposed into two components: s_x and s_y, with the following geographical meaning: Nodes located at level $l = (height - 1)$ refer to groups of s_x columns of s_y hexagons. Figure 1 depicts an example for $N_h = 16$ and $s = 4$.

As stated before, N_h is adjusted to be a power of s or s to be a root of N_h. In order for the Global H-tree to be perfectly balanced and full, N_x must be a power of s_x and N_y a power of s_y for all nodes to have exactly s children. We choose $N_x = N_y$ and $s_x = s_y$ so that the only condition that must apply is N_h to be a power of s.

For the structure to become a tree, $s_x = s_y \geq 2$, i.e. $s \geq 4$. Furthermore, the degree of the nodes is upper bounded by the number of hexagons in the grid, so $4 \leq s \leq N_h$.

In addition, the number of nodes N_n in the Global H-tree can be computed as follows:

$$N_n = \sum_{k=0}^{\log_s N_h} s^k = \frac{s^{\log_s N_h + 1} - 1}{s - 1} = \frac{s \cdot N_h - 1}{s - 1}. \quad (8)$$

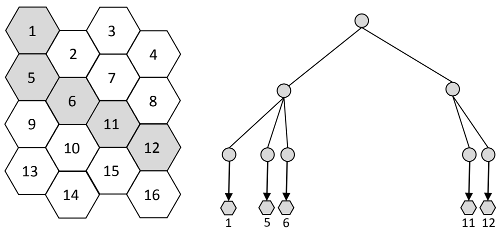


Fig. 2. Construction of a Dynamic H-tree for one particle: On the left side the hexagons visited by the particle are shown. On the right, its corresponding Dynamic H-tree. The paths that lead to each hexagon have been copied from the Global H-tree shown in Figure 1. At the leaf level, not only the visited hexagons are stored, but also their corresponding *HTC*.

Important considerations when choosing s are:

- The greater the s, the fewer the nodes. This can be directly translated into less memory requirements.
- The greater the s, the shallower the tree. Since accessing and generating a partial tree has a cost in time $(\log_s N_h)$, the lower the access time is.
- As shall be shown, the greater the s, the longer the run-time of the resampling step when using the H-tree.

B. The Dynamic H-tree

Each one of the N_p particles in the particle filter builds its own (sparse) Dynamic H-tree based on the *aspect* of the Global H-tree (the relative position between the nodes and links that lead to the hexagons). The main difference with respect to the Global H-tree is that the Dynamic H-tree also stores *HTC*s (hexagon transition counters) of the visited hexagons.

The first time the particle visits a hexagon, the path to the hexagon is obtained from the table of paths of the Global H-tree and the corresponding nodes that lead to the hexagon are copied into the Dynamic H-tree. If the particle revisits a hexagon, the Global H-tree is only needed to retrieve the path to the hexagon in the Dynamic H-tree. Figure 2 illustrates an example for a particle that has visited five hexagons.

In contrast to [9], using the Global H-tree as a reference, there is no need to rebalance the Dynamic H-trees when new hexagons are stored. Nevertheless, this approach still allows for occasional redefinition of the trees, if we need to extend the mapping area, for instance: The number of hexagons in the grid is incremented, the Global H-tree is rebuilt accordingly and the Dynamic H-trees are restructured based on the new Global H-tree.

We have stated that each particle owns a dynamic tree. We shall see that some parts of a Dynamic H-tree (subtrees) can be shared by particles that have a common history of visited hexagons. This is the strength of the H-tree and it is further explained in the next subsections within the context of the resampling and update steps of the FootSLAM algorithm.

C. Resampling Step in the Dynamic H-tree

As explained above, at the resampling step a set of N_p' particles is generated from the existing N_p particles based on their

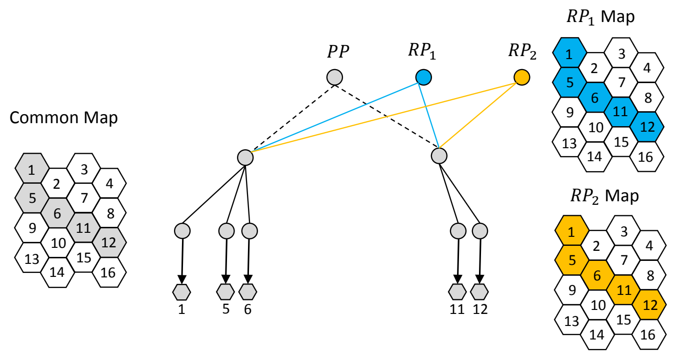


Fig. 3. Resampling in the H-tree: Two resampling particles RP_1 and RP_2 resample from a parent particle PP by copying the pointers of its Dynamic H-tree root-node. RP_1 and RP_2 share the common hexagon map depicted on the left.

weights. A naive approach duplicates the state information for all the particles that resample from a parent particle, as we showed with the Hash table (Section III-B2).

The H-tree allows to use another approach whereby all resampling particles share the information from the parent particle, avoiding duplicity of information: Equivalently to [9] a resampling particle can be copied from a parent particle by merely copying its Dynamic H-tree root-node pointers (maximum s). Figure 3 illustrates this concept for two resampling particles RP_1 and RP_2 that sample from a parent particle PP.

Operating in this manner, the resampling part of the algorithm does not depend on the number of visited hexagons by each particle, but on the degree of the H-tree s, which is a constant value. The lower the s, the shorter it takes to perform resampling.

D. Update of Hexagon Transition Counters in the Dynamic H-tree

When two or more particles resample from the same parent particle, they share the same nodes and the same visited hexagons with their corresponding *HTC*s (hexagon transition counters). If one of these particles needs to update the *HTC* of a hexagon, then the path leading to it needs to be duplicated (with replacement) [9]. Duplicating a path means duplicating its nodes and links. Figure 4 shows an example for particles RP_1 and RP_2.

As stated above, accessing or updating a leaf has a logarithmic cost $\mathcal{O}(\lg N_h)$. In contrast to naive FootSLAM, the update step using the H-tree shows a dependency on the number of hexagons in the grid N_h.

E. Complexity Analysis

The advantage of using the H-tree is that instead of duplicating the entire tree when resampling, only those paths that refer to a new or revisited hexagon need to be duplicated.

The resampling step requires always constant time, since it depends only on the degree of the tree s:

$$T_k^{res}(t) = \mathcal{O}(1).$$

Nevertheless, the update step requires time logarithmic in the number of hexagons in the grid N_h, which can be dynami-

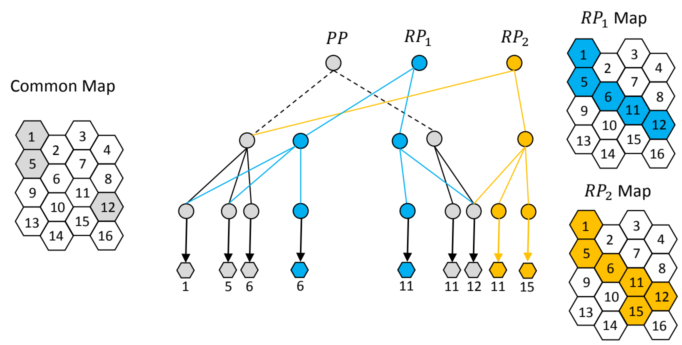


Fig. 4. Update step in the Dynamic H-trees: After the resampling step, RP_1 exits hexagon number 11 and revisits hexagon number 6. These hexagons' *HTC*s are common to particle RP_2 and cannot be directly updated. Instead, the paths that lead to them must be duplicated before the update to avoid changing the map of particle RP_2. Particle RP_2 visits a new hexagon (number 15) and needs to copy some of the nodes to avoid adding visited hexagons to RP_1. On the left, the common map to both particles after the update. On the right, the map that each particle possesses.

cally incremented over time to meet the mapping requirements:

$$T_k^{upd}(t) = \mathcal{O}(\lg N_h) = \mathcal{O}(\lg t).$$

Hence, the new bottleneck of the FootSLAM algorithm is located at the update step and consequently, the run-time of H-tree FootSLAM is bounded by:

$$T^{FS}(t) = \mathcal{O}(t \lg t).$$

Note that this analysis corresponds to our envisioned implementation of large-scale FootSLAM whereby the number of hexagons in the grid N_h can be increased dynamically to allow the mapping of larger areas.

However, in our current H-tree implementation N_h remains constant through the whole FootSLAM estimation process and the Global H-tree is built according to it. In this case, FootSLAM requires linear time, given that the $\lg t$ factor is fixed by the constant height of the tree over time.

F. Memory Savings

A further advantage of using a tree-based data structure is the savings in memory, visible between resampling steps. When resampling, N_p' new unique particles are generated from the old set of N_p particles by copying the pointers of the parent particle's root-node. If on average X particles are drawn from a given particle, then we experience a $1/X$ reduction in memory since we can reclaim the memory occupied by the obsolete particles. In the limit, when all particles resample from one single surviving particle, we experience a ratio of $1/N_p$ savings in memory. Note that these memory savings are relative to the previous resampling step.

Figure 5 illustrates this idea over 4 resampling steps. At each resampling step some hypotheses are erased (marked with a cross) and memory is deallocated.

The reason for using a geographic approach for the H-tree is even clearer at this point: We live and work in buildings where large, unvisited spaces are rare. Therefore, our geographically defined Dynamic H-trees for a completed FootSLAM map will tend to become densely populated, except for perhaps the last (meter level) nodes. The underlying assumption which

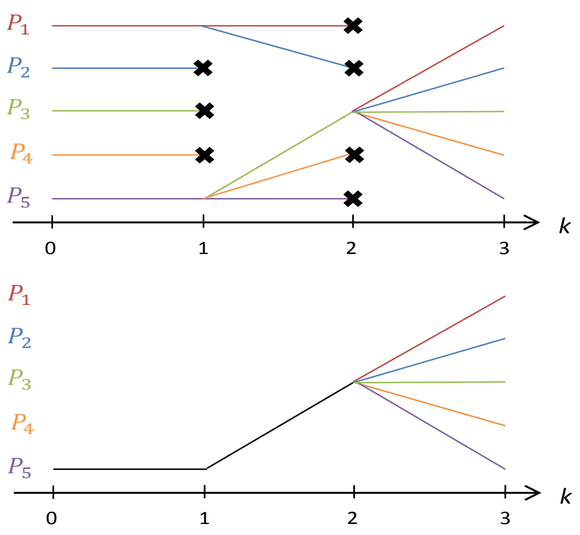


Fig. 5. On the top, particles' hypotheses lifetime: Five different particles P_i are initialized at time step $k = 0$. At resampling time step $k = 1$, particle P_1 and P_2 resample from particle P_1. P_2 previous hypothesis is erased, marked with a cross. Likewise, particles P_3, P_4 and P_5 resample from P_5. At resampling time step $k = 2$, all particles are resampled from P_3. The only surviving hypothesis is that of P_3. On the bottom, the surviving history of hypotheses for the five particles is shown. At $k = 2$ the memory requirements have been reduced 5 times with respect to $k = 1$.

we believe justifies this is that for a given area, human accessibility is more or less uniform, except for meter level granularity.

If we assume that the space accessible to a pedestrian is not randomly distributed, but consists of groups of accessible areas (hexagons, in our case), then a geographic tree-based data structure will lead to a lower number of nodes needed in each Dynamic H-tree. Our Global H-tree is built grouping hexagons that are nearby. Thus, visiting surrounding hexagons between two resampling steps translates into very local changes in the Dynamic H-trees. Only the nodes at the lower levels, i.e., a subpath, need to be generated. For example, suppose that particle RP_2 in Figure 4 visits hexagon number 16 after hexagon number 15. Then, adding hexagon 16 to the tree requires only adding one extra leaf node. As a consequence, in average less memory needs to be allocated than in the case in which hexagons are randomly stored at the leaf level.

V. Compact Representation of Hexagon Transition Counters

Using concepts of source coding and lossless data compression [12], we can encode the different combinations of transition counts that the counters of an *HTC* (hexagon transition counters) can take on for any hexagon h. This codification is a compact representation (alphabet) for the set of transition counters across the edges of the hexagons.

The alphabet can map up to T transitions per edge. Therefore, there are $(T + 1)$ possible values (to account for 0 transition counts) for the six counters to adopt and repetition is allowed. Hence the alphabet A is composed of $S = (T+1)^6$ symbols. The first symbol A_0 and last symbol A_{S-1} of the

$A_0 \leftrightarrow \{0,0,0,0,0,0\}$

$A_{0\to e=0}$	$A_{0\to e=1}$	$A_{0\to e=2}$	$A_{0\to e=3}$	$A_{0\to e=4}$	$A_{0\to e=5}$
$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
{1,0,0,0,0,0}	{0,1,0,0,0,0}	{0,0,1,0,0,0}	{0,0,0,1,0,0}	{0,0,0,0,1,0}	{0,0,0,0,0,1}

Fig. 6. Example of symbol transitions given symbol $A_0 \leftrightarrow \{0, 0, 0, 0, 0, 0\}$. Given a transition across any of the edges, the next possible symbols have been precomputed.

alphabet correspond, respectively, to:

$$\begin{array}{rcl} A_0 & \leftrightarrow & \{0,\ 0,\ 0,\ 0,\ 0,\ 0\} \\ A_{S-1} & \leftrightarrow & \{T, T, T, T, T, T\} \end{array}$$

Similarly to the Global H-tree, this alphabet is constructed at the beginning of the FootSLAM estimation process and exists only once. The alphabet is particularly effective during the exploration phases of FootSLAM that requires much particle diversity. As it shall be explained next, it will help reduce FootSLAM memory requirements.

1) Memory Savings: The alphabet is used to avoid storing the full *HTC* of the hexagons that the particles visit. Instead, the symbol corresponding to the *HTC* is stored at the leaf level of the particle's Dynamic H-tree.

The *HTC* data structure consists of a vector of length six. In our Java implementation it has been constructed using an array of six Byte values, which requires 24 Bytes. On the other hand, each alphabet symbol is represented by a Short value, which requires only 2 Bytes.

Nevertheless, one should note that once a particle enters or leaves a hexagon across an edge whose counter had registered the maximum number of transitions, T, the symbol is converted back to an array of Bytes (*HTC*). The particle's Dynamic H-tree will then store the full *HTC* for that hexagon.

2) Efficient Map Updates: Say a particle is located at a hexagon whose associated set of transition counters M_j corresponds to symbol A_j. The particle can leave the hexagon using any of the six edges. The particle can update the symbol by:

- Retrieving the corresponding M_j of symbol A_j.
- Updating M_j given the transition across edge e: $M_{j\to e}$ as explained in Section II-C.
- Obtaining the symbol $A_{j\to e}$ that corresponds to the updated *HTC* $M_{j\to e}$, if it exists.

However, for efficient map updates, for each alphabet symbol A_j the six next possible symbols $A_{j\to e}\forall e$ are precomputed and stored during the construction of the alphabet. Figure 6 represents an example for symbol A_0.

Operating in this manner, the update of a symbol requires only to know which edge was crossed when leaving the hexagon, avoiding explicit conversion to an *HTC*.

During exploration phases, when particles are visiting new hexagons and there exists much diversity of hypotheses, memory requirements grow with the visited area. The alphabet will play a significant role during those phases, reducing the rate of growth of the memory requirements.

VI. Storage of Diversity

This section presents an introduction to a possible approach to store particle's diversity when mapping large areas, yet to be implemented and tested.

The dimension of a FootSLAM map grows as more and more places, floors and rooms are visited. Two problems arise with a growing map. First, the total map size is constrained due to memory limitations. The second problem is a diversity loss of the particles over long trajectories.

As described in [13] a particle filter tends to loose diversity of hypotheses for those areas that are far away in time. The reason is the resampling process, where locally good particles are duplicated and weak particles are erased. This causes particle depletion in areas that had been previously visited, since the resampling process eliminates hypotheses that were valid in the previous locations. Thus, particles' estimates and their diversity for those areas should be conserved or frozen and independent of a future resampling. In order to solve the short-term memory limitation problem and the diversity loss, parts of a map can be compressed and stored in a different structure.

In [14] an approach is presented to combine multiple maps of particles to one Gaussian map representation as soon as the specific part of the map is out of view. In FootSLAM, each particle holds information regarding the hexagon it visits and their corresponding *HTC*s (hexagon transition counters). This information can be effectively compressed in one *cumulative map*, which combines the information of all particles (Equations (5) and (6)).

Figure 7 illustrates the processes of storing and restoring a part of the map. The particles' maps are compressed into a cumulative map, removed from the Dyamic H-trees and saved into long term memory. When revisiting the area, the cumulative map is selected and used as prior map for FootSLAM, as described in [6] Section 2.

Now the challenge is to determine the moment in time for the storage of certain areas of the map. A map area which is far away from the actual pose and exceeding a certain defined distance can be selected as soon as that distance is reached. The distance should be chosen carefully, as the particle filter needs sufficient space for different hypothesis to resolve ambiguity. Another option is to use a logical separations between buildings or even floors. If an absolute position measurement (e.g. GPS, RFID tag, radio positioning) is available between buildings, the map can also be divided. Another option is a non-automated, manual separation from the user by signaling as soon as an area or building is left.

VII. Experimental Verification

A. Methodology

To evaluate the performance of FootSLAM, a pedestrian undertook a walk that lasted over 13 minutes using an IMU sensor located on his foot to measure his steps. FootSLAM has been specifically developed to map indoor environments. Nevertheless, this walk was undertaken outdoors covering a relatively large area (ca. 100m by 100m) to show the behavior of FootSLAM during the exploration phase.

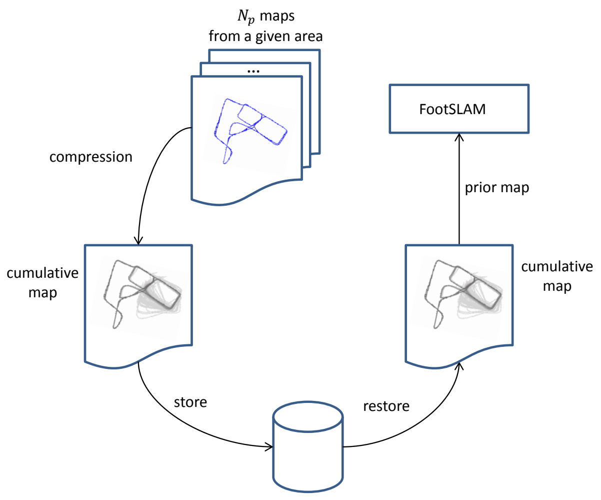


Fig. 7. Storing and restoring processes used to avoid losing diversity of hypotheses in the particle filter. The existing N_p maps from a given area (on the top) are compressed into a *cumulative map*, which is stored into disk. The particles' maps are erased from memory. When the particles revisit that same area, the cumulative map is retrieved from disk and used as a prior map during the FootSLAM algorithm to guide the particle's trajectories and help the convergence of the algorithm.

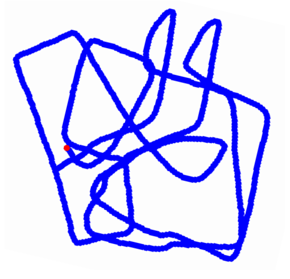
The recorded data, after being processed by a lower-level Unscented Kalman Filter (UKF), were used as input for FootSLAM. The UKF output data - called odometry data (Figure 8(a)) - were processed offline by FootSLAM using $N_p = 30000$ particles and a grid of $N_h = N_x \cdot N_y = 256 \cdot 256$ hexagons. Figures 8(b) and 8(c) show the FootSLAM maps corresponding to the odometry data. Clearly, FootSLAM was able to correct the drifting errors visible in the odometry data.

In the following we will use the term naive FootSLAM to refer to the implementation of FootSLAM of Section III-B and H-tree FootSLAM for the complexity-reduced implementation based on the H-tree. Naturally, the resulting map is the same no matter which approach is used.

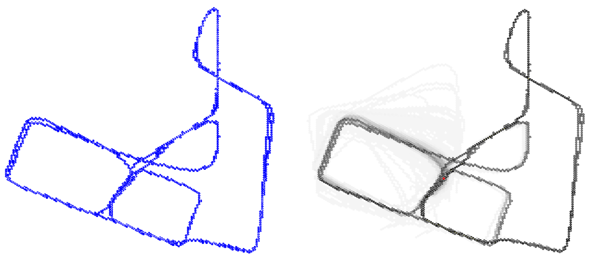
The maximum number of mapped transitions by the alphabet, T, and the degree of the nodes in the Global H-tree, s, are varied and the performance of FootSLAM is studied accordingly.

FootSLAM has been implemented on a Java platform. In the following subsections, the constraints that the implementation presents for T and s are explained.

1) Maximum Number of Mapped Transitions by the Alphabet: We represent alphabet symbols by means of a Short type, which in Java covers a range of $(2^{32} - 1)$ possible values. Consequently the number of symbols in the alphabet S is bounded by this value and the maximum value that T can take on is: $T = \lfloor \log_6(2^{32} - 1) \rfloor - 1 = 5$. From the moment when a particle has crossed one of the edges of a hexagon more than 5 times, the symbol is converted to *HTC*.



(a) Odometry



(b) Maximum a posteriori map (c) Cumulative map

Fig. 8. Odometry recorded during the walk (Figure (a)) and FootSLAM output maps: map corresponding to the maximum a posteriori hypothesis (hypothesis of the heaviest particle in the particle filter) for the history of pedestrian's poses (Figure (b)) and cumulative map with the aggregated information of all particles (Figure (c)).

2) Limit for the Degree of the Nodes: We use a Byte representation for the links of each node, so we must limit the degree s of the nodes to be 255 or fewer. Applying all conditions stated in Section IV-A, the only two values we can use for the H-tree are: $s = 4$ and $s = 16$.

Note that we have chosen to encode the path using a vector of Byte values due to the considerable amount of paths to be stored (N_h) when the Global H-tree is generated. Using a Short type or an Integer type is also possible if the available memory allows it.

B. Results

In this section we present the performance of naive FootSLAM and H-tree FootSLAM both relative to time and memory requirements.

1) Time Performance: Figure 9 shows the measured time performance of naive FootSLAM (red curve) and H-tree FootSLAM for two different degree values: $s = 4$ (pink line) and $s = 16$ (blue line for $T = 5$ and green line for $T = 0$, which are overlapping).

Our experiment results are consistent with the theoretical analysis of FootSLAM: Naive FootSLAM shows a quadratic in time complexity growth rate (the dashed cyan curve corresponds to the best quadratic-fitting curve). On the other hand and as discussed in Section IV-E, H-tree FootSLAM requires time in $t \lg t$, which for a fixed number of hexagons in the grid N_h, results in linear time complexity growth rate.

The use of a tree-based data structure like the H-tree shows a clear advantage over the previous naive FootSLAM implementation. After ca. 400 seconds, naive FootSLAM is not able to perform in real-time. On the other hand, the time requirements of H-tree FootSLAM stay always under the real-time processing limit (black line).

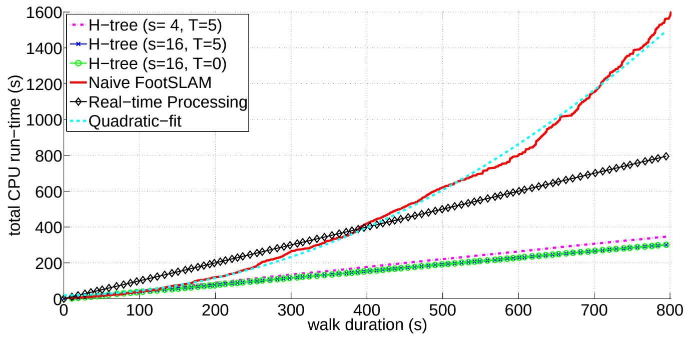


Fig. 9. Measured run-time of the FootSLAM algorithm for the walk depicted in Figure 8 for different FootSLAM implementations: naive FootSLAM (red curve) and H-tree FootSLAM (dash-dotted pink line, blue line with cross markers and green line with circle markers). Real-time processing is depicted by a black line diamond markers. The dashed cyan curve represents the best fitting quadratic curve to the performance of naive FootSLAM.

The use of different values for the degree of the nodes gives the same order of complexity growth rate (linear times log), but changes the speed of growth (the slope). In this case, a greater degree value (blue and green lines) improves the performance of H-tree FootSLAM. Such result does not imply that the performance will always improve for a greater degree value: A greater s will also slow down the resampling step.

Unsurprisingly, using the alphabet does not improve the performance in terms of run-time: for $T = 0$ (no alphabet) and $T = 5$ the performances are practically the same (the blue and green line overlap). Our approach of storing the possible symbol transitions (Section V-2) has allowed the alphabet-based implementation to maintain the speed of the update step of the implementation with no alphabet. The advantage of using the alphabet is visible in terms of memory requirements, as we shall show next.

2) Memory Performance: Figure 10 depicts the measured memory performance of naive FootSLAM (red curve) and H-tree FootSLAM for two degree values, $s = 4$ (pink curve) and $s = 16$ (blue and green curves). The sawtooth shape of the curves are due to the memory deallocation process executed by Java's garbage collector.

As argued at the end of Section IV-F, H-tree FootSLAM performs better than naive FootSLAM in terms of memory. This is visible in Figure 10 since the highest memory requirements of H-tree FootSLAM correspond to ca. 3750MB (green curve, $s = 4$), whereas naive FootSLAM highest memory requirements correspond to ca. 8000MB.

As expected, given a fixed maximum number of mapped transitions ($T = 5$), the memory performance of H-tree FootSLAM for $s = 16$ (blue curve) improves that of H-tree FootSLAM with $s = 4$ (pink curve). This is because the number of nodes that need to be generated to insert all visited hexagons is lower for a greater degree value (Equation (8)). The improvement is visible at the lower peak memory requirements and at the slower rate of growth of memory allocations (a less steep slope of the saw tooth).

On the other hand, given a degree value of $s = 16$, the use of the alphabet (blue curve) improves the memory performance of the H-tree implementation without an alphabet (green curve).

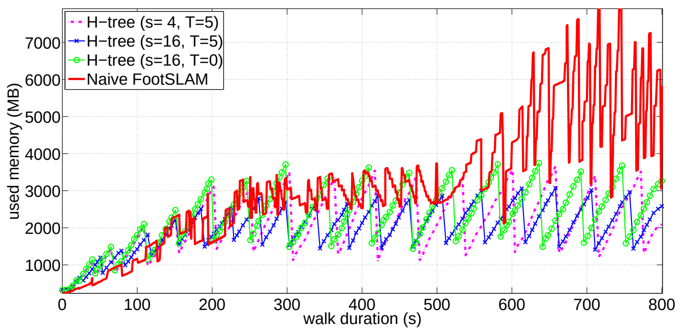


Fig. 10. Measured memory performance of naive FootSLAM (red curve) and H-tree FootSLAM with degree values $s = 4$ (dash-dotted pink curve) and $s = 16$ (green curve with circle markers ($T = 0$) and blue curve with cross markers ($T = 5$)).

VIII. Conclusion and Outlook

FootSLAM or simultaneous localization and mapping (SLAM) for pedestrians is a technique that addresses the indoor positioning and mapping problem based on human odometry (aka pedestrian dead reckoning), e.g. with a foot-mounted inertial sensor. FootSLAM builds on a Rao-Blackwellized Particle Filter in which each particle holds its own history of pedestrian's poses and map estimates. To that end, the 2D space has been divided into a grid of uniform and adjacent hexagons.

In this paper the complexity of a naive FootSLAM implementation has been studied. The resampling step represents the bottleneck, showing a linear dependency with the covered area per time step. Such dependency makes FootSLAM exhibit a quadratic-in-time complexity growth rate that prevents real-time processing.

A new geographic tree-based data structure, called the H-tree, has been introduced to reduce FootSLAM's computational growth rate in time from quadratic to linear times logarithmic.

The H-tree consists of two substructures: one Global H-tree per FootSLAM estimation process and one Dynamic H-tree per particle. The Global H-tree is a predefined structured used as a reference to build the Dynamic H-trees, thus avoiding the computational cost of rebalancing the trees when new hexagons are visited. Furthermore, both H-tree substructures are geographic-based, which helps control the growth in the number of nodes needed in the Dynamic H-trees.

The main advantage of using the H-tree is visible at the resampling step, now independent of the number of hexagons visited by the particles. The cost of using this structure is present at the update step of the particle filter, with a logarithmic cost per time step.

In addition, an alphabet of symbols has been introduced to compactly represent map estimates (hexagon transition probabilities) by the particles. The alphabet reduces the memory requirements of FootSLAM especially during exploration phases in which much particle diversity of hypotheses is needed.

We have validated the computational savings of the H-tree and alphabet-based FootSLAM implementation with respect to a naive FootSLAM implementation, both in terms of time and memory consumption. Our experiments show that the new implementation is suitable for mapping and positioning applications with real-time requirements. Furthermore, the reduction in complexity will allow FootSLAM to use more particles when mapping large areas, thus helping the accuracy of the algorithm. In our view these results are the first step towards efficiently undertaking the mass market offline mapping process.

Current work is focusing on the extension of the H-tree and the alphabet to allow real-time processing of FootSLAM in 3D environments.

Further work should target the possibility of increasing the number of hexagons in the grid N_h as required by the walk itself. Such increase would need reshaping the Global H-tree and the Dynamic H-trees of each particle, but would allow for self adaptability to the the environment and mapping of larger areas when needed.

Further research should also address the quantification of FootSLAM accuracy.

Acknowledgment

The authors would like to thank Mohammed Khider for the support and fruitful discussions.

References

[1] E. Foxlin, “Pedestrian tracking with shoe-mounted inertial sensors,” in *IEEE Computer Graphics and Applications*, vol. 25, no. 6, 2005, pp. 38–46.
[2] S. Beauregard, Widyawan, and M. Klepal, “Indoor PDR performance enhancement using minimal map information and particle filters,” in *Proc. of the IEEE/ION PLANS 2008, Monterey, USA*, 2008.
[3] O. Woodman and R. Harle, “Pedestrian localization for indoor environments,” in *Proc. of the UbiComp 2008, Seoul, South Korea*, 2008.
[4] B. Krach, P. Robertson, M. Angermann, and M. Khider, “Inertial systems based joint mapping and positioning for pedestrian navigation,” in *Proc. ION GNSS 2009, Savannah, Georgia, USA*, 2009.
[5] M. Angermann and P. Robertson, “FootSLAM: Pedestrian simultaneous localization and mapping without exteroceptive sensors- hitchhiking on human perception and cognition,” *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1840 –1848, 5 2012.
[6] P. Robertson, M. Garcia Puyol, and M. Angermann, “Collaborative pedestrian mapping of buildings using inertial sensors and FootSLAM,” in *ION GNSS 2011, Portland, Oregon, USA*, september 2011.
[7] M. Garcia Puyol, M. Frassl, and P. Robertson, “Collaborative mapping for pedestrian navigation in security applications,” in *Future Security 2012, Bonn, Germany*, september 2012.
[8] H. Durrant-Whyte and T. Bailey, “Simultaneous Localization and Mapping: Part I Tutorial,” 2006.
[9] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, “FastSLAM: A factored solution to the simultaneous localization and mapping problem,” in *Proc. AAAI National Conference on Artificial Intelligence, Edmonton, Canada*, 2002.
[10] T. H. Cormen, C. Stein, R. Rivest, and C. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.
[11] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *International Conference of Management Data*. ACM, 1984, pp. 47–57.
[12] K. Sayood, *Introduction to Data Compression*, 3rd ed. Morgan Kaufmann, 2005.
[13] T. Bailey, J. Nieto, and E. Nebot, “Consistency of the FastSLAM algorithm,” in *Robotics and Automation. ICRA. Proceedings IEEE International Conference on*, may 2006, pp. 424 –429.
[14] A. Brooks and T. Bailey, “HybridSLAM: Combining FastSLAM and EKF-SLAM for reliable mapping,” in *WAFR*, 2008, pp. 647–661.