Efficiently Identifying Close Track/Observation Pairs in Continuous Timed Data

Jeremy Kubica\textsuperscript{a}, Andrew Moore\textsuperscript{a}, Andrew Connolly\textsuperscript{b}, and Robert Jedicke\textsuperscript{c}

\textsuperscript{a}Carnegie Mellon University, Robotics Institute, 5000 Forbes Avenue, Pittsburgh PA, USA;
\textsuperscript{b}Department of Physics and Astronomy, University of Pittsburgh, Pittsburgh PA, USA;
\textsuperscript{c}Institute for Astronomy, 2680 Woodlawn Drive, Honolulu HI USA

ABSTRACT

In this paper we examine new data structures and algorithms for efficient and accurate gating and identification of potential track/observation associations. Specifically, we focus on the problem of continuous timed data, where observations arrive over a range of time and each observation may have a unique time stamp. For example, the data may be a continuous stream of observations or consist of many small observed subregions. This contrasts with previous work in accelerating this task, which largely assumes that observations can be treated as arriving in batches at discrete time steps. We show that it is possible to adapt established techniques to this modified task and introduce a novel data structure for tractably dealing with very large sets of tracks. Empirically we show that these data structures provide a significant benefit in both decreased computational cost and increased accuracy when contrasted with treating the observations as if they occurred at discrete time steps.

Keywords: Data Association, Algorithms, Data Structures

1. INTRODUCTION

One vital and often computationally expensive aspect of multiple target tracking is the question of data association: given a set of current track estimates and new observations, determine which observation corresponds to which track. The determined track/observation associations are used not only to identify which target caused a given observation, but more importantly to update the estimate of a target’s position and trajectory. While there have been a vast number of proposed and developed approaches to both the general tracking and data association problems, many of these techniques include the underlying assumption that new sets of observations arrive together at discrete time steps. Below we remove this assumption and examine data structures and algorithms to quickly identify potential associations in “continuous time” data where observations may occupy a range of times. This corresponds to answering a continuous time gating query, which finds all points within some gate of the predicted track position.

Below we examine data structures and algorithms for observations that are distributed over a finite time interval. We show that approaches designed for the discrete time steps can easily be adapted to function on continuous time observations, providing the benefits of continuous time while retaining the tractability of discrete time approaches. We introduce a novel data structure for dealing with large sets of tracks in continuous time queries. Empirically we show that these data structures provide a significant benefit both in decreased computational cost and increased accuracy. Finally, we discuss these approaches as compared to “flattening” the data to discrete time steps. We show that in some cases flattening the data to discrete time steps can lead to a loss in accuracy and an increase in computational cost.

The approaches we present are designed to be general and applicable to a wide range of tracking algorithms and methodologies, but our primary motivating example in this paper is asteroid tracking. Here we wish to determine which observed objects correspond to the same true underlying object from a series of visual observations of the night sky. These linkages can then be used to determine tentative orbits, attribute the observations to a known orbit, and assess potential
risk of an asteroid. This domain is particularly interesting for several reasons. First, the observations arrive in many small batches spread out over an entire viewing period. Each image covers only a small fraction of the night sky, and thus objects are observed over a range of times. We show that this type of structure is particularly well suited to benefit from the approaches that we present. Second, the asteroid tracking problem is a large-scale multiple target tracking problem. In particular, the algorithms and data structures discussed in this paper are being developed with respect to the problem of asteroid tracking in the next generation of large sky surveys, such as PanSTARRS and LSST. Here the high numbers of true objects and additional noise observations make exhaustive techniques effectively intractable.

2. PROBLEM DESCRIPTION

In this paper we examine an important aspect of the data association problem, identifying feasible track/observation associations. This problem is inherently a spatial query, finding all observations that lie “near” the predicted track positions. We refer to this problem as spatial data association to indicate its role within the full data association problem. The data consists of two different sets: observations and tracks.

**Observations:** Observations are $D$-dimensional points that may correspond to a true object or be the result of noise. We denote the $i$th observation as $\mathbf{x}_i$ and indicate the time of observation as $t_i$. We use $N_x$ to denote the number of observations.

**Tracks:** Tracks represent the estimated target motion. We allow a general definition of a track as any function of the independent variable through the $D$-dimensional space. We denote the $j$th track as $\mathbf{g}_j(t)$ and use $N_T$ to denote the number of tracks. While much of our discussion below applies to a range of track models, we primarily restrict the discussion to the linear and quadratic models to keep the discussion simple and consistent.

Spatial data association is identical to gating, where observations are filtered by whether they fall within a window or gate around the track’s predicted position. Formally we can phrase the problem as a search query. Given sets of tracks $\mathbf{G}$ and observations $\mathbf{X}$, find all track/observation pairs such that the observation is “close” to the track’s predicted position:

$$\text{dist} (\mathbf{g}_j(t_i), \mathbf{x}_i) < \epsilon$$

where $\text{dist}$ is the given distance measure. This measure may be simple, such as Euclidean distance, or may depend on the accuracy of the estimate of the track’s parameters. For example, Kalman filters provide a covariance estimate that can be used to define a ellipsoidal gate.

It is important to note that we use the notation $\mathbf{x}_i[d]$ to denote the $i$th vector’s value in the $d$th dimension. Since $\mathbf{g}(t)$ is a $D$-dimensional vector, for consistency we denote the $j$th track’s position in the $d$th dimension at time $t$ as $\mathbf{g}_j(t)[d]$.

2.1. Continuous Time Data

In many real world applications, the observations may not arrive together at discrete time steps. Instead they may be spread over an interval of time or reported in small batches. For example, distributed sensors may report observations at different times or on different cadences. We call such systems continuous time systems. We consider two different types of continuous time systems:

1. **Fully Continuous** - In a fully continuous time system, each observation may occur at any real-valued time within some interval. Thus no two observations may share exactly the same time.

2. **Semi-Continuous** - In a semi-continuous time system, observations arrive in discrete batches. However, each batch may contain observations for only a fraction of the objects.

For the most part we are interested in semi-continuous systems as they arise in many real-world systems. Astronomical observations are one particular semi-continuous time system. Each image of the night sky only contains a small fraction of the objects of interest and thus a full set of observations may be spread out over a significant period of time.

We can partition the observations’ time steps into two natural granularities: viewings and fields. A field corresponds to a set of observations that are made (nearly) simultaneously ($t_{\text{field}} - t_e \leq t \leq t_{\text{field}} + t_e$), but may only cover a limited region of the space. In the semi-continuous case $t_e = 0$ and in the fully continuous case $t_e > 0$. In contrast, a viewing consists of a set of such fields, covering an extended region of both space and time. Although the viewings do not need to cover the entire observation space, below we use the term to denote a set of fields that do cover the entire observation space. In this respect a single viewing functions as a time step containing observations from the entire observation space.
Flattening an observation (solid circle) to a discrete time step \( t_0 \) (shaded circle) introduces an error between the observed position and the predicted position (open circle).

### 2.2. Flattening

A natural way to deal with (semi-)continuous data is simply to flatten it onto discrete time steps. Specifically, we choose a set of discrete time steps \( T \) and treat each observation as though it occurred on the closest time step:

\[
t_i = \text{ARGMIN}_{t_i \in T} |t - t_i|
\]

Depending on the application we can flatten the times into different granularities or even adaptively choose the time steps.

It is important to note that flattening may introduce systematic noise into the data, by ignoring dynamics of the underlying track. Such an error is shown in Figure 1. However, for many tasks this issue will be negligible, as the shift in time and thus the systematic noise will be relatively small. Further it is possible that we can compensate for any error by simply increasing the size of the gate or augmenting the gate to account for potential track motion.

Even in cases where it may be possible to safely flatten the data, we may be able to make significant computational savings by treating the data as continuous. Specifically, by treating the data as continuous we may be able to examine many fields at once and reduce redundant work. This is especially true if each field only covers a small fraction of the targets. This benefit also applies to semi-continuous data and is thus applicable to many real-world domains.

### 3. DATA STRUCTURES FOR EFFICIENT SPATIAL DATA ASSOCIATION

In order to reduce the computational cost of spatial data association, we must limit the number of track/observation pairs that must be tested. A simple brute force approach to the problem would explicitly test all pairs, requiring \( N_T \cdot N_T \) distance calculations. However, by exploiting spatial structure within the data we skip many obviously bad candidate pairs.

#### 3.1. Tree-based Approaches with Discrete Time Steps

A common and effective approach to making spatial data association tractable is to use a tree-based data structure, such as a kd-tree, to efficiently find potential associations at each time step. For example, Uhlmann describes the use of kd-trees to accelerate spatial data association in large-scale multiple target tracking. Below we provide a very brief overview of kd-trees and their use for fast proximity queries.

KD-trees are hierarchical data structures that partition space by recursively splitting it along axis-aligned hyper-planes. Each node in the tree represents a region of the entire space and (either explicitly or implicitly) a set of data points. The hierarchical structure means that we can represent both the region and set of data points owned by an internal node implicitly as the union of its childrens’ regions and data points. An example kd-tree is shown in Figure 2.

The hierarchical structure of the tree-based data structures makes spatial queries very efficient. Consider the problem of finding all points that fall within some radius \( r \) of a given query point \( q \). We simply descend the tree in a depth first manner and look for data points within \( r \) of \( q \). If we ever reach a node with no children, called a leaf node, we explicitly test the data points owned by that node to determine if their distance with \( q \) is less than \( r \). If so, we add them to our list
of results. However, we can exploit the spatial structure to stop exploring a branch of the tree if we find that no point contained in that branch could fall within our search radius. For example, in Figure 2.C we can prune the subtree at node 8 because the entire node falls outside of our search radius. Thus we do not have to explore any of node 8’s children or test their associated points. In general, the ability to prune out infeasible regions of the search space can provide significant computational savings.

In the case of spatial data association we can construct a tree from the predicted track positions at each time step and use the new observations at that time step as query points. However, if the observations do not arrive in discrete time steps, this approach may become less efficient. At the extreme, fully continuous data would require building a new tree for each individual observation, providing no computational advantage. One possibility is to flatten the observations onto discrete time steps introducing some systematic error. However, by developing approaches that work directly on the continuous time data, we can preserve the data’s accuracy while possibly increasing computational efficiency.

3.2. KD-trees on Points with Continuous Times

We can adapt the tree-based spatial data structures to explicitly handle observations with a range of times. As a consequence we must also adapt the spatial queries themselves to work with respect to moving points instead of just stationary ones. In doing so we can efficiently ask whether any observations fall near a query trajectory over some time interval of interest.

We build a single kd-tree that allows us to consider a range of times by augmenting the data points to include time as a dimension. Thus we build a tree on the $D+1$ dimensional data. As shown in Figure 3.A, this allows us to partition and index the data both spatial and temporally. Like a standard kd-tree, the augmented kd-tree is constructed in a recursive top-down fashion. At each level, the current set of observations is split into two subsets that are recursively used to build the children nodes. The partitioning is done by choosing the widest dimension of the node’s bounding box and splitting on the midpoint of that dimension. Nodes are assigned to the left child if they fall below that threshold and the right child if they fall above it. When comparing the relative widths of the time and space dimensions we must weight them accordingly. In the experiments below we used the heuristic approach of weighting the dimensions by their initial width, thus splitting on the dimension whose width was the largest percentage of its initial width.

The search for potential track/observation associations then becomes a question of proximity to a curve. This search is shown in Figure 4. For a given query track $g(t)$ we again traverse the tree in a depth first search. If we hit a leaf node, we explicitly test whether the observations at that node are close to the track. However, we can prune the search if we ever find that the track cannot hit any observation contained within the bounds of the node. As shown in Figure 3.B, we can determine whether this criteria is met by taking the node’s bounding box of the original $D$ dimensions, extending it on each side by the threshold amount $\varepsilon$, and testing for intersection (Figure 4, Line 1). If the track does not intersect the box within the node’s time bounds, we can safely prune the search. For simple motion models, such as linear or quadratic, this test can be performed very inexpensively.
Figure 3. We can add time as a dimension to the kd-trees, partitioning the data in both space and time (A). When pruning the augmented kd-tree we must ask whether the query trajectory comes within the threshold $\varepsilon$ of the tree node during the time frame that the node covers (B). Tracks $a$ and $b$ pass nearby the node while track $c$ does not.

<table>
<thead>
<tr>
<th>Observation KD-Tree Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Query track $g$ and kd-tree node $T$.</td>
</tr>
<tr>
<td><strong>Output</strong>: List $L$ of compatible observations.</td>
</tr>
<tr>
<td>1. <strong>IF</strong> $g(t)$ is within $\varepsilon$ of $T$'s bounding box for any $t$ in $T$’s time bounds.</td>
</tr>
<tr>
<td>2. <strong>IF</strong> $T$ is a leaf:</td>
</tr>
<tr>
<td>3. <strong>ELSE</strong></td>
</tr>
<tr>
<td>4. Test $g$ against all observations in $T$ adding compatible observations to $L$.</td>
</tr>
<tr>
<td>5. Recursively search using $T$’s right child.</td>
</tr>
<tr>
<td>6. Recursively search using $T$’s left child.</td>
</tr>
</tbody>
</table>

Figure 4. The recursive observation based kd-tree search.

3.3. Track-based Data Structures

While partitioning the observations with a tree-based data structure provides a potentially significant reduction in computational cost, it requires that we have enough data points to allow sufficient pruning. If the data is arriving in small batches, then building a tree on the new observations may not provide any computational advantage. One alternative is just to reverse the approach and construct a kd-tree on the predicted track positions. Then each new observation can quickly search the track-tree to find any nearby tracks. However, if the number of incoming observations is small then many of the track predictions will not have a corresponding observation and thus will be “wasted.” For example, although we will have a large set of current tracks in the astronomy domain, each image will include only a limited number of observations.

Instead of constructing a data structure over static positions, we can construct a data structure from the tracks’ actual trajectories over some interval of time. We again use a tree-based structure that partitions the tracks into coherent bundles according to how they move over the time interval of interest. To this end, we use an approach similar to that of a ball-tree or metric tree.\textsuperscript{4,5} We call the resulting structure a \textit{track-based ball-tree}.

The track-based ball-tree uses a tight coupling between the track-to-track distance measure and tree construction. This distance measure is used to both define the nodes’ bounds and to partition the tracks. Because we want coherent bundles of tracks that can be easily pruned, we define the distance between two tracks so as to provide a maximum bound on their separation during the time interval of interest. We can then use this distance to define the node’s bounds. Specifically, the bounds of a node are defined with reference to a central anchor track, $g_a$, and a node radius $r$. The anchor track serves to indicate one possible predicted position of tracks in the node and the radius indicates the maximum offset from this prediction in each dimension. The radius is defined as the smallest $r$ such that:
The track ball-tree nodes are defined in reference to an anchor track \( g_a(t) \) and radius \( r \). The radius indicates the maximum distance from the anchor track in any dimension.

Figure 5. The track ball-tree nodes are defined in reference to an anchor track \( g_a(t) \) and radius \( r \). The radius indicates the maximum distance from the anchor track in any dimension.

\[
|g_a(t)[d] - g(t)[d]| \leq r[d] \quad \forall t_s \leq t \leq t_e \quad \forall g \in \text{node} \quad \forall d
\]  

(3)

where \( t_s \) and \( t_e \) indicate the time interval of interest. Thus at any time \( t \in [t_s, t_e] \) all of the tracks in the node are guaranteed to be at most \( r[d] \) from the anchor track in the \( d \)th dimension. Figure 5 illustrates these bounds. Since we want the anchor track to be central to the node, we use an artificial track created by taking the average parameters of the node’s tracks.

We construct the track-based ball-tree by recursively partitioning the tracks. As shown in Figure 6, at each level we split the tracks into two sets by choosing two well separated tracks and dividing the remaining tracks based on their proximity to those tracks. Formally, we term the two tracks used for splitting \( g_R \) and \( g_L \). We assign a track \( g_i \) to the node’s right child if and only if its distance to \( g_R \) is smaller than its distance to \( g_L \). This partitions the tracks into increasingly coherent bundles. Since we want to form two tight bundles of tracks at the next level, we use a distance function similar to the one that will determine the bounds, the maximum distance between two tracks on a given time interval \( [t_s, t_e] \) with respect to some dimension \( d^* \):

\[
\Delta(g_i, g_j) = \text{MAX}_{t_s \leq t \leq t_e} \left| g_i(t)[d^*] - g_j(t)[d^*] \right|
\]  

(4)

The dimension \( d^* \) is chosen to be the widest, and thus loosest, dimension. The splitting process is illustrated in Figure 6.

Figure 6. A set of tracks is split by choosing two temporary anchor tracks (A) and for each other track calculating which anchor is closer. The resulting partition is shown in (C) and (D).

In the experiments below, \( g_R \) and \( g_L \) are chosen by: picking a random track \( g_i \), finding \( g_R \) as the furthest track from \( g_i \), and finding \( g_L \) as the furthest track from \( g_R \).

The search for track/observation pairs follows the approach given Figure 7. Given a candidate observation \( x \) we do a depth first search of the tree. We prune any node if we can determine that no track owned by this node could be close to \( x \). We use a conservative test that considers each dimension independently. Specifically, for each dimension \( d \) we determine whether any track in the node could come near \( x \) by testing if the anchor track comes within distance \( r[d] + \varepsilon[d] \) of \( x \) (Figure 7, Line 3).
Track-Based Ball-Tree Search

**Input:** Query observation $x$ and track tree node $T$.

**Output:** List $L$ of compatible tracks.

1. prune ← FALSE
2. FOR $d$ = 1 to $D$
3. IF $|x[d] - g_d(t_{obs})[d]| > \varepsilon[d] + r[d]$
4. prune ← TRUE
5. IF prune == FALSE:
6. IF $T$ is a leaf:
7. Test $x$ against all tracks in $T$ adding compatible tracks to $L$.
8. ELSE
9. Recursively search using $T$'s right child.
10. Recursively search using $T$'s left child.

Figure 7. The track-based tree search. Each dimension is tested independently to determine if we can prune.

### 4. EXPERIMENTS

To test the performance of the algorithms we simulated data from quadratic tracks. The use of simulated data allows us to conduct controlled experiments on the effect of various parameters, such as the number of fields per viewing. This also allows us to determine if the algorithms scale up to a large number of targets, an absolute necessity for such domains as the next generation of sky surveys.

#### 4.1. Tracking Algorithm

In the below experiments all tracking is done using Kalman filters with independent dimensions. Thus the track estimate consists of $D$ separate 3-dimensional state vectors (position, velocity, and acceleration) and $D$ separate covariance $3 \times 3$ matrices. All tracks were started with the correct state estimate.

We used a very simple form of multiple hypothesis tracking. Specifically, we create a new set of tracks at each viewing period that consists of those tracks from the previous time steps that were extended with a new associated observation. The previous set is then discarded, making the extended tracks our current set of track estimates. To generate the new set, each track from the previous time step is associated with new observations. The criterion for a valid match is threshold distance of $\varepsilon$ in each dimension:

$$|x[d] - g(t_{obs})[d]| \leq \varepsilon[d]$$

(5)

Each time a track/observation pair produces a viable match, a new track is created from the pair and added to the new set of tracks. Thus one track can be associated with multiple observations and form multiple new tracks. Similarly, one observation may be associated with multiple tracks.

The above tracking algorithm results in very simple track maintenance logic. New tracks are never created from singleton observations. Rather tracks are always the result of extending one of the previous tracks. Further, tracks are pruned if they are not associated with any observation during the current viewing period. While the above tracking algorithm is very simplistic, it has the desirable quality that it returns all possible tracks that are compatible with the initial track estimates and the threshold. The lack of combinatorial data association removes any inherent pruning that might incorrectly remove a true track.

Finally, it should be noted that the data structures and algorithms were developed independently of the exact track logic used. While the track logic discussed above is very simplistic, it is meant only as an illustrative example. The data structures and algorithms themselves are applicable to a wide range of tracking approaches.

#### 4.2. Effect of Flattening

The first experiment was designed to demonstrate the potential effects of flattening on continuous time data. In many domains the amount of flattening may be small and thus the errors may be slight. However, it is interesting to look at cases where the flattening is significant. Such a case could occur in the asteroid tracking domain if each night was treated as a single time step and all observations for that night were flattened back to one time.
Table 1. The performance of the multiple hypothesis tracker for both flattened and non-flattened data as the range of observation times increases. The columns indicate percent correct (P_C), percent found (P_F), and weighted number incorrect (WI).

<table>
<thead>
<tr>
<th>Δt</th>
<th>Flattened</th>
<th>Non-Flattened</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC</td>
<td>PF</td>
</tr>
<tr>
<td>0.0</td>
<td>0.009</td>
<td>0.992</td>
</tr>
<tr>
<td>0.2</td>
<td>0.009</td>
<td>0.992</td>
</tr>
<tr>
<td>0.4</td>
<td>0.009</td>
<td>0.989</td>
</tr>
<tr>
<td>0.8</td>
<td>0.009</td>
<td>0.975</td>
</tr>
</tbody>
</table>

4.2.1. Experiment

The data consisted of continuous time observations from 100 different, random, quadratic tracks. The observations were generated at 5 viewings, which were spaced 1 time unit apart. The viewings widths’, Δt, were varied from 0.0 to 0.8 time units. Thus the data ranged from flat to nearly continuous. Each track was seen exactly once during each viewing, with a time of observation that was chosen randomly over the time interval of the viewing.

We tested the accuracy of the returned tracks under the two different approaches: flattening the data and using the continuous data. Further, we examined how this accuracy changed as we varied the width of the viewing. The gating threshold was fixed apriori and was constant for all tracks and viewings. The performance of the algorithms was monitored by three different measures:

1. **Percent Correct (P_C)** - The percentage of returned tracks that exactly matched at least one true underlying track. This score measured the false positive rate.

2. **Percent Found (P_F)** - The percentage of true underlying tracks that exactly matched at least one of the returned tracks. This score measured the false negative rate.

3. **Weighted Number Incorrect (WI)** - The weighted number of incorrect tracks (false positives) that have a better score than at least 10% of the true tracks. The weight of each track is determined by the number of true tracks the incorrect track’s score exceeds. This score was calculated by sorting the returned tracks and traversing the list from best to worst according to their probability under the Kalman filter. At each step, the total score is incremented by the number of incorrect tracks seen so far. The counting stops after 90% of the true tracks have been seen. This score accounts for both the number and position of false positives.

These measures give a snapshot of algorithm performance for one setting of the gating threshold. It is worth noting that we could trade off percentage correct and percentage found by varying that threshold.

4.2.2. Results and Discussion

The results, in Table 1, illustrate the danger of flattening the observations. At a field width of Δt = 0.0 the data is already flat and both approaches produce the same results. As the field width increased, over twice as many tracks were missed when flattening the data as compared to keeping the continuous times. Further, the weighted error increased by over a factor of 2, indicating that the tracker was returning more highly ranked false positives.

The above experiment shows the (exaggerated) dangers of flattening. While in many cases the amount of flattening may be slight, it is important to consider whether systematic errors are being introduced.

4.3. Effectiveness of Tree Structures

The second experiment was designed to evaluate the computational performance of the various approaches and data structures. Even in cases where flattening may be a safe option, the use of continuous time data structures may still provide computational benefits.
Each of the 4 viewings is divided into 5 fields that tile the observed space.

4.3.1. Experiment
To test the efficiency of our proposed algorithms, we examined their relative performance as the number of fields was varied. The number of viewings was held constant and the number of fields $K$ was varied from 5 to 5000. We used one viewing per time unit, with each viewing covering a time range of 0.5 time units, providing a gap of 0.5 time units between viewings. The fields were equally spaced over this time span and formed a disjoint partitioning of the sky. The actual observations were generated from either 5000 or 10000 random, quadratic tracks. Each object appeared in exactly one field per viewing. As illustrated by the simple 1-dimensional example in Figure 8 the fields provided spatial structure, containing those observations that were in a given region of space at a given time.

The performance was measured by the computational cost of performing the multiple hypothesis tracking. Specifically, we looked at three measures:

1. The number of distance computations.
2. The number of pruning queries during tree traversal.
3. The wall-clock running time (including data structure construction).

The first two measures indicate processor independent cost, but do not account for time required to build data structures. In contrast, the wall-clock running time indicates the total computational cost, but is processor dependent and may be reduced by further optimizations, such as reordering the data for cache friendliness.

4.3.2. Data Structures
We tested both data structures and an exhaustive search:

- **Exhaustive** - At each time step, test all track/observation pairs. This approach is denoted $E$ in the results.
- **Observation-based kd-tree** - Construct a kd-tree of observations at each time step (either field or viewing) and use this kd-tree to quickly find candidate observations for each query track. This approach is denoted $T_O$ in the results.
- **Track-based Ball-tree** - Construct a track-based ball-tree on the tracks at the beginning of each viewing. Use this tree to quickly find candidate tracks for each query observation. This approach is denoted $T_B$ in the results.

In addition, we examined two different strategies for when to perform the data association:

- **Viewing Based** - Perform the data association after each viewing. Thus we wait and perform the data association using all fields in the viewing at the same time. This approach is denoted with a superscript $V$. 
The two different strategies will have the most significant effect on the observation-based kd-tree. Under the viewing-based approach one kd-tree is built on all observations of each viewing. In contrast, under the field-based approach one kd-tree is built for each field, meaning that we construct $K$ small trees instead of one large tree.

Note that despite the use of two different sized time steps for data association, the track maintenance and filtering is only performed after each viewing. In other words, even with a field-based strategy, the set of tracks given at the beginning of each viewing is built for each field, meaning that we construct $K$ small trees instead of one large tree.

The results, presented in Table 2, indicate several very important trends when dealing with data with a high number of fields. First, the results demonstrate the sheer intractability of an exhaustive approach to spatial data association. While this in itself is well established, when combined with continuous or semi-continuous time data it illustrates the potential need for algorithms that use the spatial structure of the data while preserving the continuous time aspect. Second, the results demonstrate clear advantages to using the above continuous time approaches on semi-continuous time and discrete time data. Here conventional wisdom might suggest that we build one kd-tree on the observations from each field separately. This corresponds to building a new data structure at each time step. In general, this would require very little flattening and allow us to harness the spatial structure of the data. However, the results indicate that it may be more efficient to treat the data as having continuous times and use the above approaches.

If we are interested in performing the spatial data association only once per viewing, then we can achieve a significant computational advantage by treating the semi-continuous time data as fully continuous and building a single tree of observations for all fields in the viewing. The results for $T^F_O$ indicate the performance we get if we build a data structure on the observations from each field separately. This corresponds to building a new data structure at each time step. In contrast, the results for $T^E_O$ indicate the performance we get if we build a data structure on all observations from the viewing and use this one data structure for all of the query tracks. As shown in Table 2, both the running times and required number of pruning queries are drastically lower when considering all of the observations in the viewing at once ($T^E_O$) as compared to constructing new data structures at each time ($T^F_O$).

Further, if we are interested in performing spatial data association at each field (such as for an online tracking application), then it may be computationally advantageous to use a track-based structure. The track-based structures are

<table>
<thead>
<tr>
<th>Data</th>
<th>Running Time (sec)</th>
<th>Dist. Calculations</th>
<th>Pruning Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$K$</td>
<td>$E^v$</td>
<td>$E^f$</td>
</tr>
<tr>
<td>5000</td>
<td>5</td>
<td>39.1</td>
<td>22.5</td>
</tr>
<tr>
<td>5000</td>
<td>10</td>
<td>39.7</td>
<td>21.9</td>
</tr>
<tr>
<td>5000</td>
<td>50</td>
<td>40.3</td>
<td>22.4</td>
</tr>
<tr>
<td>5000</td>
<td>100</td>
<td>41.5</td>
<td>23.8</td>
</tr>
<tr>
<td>5000</td>
<td>500</td>
<td>47.7</td>
<td>36.3</td>
</tr>
<tr>
<td>10000</td>
<td>5</td>
<td>190.2</td>
<td>110.2</td>
</tr>
<tr>
<td>10000</td>
<td>10</td>
<td>193.1</td>
<td>89.4</td>
</tr>
<tr>
<td>10000</td>
<td>50</td>
<td>194.7</td>
<td>86.2</td>
</tr>
<tr>
<td>10000</td>
<td>100</td>
<td>197.9</td>
<td>88.7</td>
</tr>
<tr>
<td>10000</td>
<td>500</td>
<td>215.6</td>
<td>116.1</td>
</tr>
</tbody>
</table>

Table 2. The average performance of the multiple hypothesis trackers over different numbers of observations and fields. The table is broken into four parts which show the data set size, the average running time (in seconds), the average number of track/point distance computations (in millions) and the average number of pruning queries (in millions). Note that results by both viewing and field are shown only once for algorithms that are insensitive to the order in which the data is presented. Also note that the exhaustive search never prunes and thus incurs 0 pruning queries on all data sets.

- **Field Based** - Perform the data association after each field. This approach corresponds to an “online” tracking approach where we process the new observations as they arrive. This approach is denoted with a superscript $F$. 

The results, presented in Table 2, indicate several very important trends when dealing with data with a high number of fields. First, the results demonstrate the sheer intractability of an exhaustive approach to spatial data association. While this in itself is well established, when combined with continuous or semi-continuous time data it illustrates the potential need for algorithms that use the spatial structure of the data while preserving the continuous time aspect. Second, the results demonstrate clear advantages to using the above continuous time approaches on semi-continuous time and discrete time data. Here conventional wisdom might suggest that we build one kd-tree on the observations from each field separately. This corresponds to building a new data structure at each time step. In general, this would require very little flattening and allow us to harness the spatial structure of the data. However, the results indicate that it may be more efficient to treat the data as having continuous times and use the above approaches.

4.3.3. Results and Discussion

The results, presented in Table 2, indicate several very important trends when dealing with data with a high number of fields. First, the results demonstrate the sheer intractability of an exhaustive approach to spatial data association. While this in itself is well established, when combined with continuous or semi-continuous time data it illustrates the potential need for algorithms that use the spatial structure of the data while preserving the continuous time aspect. Second, the results demonstrate clear advantages to using the above continuous time approaches on semi-continuous time and discrete time data. Here conventional wisdom might suggest that we build one kd-tree on the observations from each field separately. This corresponds to building a new data structure at each time step. In general, this would require very little flattening and allow us to harness the spatial structure of the data. However, the results indicate that it may be more efficient to treat the data as having continuous times and use the above approaches.

If we are interested in performing the spatial data association only once per viewing, then we can achieve a significant computational advantage by treating the semi-continuous time data as fully continuous and building a single tree of observations for all fields in the viewing. The results for $T^F_O$ indicate the performance we get if we build a data structure on the observations from each field separately. This corresponds to building a new data structure at each time step. In contrast, the results for $T^E_O$ indicate the performance we get if we build a data structure on all observations from the viewing and use this one data structure for all of the query tracks. As shown in Table 2, both the running times and required number of pruning queries are drastically lower when considering all of the observations in the viewing at once ($T^E_O$) as compared to constructing new data structures at each time ($T^F_O$).

Further, if we are interested in performing spatial data association at each field (such as for an online tracking application), then it may be computationally advantageous to use a track-based structure. The track-based structures are
constructed on the larger data set and do not need to be completely rebuilt for each new field. Again, the results for $T^F_B$ indicate the performance we get if we build a data structure on the observations from each field. In contrast, the results for $T^R_B$ indicate the performance we get if we construct a tree data structure on the track and use the new observations arriving at each field as queries. As shown in Table 2, the performance of the single track-based data structure ($T^F_B$) is superior to that of the multiple observation based data structures ($T^R_B$). For example, with 10,000 objects and 500 fields, we see a 7.6 fold performance increase by using the track-based structures.

Finally, it should be noted that the above experiments provided an **optimistic** view of data on different fields. Because each field only contained observations from the same region, these fields themselves contained a significant amount of spatial structure. When a single kd-tree was built on each field, the trees had the advantage that a significant number of tracks could be pruned at the **top level**.

**5. MULTIPLE TRACK OBSERVATIONS PER VIEWING**

One natural extension to this work is domains in which we may **see multiple** observations of the same target during single viewing. We can easily account for this possibility by returning a **sorted** list of observations that could feasibly match a given track. These observations are sorted in time. We can then perform exhaustive multiple hypothesis tracking on this reduced set of observations. Note that while this extension is simple for spatial data association, it may lead to significantly more complicated track filtering and maintenance logic.

**6. RELATED WORK**

There have been several approaches suggested to performing spatial data association with continuous time data. Uhlmann proposes two different approaches for handling (semi-)continuous data consisting of batches of observations arriving from components of a distributed missile tracking system. In the first approach, viewings are grouped into batches covering a given time interval and the tracks are projected to the middle time of the batch. The tracks’ gates are extended to account for the maximum potential movement during this time interval. Spatial data association is done by finding all observations in the batch that fall within the extended gate. In the second approach, the tracks’ gates are again extended to account for motion, but this time accounting for the motion model. A tree can then be constructed on these gates and the observations used as queries. Since the gates will only be valid for a given time span, elements in the tree must be maintained. Neither of these approaches takes full advantage of the temporal aspects of the data. They use a single gate over a range of time. In contrast, our approaches effectively use a gate that moves with the track’s motion model.

The use of tree-based structures on regions and points to accelerate trajectory based queries has also been used in other fields. For example, in the computer graphics problem of ray-tracing, placing objects in tree structures is a common and successful approach. For example, polygons can be placed in a tree structure that is used by the query rays in order to determine ray/polygon intersection. Our temporal kd-trees are similar to such approaches, but are extended to make use of the independent variable and work with nonlinear tracks.

The approach of building data structures on tracks has also been considered in a variety of domains. Arvo and Kirk proposed **ray classification**, a technique to accelerate computer ray tracing. Rays are represented as points in 5-dimensional parameter space and partitioned into different groups. A similar technique has been presented in databases to answer queries about moving objects. Again, the linear tracks are effectively treated as points in parameter space for the construction of a tree structure. By bounding the lines’ parameters, it is possible to create time parameterized bounds of where that set of tracks could be for all future times. In previous work we tested a similar method, extended to more complex track models, to provide comparison with our ball-tree based data structure. Our experiments indicated that this parameter space approach was not as effective as our ball-tree based data structure. Finally, Pfoser et. al. presented two tree models for querying piecewise linear tracks. These structures exploited the fact that 2-dimensional tracks could be broken into line segments and treated as a set of 3-dimensional objects.

**7. CONCLUSIONS**

We examined the problem of spatial data association on continuous time data. We examined the approach of flattening and showed that a “large” change in time can lead to significant error and thus reduce accuracy. Combined with the intractability of exhaustive methods in large domains, these results indicate the need to adapt spatial structure algorithms
to continuous time data. Further, the use of continuous time methods can lead to an improvement in performance, even on some discrete time data.

We presented two algorithms designed to work on continuous time data: a track-based ball-tree and an observation-base kd-tree. We showed that both of these algorithms could lead to significant computational savings on continuous time data without adding systematic noise. Finally, we showed that it may also be advantageous to use these approaches on discrete time data, such as observations with many fields. The continuous time data structures allow us to exploit the track's movement through time and thus prune regions of space over entire spans of time, reducing computations from performing those prunings at each individual time.

Finally, it should be noted that techniques we presented are general with respect to the overall tracking algorithms. The approaches are designed to efficiently propose potential track/observation associations and thus can be used within any system that requires this subtask.

Acknowledgements

Jeremy Kubica is supported by a grant from the Fannie and John Hertz Foundation. This project was supported by a grant from the National Science Foundation (Grant CCF-0121671).

REFERENCES