Agent Classification using Implicit Models

Nicholas M. Stiffler

Jason M. O’Kane

Abstract—We present an algorithm that uses a sparse collection of noisy sensors to characterize the observed behavior of a mobile agent. Our approach models the agent’s behavior using a collection of randomized simulators called implicit agent models and seeks to classify the agent according to which of these models is believed to be governing its motions. To accomplish this, we introduce an algorithm whose input is an observation sequence generated by the agent, represented as sensor label-time pairs, along with an observation sequence generated by one of our implicit agent models and whose output is a measure of the similarity between the two observation sequences. Using this similarity measure, we propose two algorithms for the model classification problem: one based on a weighted voting scheme and one that uses intermediate resampling steps. We have implemented these algorithms in simulation, and present results demonstrating their effectiveness in correctly classifying mobile agents.

I. INTRODUCTION

Many different kinds of distributed sensing systems have been proposed to observe the movements of various kinds of agents (such as humans, animals, or robots) through their environments. In this paper we consider the problem of classifying the behaviors of those agents, based on the observations of a sparse collection of unreliable, very-low-resolution sensors.

Specifically, we describe an algorithm whose input is a sequence of observations generated by these kinds of sensors, along with two or more agent models that predict how the observed agents might move through the environment. The output of the algorithm is the index of the agent model that best describes the observed behavior, along with a measure of the algorithm’s confidence in that selection.

In a broad sense, efficient algorithms for these kinds of problems would have a variety of practical applications, including detection of malicious behavior in crowded places such as the Shilin Night Market in Taipei, or for monitoring large-scale regions of ocean space using a relatively sparse density of sensors (Figure 1). In both of these scenarios, there would be significant value in having access to a predictive model of how the agents in the system behave. In this paper, we consider how to define and utilize such predictive models even when the behavior of the agents is too complex to be readily modeled as an explicit probability distribution.

Current methods for these kinds of problems, such as hidden Markov models (HMMs) generally rely on complex probabilistic models that are difficult for humans to construct and over which inference can become computationally expensive if there are many hidden variables. We avoid these limitations by replacing the standard probabilistic models for agent movements with implicit models based on learned or expert-coded simulations. Such implicit models allow a broad class of agent behavior patterns to be modeled and reasoned about with minimal computational expense.

We are also specifically interested in the effects of highly unreliable sensors including, in the extreme case we consider here, one-bit sensors with high rates of false positive and false negative errors.

The contribution of this research is a pair of classification algorithms that use a sparse collection of sensors, along with a set of implicit agent models, to characterize the observed behavior of a mobile agent. Specifically:

1) We introduce a formal definition for implicit agent models by describing a small set of operations that these models must provide.

2) We present a criterion for measuring the similarity between two sequences of time-stamped observations, using a generalization of the classical Levenshtein distance algorithm.

3) Based on this similarity measure, we describe two algorithms that can classify mobile agents, using a collection of implicit models.

4) We present a series of simulated results to show that these algorithms solve the classification problem correctly in a substantial majority of cases.

The remainder of the paper is organized as follows. We begin with a discussion of related work in Section II. In Section III we present our problem formulation, followed by an algorithmic description of our classification algo-
rithms in Section IV. We then present some simulation results in Section V. We conclude the paper in Section VI with some closing remarks and thoughts on future work.

II. RELATED WORK

This paper builds upon and extends several bodies of existing work.

A. Agent movement models

Our problem relies heavily on the existence of predictive models that attempt to predict how observed agents will move in the future. There are two principal trains of thought for how to construct such models. Most directly, other researchers have created specialized simulation techniques for various specific kinds of agents, including crowds [11], evacuees in disasters [24], vehicles in traffic [1], and evasive people in general [25]. We utilize models of this type, and build upon the existing work by using its models for inference.

The other common method for describing the movement patterns of an agent is to use a Markov model, in which time is divided into a sequence of discrete stages, and the likelihood of each state at time \( t \) is expressed as a conditional probability given the state at time \( t-1 \). In this context, the term “state” refers to the agent’s location, along with any other internal information that has some impact on how that location changes. When the state cannot be directly observed, the resulting model is called a hidden Markov model (HMM) [3]. These kinds of transition systems also form an integral part of the study of partially observable Markov decision processes (POMDPs) [14], [15], [19], [22], [23], [26], [35], which generalize HMMs by allowing a decision-maker to choose a sequence of actions that also influence the state. However, because of the number of parameters involved, in most cases the state spaces and the transition models used in this research are relatively simple. The agent models used in this project can be viewed, in the strictest sense, as HMMs. However, by using an implicit, simulation-based representation, it becomes practical to use realistic models whose states and transitions would be challenging to represent directly.

B. Probabilistic filtering

Techniques for filtering data from multiple sensors have been studied extensively in many contexts, including the robotics community. Many recent successes have been based on a probabilistic approach [32], in which a robot and its sensors are modeled using conditional probability distributions. In this case, the robots’ knowledge at any point in time is a probability distribution over a space of possible states. This distribution can be represented exactly using parametric models [16]–[18], [20], [30] or approximated by discretization [4], [5], [10], [28] or sampling [8], [9]. The present research differs from those methods because we do not require explicit probability models for the agents’ movements.

C. Distributed target tracking

Our research is also closely related to prior work on target tracking algorithms, in which the goal is generally to maintain some knowledge about the location of one or more moving targets. Many of these methods utilize wireless sensor networks and seek to optimize the communication between the sensor nodes, without much regard for the reliability or content of the sensor data [2], [6], [12], [13], [27], [29], [34]. Our work is complementary, in the sense that we focus on the content of the sensors’ observations and the inferences that can be drawn from them.

III. PROBLEM FORMULATION

This section introduces our problem, including some basic notation (Section III-A), the implicit agent models that we assume are given as part of a problem instance (Section III-B), and a precise statement of the classification problem we consider (Section III-C).

A. Representing the environment, agent, and sensors

The environment is a closed and bounded set \( P \subset \mathbb{R}^2 \). A single agent is modeled as a point that moves through \( P \). The agent’s trajectory is a continuous function \( x : [0, \infty) \to P \), so that \( x(t) \in P \) denotes the position of the agent at time \( t \geq 0 \).

Scattered throughout the environment is a collection of beam sensors, denoted \( B = \{b_1, \ldots, b_n\} \). Each beam is a line segment contained in \( P \), with its endpoints on the boundary of \( P \). The beams are pairwise disjoint, except possibly at their endpoints. Figure 2 shows this notation.

As the agent moves through the environment, the beam sensors generate a collection of observations, drawn from an observation space \( Y = B \times [0, \infty) \), in which each observation \( y_i = (b_i, t_i) \), represents a beam-time pair. Each observation is caused by one of two kinds of events:

1) The observation might be a false positive error, indicating that \( x(t_i) \notin b_i \). These errors are assumed to be random and independent. The probability that a false positive error will be generated by a given sensor at each time step is a known parameter denoted \( p_{FP} \).
2) The observation might be a true positive, which means that \( x(t_i) \in b_i \). The probability that any observation is a true positive is \( (1 - p_{FP}) \).

In addition, it is also possible for the agent to cross a beam without generating a observation. This situation is called a false negative error, and occurs with probability \( p_{FN} \). To simplify the notation, we assume that all of the sensors are equally reliable, so that \( p_{FP} \) and \( p_{FN} \) are not dependent on the specific sensor that may be producing the error.

**B. Implicit agent models**

In addition to this sensor data, we assume that our system has access to a collection \( M = M_1, \ldots, M_m \) of implicit models of how the agents might behave. The models are expressed as randomized simulators that generate plausible motions for the agent. Specifically, we assume that the following three operations can be carried out efficiently:

1) An initialization operation
   \[
   a \leftarrow \text{INITIALIZE}(M_i)
   \] (1)
   that returns a new agent simulation (also called a trace) \( a \), starting at some randomly determined location in the environment.

2) A forward simulation operation
   \[
   \langle y_1, \ldots, y_k \rangle \leftarrow \text{SIMULATE}(a, \Delta t)
   \] (2)
   that uses the model to execute the simulation forward by the given time increment, and returns an ordered collection of beams crossed by the simulated agent during that time. The SIMULATE operation will generally operate in a randomized way, in order to model agents whose movements are not fully predictable.

3) A “cloning” operation
   \[
   a' \leftarrow \text{FORK}(a)
   \] (3)
   that creates another trace with the same position and internal state, such that the beam crossings resulting from future calls to SIMULATE will be distributed identically for \( a \) and \( a' \). The FORK operation will generally involve a simple copying of the state of the simulation, including the agent’s current position, along with any internal variables used by the simulator. The FORK operation is crucial to the efficiency of the algorithms proposed below because it allows the algorithms to focus more computational effort on simulation traces that more closely match the observed/anticipated observation sequences.

The intuition is that our algorithms have access to well-defined simulators for how the agents it observes might behave. Any agent model for which these operations are available is suitable for our approach; we make no assumptions on the nature of the simulation nor on the internal data that it maintains.

**C. Model classification**

Based on these definitions, we can state the model classification problem.

<table>
<thead>
<tr>
<th>Problem (model classification):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: A sequence ( Y_A = \langle y_1, \ldots, y_n \rangle ) of time-stamped observations, a collection of implicit agent models ( M_1, \ldots, M_m ), a false positive probability ( p_{FP} ), and a false negative probability ( p_{FN} ).</td>
</tr>
<tr>
<td><strong>Output</strong>: The index ( i \in {1, \ldots, m} ) of the model employed by the agent, along with a numerical estimate of the probability that the classification is correct.</td>
</tr>
</tbody>
</table>

**IV. DESCRIPTION OF ALGORITHMS**

This section introduces two algorithms to solve the model classification problem introduced in the previous section. The starting point for both of these algorithms is a method for measuring the similarity between pairs of observation sequences. Section IV-A describes this method. We describe the first model classification algorithm, which is based on a relatively straightforward weighted voting scheme, in Section IV-B. The second algorithm, which uses a resampling scheme to accelerate the classification process, appears in Section IV-C.

**A. Editing observation sequences**

Our classification algorithms are based on a criterion for measuring the similarity between any two sequences of observations. The intuition is to compare \( Y_A \) to sequences of beam crossings from our implicit agent models, under the hypothesis that the greater the similarity between \( Y_A \) and any particular beam sequence generated by some model \( M_i \), the more likely it is that the agent belongs to model \( M_i \).

To perform this comparison, we use a generalization of the Wagner-Fischer algorithm [33] which computes the Levenshtein distance (also known as minimum edit distance) between two strings over a finite alphabet [21]. Traditionally, the Levenshtein distance is defined as the smallest number of single-character insertion, deletion, and substitution operations needed to “edit” one string into another. Our problem differs in several important ways:

1) Instead of discrete characters, we have observations, each of which combines a discrete beam index with a real-valued timestamp.

2) The substitution operation is not applicable, because it does not have a physical analog in this context.

3) Because the potential sensing errors have different probabilities of occurring, we assign costs derived from the error model to each editing operation.

Specifically, our algorithm computes the optimal sequence of edits to transform a given observation sequence \( S = \langle s_1, \ldots, s_m \rangle \) into another given observation sequence \( T = \langle t_1, \ldots, t_n \rangle \), using three distinct operations:
• **Insert**: This operation accounts for the instance where \( T \) contains an observation that is missing from \( S \). Because this could be the result of a false negative occurring in \( S \), we derive the cost of this operation from the false negative error rate, so that
\[
C_I = -\log(p_{FN}).
\]  

• **Delete**: This operation occurs when there is an observation in \( S \) that does not appear in \( T \), which could be the result of a false positive occurring in \( S \). Therefore the cost of this operation is derived from the false positive error rate:
\[
C_D = -\log(p_{FP}).
\]

• **Timeshift**: This operation occurs when two observations have the same beam label, but have different time-stamps \((b^{(s)}_i = b^{(t)}_j \text{ and } t^{(s)}_i \neq t^{(t)}_j)\). This could occur when the sequences are generated by the same model, but have slightly different motions within the environment. In this case, the two sequences should generate similar observation sequences, but with relatively small variations in the timestamps. In this case, the operation cost is the true positive rate plus a constant parameter \( \alpha \) times the difference in timestamps:
\[
C_T(t^{(s)}_i, t^{(t)}_j) = -\log(1 - p_{FP}) + \alpha |t^{(s)}_i - t^{(t)}_j|.
\]  

We must also take into account the cost of a true positive, which occurs when two observations “match,” and is directly related to the false positive error rate:
\[
C_M = -\log(1 - p_{FP}).
\]  

The values used for \( p_{FP} \) and \( p_{FN} \) the probabilities of generating a false-positive and false-negative error, come directly from the sensors. The \( \alpha \) parameter used to calculate the cost of a timeshift operation is dependent on a number of variables such as changes to the agent’s path, velocity differences between the models and the actual agent, etc. Therefore, this value can and probably should be learned from empirical data.

Our dynamic programming algorithm for computing the Levenshtein distance between observation sequences under these operations appears in Algorithm 1. The approach is closely modeled after the standard algorithm.

**B. Model classification via direct voting**

Perhaps the most obvious way to use the similarity measure introduced in Section IV-A for model classification is to (1) initialize equally-sized collections of simulation traces for each of our \( m \) implicit models; (2) simulate each of these traces for the same interval of time as the true observation sequence \( Y_A \), recording the sequence of beam crossings for each; (3) compute the edit distance between \( Y_A \) and the each of these beam sequences; and (4) for each model \( M_i \), sum the edit distance values for its traces. The algorithm then selects the model with the lowest total distance as the most likely to have generated the sequence of observations \( Y_A \). Details of this approach, which we call direct voting, appear in Algorithm 2.

**C. Filtering with resampling**

Although the direct voting scheme described above works well (see Section V), in many cases it can expend computational effort performing simulate operations on traces that can be identified very early on as being a poor match for the actual observation sequence generated by the agent. To mitigate this problem, we propose a second model classification algorithm, inspired by well-known particle filtering techniques, called filtering with resampling. The intuition is to perform the simulate operations in smaller segments, and then use a randomized resampling step to eliminate traces with very large distances, in favor of forks of better-matching traces.

Pseudocode for the approach appears in Algorithm 3. After using initialize to build a set \( A \) of \( N \) traces distributed equally across the \( m \) implicit models, the algorithm alternates between two phases.

- **Simulation Phase**: the traces are simulated forward by a small, fixed time interval \( \Delta t \) and a container \( Y \) is used to store the beam crossings.

---

**Algorithm 1 ObsEditDist\((S, T)\)**

**Input:** Sequences \( S = \langle s_1, \ldots, s_m \rangle \) and \( T = \langle t_1, \ldots, t_n \rangle \)

**Output:** Cost value for the similarity between \( S \) and \( T \)

for \( i \leftarrow 0 \) to \( m \) do
  \( d[0, i] = i \cdot C_D \) end for

for \( j \leftarrow 0 \) to \( n \) do
  \( d[0, j] = j \cdot C_I \) end for

for \( i \leftarrow 1 \) to \( m \) do
  for \( j \leftarrow 1 \) to \( n \) do
    if \( b^{(s)}_i = b^{(t)}_j \) then \( \triangleright \) Same beam label
      if \( t^{(s)}_i = t^{(t)}_j \) then \( \triangleright \) Observations match
        \( \triangleright \) true positive
        \( d[i, j] \leftarrow d[i - 1, j - 1] + C_M \)
      else \( \triangleright \) Observations do not match
        \( \triangleright \) insert, delete, or timeshift
        \( ts \leftarrow d[i - 1, j - 1] + C_T\left(t^{(s)}_i, t^{(t)}_j\right) \)
        \( del \leftarrow d[i - 1, j] + C_D \)
        \( ins \leftarrow d[i, j - 1] + C_I \)
        \( d[i, j] \leftarrow \min(ts, del, ins) \)
      end if
    else \( \triangleright \) Not the same beam label
      \( \triangleright \) insert or delete
      \( del \leftarrow d[i - 1, j] + C_D \)
      \( ins \leftarrow d[i, j - 1] + C_I \)
      \( d[i, j] \leftarrow \min(del, ins) \)
    end if
  end for
end for

return \( d[m, n] \)
the model with the largest number of traces. Without reaching such a consensus, it instead outputs the traces to see if they have reached a consensus on the model for the agent. If so, this trace is used as a the output of the algorithm. If the algorithm exhausts all traces and fails to reach a consensus, it instead outputs the model with the largest number of traces.

V. SIMULATION RESULTS

We implemented this algorithm in simulation. Throughout this section, we use \( p_{FP} = 0.001 \), \( p_{FN} = 0.004 \), and \( \alpha = 0.001 \).
Algorithm 4 Resample($A, Y, Y_A$)

**Input:** Set of traces $A = A_1, \ldots, A_N$, set of observation sequences $Y$, observation sequence $Y_A = \langle y_1, \ldots, y_n \rangle$

**Output:** Set of traces $A'$, set of observations $Y'$

```
declare double $w[1, \ldots, n]$ \triangleright resampling weights
for $i \leftarrow 1$ to $N$ do
    $w[i] \leftarrow 1/\text{ObsEditDist}(Y_A, Y^{(i)})$
end for
normalize $w$ to sum to 1;
for $i \leftarrow 1$ to $N$ do
    select trace $a_j$ from $A$ with probability $w[j]$
    $a'_j \leftarrow \text{Fork}(a_j)$
    insert $a'_j$ into $A'$
    insert $Y^{(j)}$ into $Y'$
end for
return $(A', Y')$
```

- $M_7$: Remain motionless.
- $M_8$: Move randomly through the environment.
- $M_9$: Travel clockwise around the outer boundary.
- $M_{10}$: Travel counterclockwise around the outer boundary.

Both algorithms were run using $N = 1000$ simulation traces. We varied both the amount of time for which the system was able to observe the mobile agent and the resampling rate $\Delta t$ and performed 10 trials for each scenario. For each trial we recorded whether or not the algorithm correctly classified the agent’s model. Figures 4a-d show the success rate when the resampling algorithm resamples every 50, 125, 250, and 500 seconds, respectively.

Two conclusions can be drawn from these results. First, notice that the filtering with resampling algorithm does indeed outperform direct voting when the observations sequences are short. Second, we observe that, if $\Delta t$ is too small, then the resampling algorithm performs quite poorly, because the resampling steps occur before sufficiently many observations are available to assign meaningful weights.

**B. Analysis of Confusion Matrices**

To better understand the types of errors that the filtering with resampling algorithm incurs, we performed a second simulation in which we used each $M_i$ as the correct model in 50 trials, and recorded the answers returned by the algorithm. This simulation used $\Delta t = 250$, and allowed the system to observe the agent for 50000 seconds. The simulations produced the confusion matrix in Table I. The success rates of our algorithm in correctly predicting the agent’s model appear on the diagonal of this confusion matrix.

These results show that our algorithm is generally very accurate, except that model $M_7$ is often mistaken for model $M_8$. We believe that is occurs because when there are false positives in the observed sequence, they all incur a relatively high cost $C_D$ when edited into
the empty observation sequence generated by each $M_7$ trace. In contrast, editing this sequence of false positives into an $M_8$ trace may be able to use some timeshift operations, whose cost is much lower in our filter. One possible solution would be to tune the parameter $\alpha$ to more strongly penalize timeshift operations.

We also repeated this simulation using the two more complex environments depicted in Figure 5. For the environment of Figure 5a, we used 10 models similar in spirit to the ten described above. Similarly, we used six agent models for the environment in Figure 5b. Detailed descriptions of these models are omitted due to space limitations. The results of this experiment, which are similar to those for the simpler environment, appear in Tables II and III. In those tables, the motionless model appears as $M_{10}$ and $M_6$, respectively.

C. Relationship between success rate and the number of traces

Finally, we performed a simulation to measure the relationship between the algorithm’s success rate and the number $N$ of traces it uses. We used the environment in Figure 3 with $\Delta t = 250$ and observed the agent for 50000 seconds. For each $N = 10, 20, 30, \ldots, 350$, we performed ten trials and measured the success rates. The results appear in Figure 6. We observed in these results that, once a sufficiently large “critical mass” of traces are in use (in this case, around $N = 70$), the performance of the algorithm is not particularly sensitive to the number of traces.

VI. Conclusion

In this paper we introduced a notion of implicit models for agent behavior, and showed how those models can be used to classify observed agents. Both algorithms presented in this paper, direct voting and filtering with resampling use an adaptation of the Levenshtein distance to measure the similarity between two observation sequences. The training of tunable parameters that contribute to the cost calculation of an edit distance has been successfully performed in other classification problems [31], and we anticipate that similar methods may work here.

We have, of course, left several related problems unsolved. Perhaps most interestingly, we believe that our implicit models may be useful for active sensor placement, in addition to the passive inference technique described in this paper.

Moreover, we so far only consider a single agent, but practical systems of this type are quite likely to observe multiple agents simultaneously. With sensors that can only detect when an agent has crossed a beam sensor but not the identity of that agent, we will encounter a data

<table>
<thead>
<tr>
<th>$M_i$</th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
<th>$M_5$</th>
<th>$M_6$</th>
<th>$M_7$</th>
<th>$M_8$</th>
<th>$M_9$</th>
<th>$M_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>0.80</td>
<td>0.02</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_2$</td>
<td>0.00</td>
<td>0.86</td>
<td>0.00</td>
<td>0.06</td>
<td>0.08</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_3$</td>
<td>0.18</td>
<td>0.00</td>
<td>0.78</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_4$</td>
<td>0.02</td>
<td>0.10</td>
<td>0.04</td>
<td>0.82</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_5$</td>
<td>0.00</td>
<td>0.30</td>
<td>0.00</td>
<td>0.00</td>
<td>0.62</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_6$</td>
<td>0.24</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.06</td>
<td>0.68</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_7$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.02</td>
<td>0.24</td>
<td>0.70</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_8$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.12</td>
<td>0.04</td>
<td>0.06</td>
<td>0.76</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_9$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_{10}$</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>0.04</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$M_i$</th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
<th>$M_5$</th>
<th>$M_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>0.90</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_2$</td>
<td>0.00</td>
<td>0.90</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_3$</td>
<td>0.10</td>
<td>0.00</td>
<td>0.80</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
</tr>
<tr>
<td>$M_4$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.80</td>
<td>0.10</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_5$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.90</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_6$</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
<th>$M_5$</th>
<th>$M_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>0.75</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
</tr>
<tr>
<td>$M_2$</td>
<td>0.00</td>
<td>0.80</td>
<td>0.00</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>$M_3$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.60</td>
<td>0.00</td>
<td>0.10</td>
</tr>
<tr>
<td>$M_4$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.80</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>$M_5$</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>0.50</td>
<td>0.20</td>
</tr>
<tr>
<td>$M_6$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>0.90</td>
</tr>
<tr>
<td>$M_7$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>0.90</td>
</tr>
<tr>
<td>$M_8$</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.80</td>
</tr>
<tr>
<td>$M_9$</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.90</td>
</tr>
<tr>
<td>$M_{10}$</td>
<td>0.10</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table I
Confusion matrix for the environment in Figure 3.

Fig. 5. [a] An environment that resembles the layout of a city. [b] A circular hallway with many adjoining rooms.

Table II
Confusion matrix for the environment in Figure 5a.

Table III
Confusion matrix for the environment in Figure 5b.

Fig. 6. The success rate vs. the number of traces in the filter and resample algorithm.
association problem when multiple agents are present in an environment.

Finally, our current problem formulation uses beam sensors that are pairwise disjoint. To extend this work to other more interesting sensor models or to relax the constraint that the beam sensors be pairwise disjoint, we would have to deal with the case of transpositions, in addition to insertions, deletions, and time-shifts. This would require us to adapt our current algorithm (which is a generalization of the Levenshtein distance), to an algorithm that can compute the Damerau-Levenshtein distance [7], which correctly accounts for transpositions.

ACKNOWLEDGEMENT

We acknowledge support for this work from NSF (IIS-0953503).

REFERENCES