Variable KD-Tree Algorithms for Efficient Spatial Pattern Search

Jeremy Kubica Joseph Masiero Andrew Moore
Robert Jedicke Andrew Connolly

CMU-RI-TR-05-43

September 2005

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

In this paper we consider the problem of finding sets of points that conform to a given underlying model from within a dense, noisy set of observations. This problem is motivated by the task of efficiently linking faint asteroid detections, but is applicable to a range of spatial queries. We survey current tree-based approaches, showing a trade-off exists between single tree and multiple tree algorithms. To this end, we present a new type of multiple tree algorithm that uses a variable number of trees to exploit the advantages of both approaches. We empirically show that this algorithm performs well using both simulated and astronomical data.

Contents

1 Introduction

Consider the problem of detecting faint asteroids from a series of images collected on a single night. Inherently, the problem is simply one of connect-the-dots. Over a single night we can treat the asteroid's motion as linear, so we want to find detections that, up to observational errors, lie along a line. However, as we consider very faint objects, several difficulties arise. First, objects near our brightness threshold may oscillate around this threshold, blinking into and out of our images and providing only a small number of actual detections. Second, as we lower our detection threshold we will begin to pick up more spurious noise points. As we look for really dim objects, the number of noise points greatly increases and swamps the number of detections of real objects.

The above problem is one example of a model based spatial search. The goal is to identify sets of points that fit some given underlying model or class of models. Although the discussion above uses a simple linear model, this general task encompasses a wide range of real-world problems including: data mining, computational statistics, computer vision, and pattern recognition. For example, we may want to detect a specific configuration of corner points in an image or search for a known type of multi-way structure in scientific data. We focus our discussion on problems that have a high density of both true and noise points, but which may have only a few points from the actual model of interest. Returning to the asteroid linking example, this corresponds to finding a handful of detections that lie along a line within a data set of millions of total detections.

Below we survey several tree-based approaches for efficiently solving this problem. We show that both single tree and conventional multiple tree algorithms can be inefficient and that a distinct trade-off exists between these two approaches. To this end, we propose a new type of multiple tree algorithm that uses a *variable* number of trees to achieve some of the benefits of both approaches. We empirically show that this algorithm performs well using both simulated and real-world data.

2 Problem Definition

Our problem consists of finding sets of points that fit a given underlying model or class of models. In doing so, we are effectively looking for known types of structure buried within the data. In general, we are interested in finding sets with k or more points, thus providing a sufficient amount of support to confirm the discovery. Finding this structure may either be our end goal, such as in asteroid linkage, or may just be a preprocessor for a more sophisticated statistical test, such as renewal strings [1]. We are particularly interested in high-density, low-support domains. In other words, we are considering problems where there may be many hundreds of thousands of points, but only a handful actually support our model.

Formally, the data consists of N unique D-dimensional points. We assume that the underlying model has c free parameters that can be estimated from c' unique points. Since $k \geq c'$, the model may over-constrained. In these cases we divide the points into two sets: *Model Points* and *Support Points*. Model points are the c' points used to fully

define the underlying model. Support points are the remaining points used to confirm the model. For example, if we are searching for sets of k linear points, we could use a set's endpoints as model points and treat the middle $k-2$ as support points.

The prototypical example used throughout this paper is the intra-night asteroid linkage problem. Since the points contain a temporal component, we treat this information separately. Formally, we define this problem as:

> For each pair of points find the $k-2$ best support points for the line that they define (such that we use at most one point at each time step).

In addition, we place restrictions on the validity of the initial pairs by providing velocity bounds. It is important to note that although we use line discovery as a running example, the techniques described below can be applied to a large range of spatial problems.

3 Overview of Previous Approaches

3.1 Constructive Algorithms

A wide range of algorithms exist that constructively answer the model matching problem. These approaches "build up" solutions starting from individual points and repeatedly finding new model or support points. Below we briefly discuss several of these approaches.

Perhaps the simplest approach is to perform a two-tiered brute force search. First, we exhaustively test all sets of c' points to determine if they define a valid model. Then, for each valid set we test all of the remaining points for support. For example in the asteroid linkage problem, we can initially search over all $O(N^2)$ pairs of points and for each of the resulting lines test all $O(N)$ points to determine if they support that line. A similar approach within the domain of target tracking is sequential tracking (for a good introduction see [3]). The points at early time steps are used to estimate a track that is then projected to later time steps and associated with new compatible points. We denote these general types of constructive approaches that do not use spatial structure as *brute force* algorithms.

These approaches can be improved by using spatial structure in the data. Again returning to our asteroid example, we can initially place the points in a KD-tree. We can then limit the number of initial pairs examined by using this tree to find points compatible with our velocity constraints. Further, we can use the KD-tree to only search for support points in localized regions around the line, pruning away large numbers of obviously infeasible points. Similarly, KD-trees have been used in tracking algorithms to efficiently find points near predicted track positions [4]. We call these adaptations *single tree* algorithms.

A significant drawback of the constructive approach is that we do not use information from all aspects of the problem at the same time. As we search for the next point to add to our set, we are, at most, using spatial structure within current candidate points under consideration. For this reason we may spend a significant amount of time exploring bad early associations and miss pruning opportunities that are "obvious" if we

consider all aspects of the problem. For example, in the line finding problem we may find *many* initial pairings that look promising, but are not confirmed by *any* supporting points. This is especially a problem in domains where the initial points in our model may only be weakly constrained. For example in asteroid tracking, large gaps in time may produce very weak constraints on the compatibility of initial points in the track. Ideally, we would like to harness structure from all aspects of the problem to detect and avoid such dead-ends significantly earlier.

3.2 Random Sampling Approaches

An alternative to exhaustively exploring all possible associations is to use a random sampling approach. For example, the RANSAC algorithm looks for spatial structure in the data by randomly sampling a set of points, computing the model defined by these points, and testing for additional supporting points [2]. However, this type of approach is not a good fit to our problem. First, we are interested in finding *all* sets of points that fit the model(s). Second, we are interested in high density, low support domains. When the total number of points is very large and the number of points from any particular model is small, the probability of sampling a set of points from the *same* model becomes vanishingly small. For this reasons we do not consider random sampling approaches below.

3.3 Parameter Space Methods

Another approach is to search for sets by searching the parameter space of the models themselves. One popular such algorithm is the Hough transform [5]. The idea behind these approaches is that we can test whether each point is compatible with each region of parameter space, allowing us to search parameter space to find the valid sets. However, this method can be expensive in terms of both computation and memory, especially for high dimensional parameter spaces. Further, if the model's total support is low, the true model occurrences may be effectively washed out by the noise. For these reasons we do not consider parameter space methods below.

3.4 Multiple Tree Algorithms

The primary benefit of tree-based spatial search algorithms is that they are able to use spatial structure within the data to limit the cost of certain aspects of the search. However, there is a clear potential to push further and use structure from multiple aspects of the search *at the same time*. In doing so we can hopefully avoid many of the dead ends and wrong turns that may result from exploring bad initial associations in the first few points in our model. For example, in the domain of asteroid tracking we may be able to limit the number of short, initial tracks that we have to consider by using spatial structure from later time steps in our search to ignore obviously bad pairings. This idea forms the basis of multiple tree search algorithms [6, 7, 8].

The basic idea behind multiple tree methods is that we can search over *combinations* of tree nodes to incorporate structure from multiple aspects of the problem into the search. In standard single tree algorithms the current search is defined by a single

query point and the search state is represented by a single tree node. In contrast, multiple tree algorithms search over *sets* of points and represent the current search space with multiple tree nodes. For example, if we are interested in finding all pairs of points within some range of each other, we can use two tree nodes. In this respect, at each stage in the search we are effectively saying: "One of the points in the set is owned by the first tree node, another is owned by the second tree node, etc."

The use of multiple trees provides two significant potential advantages. First, this approach allows us to prune the search using structure from all aspects of the problem. We can prune the search if we ever find that the current set of tree nodes is not mutually compatible given our model. Second, we can remove redundant work performed for similar queries by asking pruning queries for entire nodes instead of individual points.

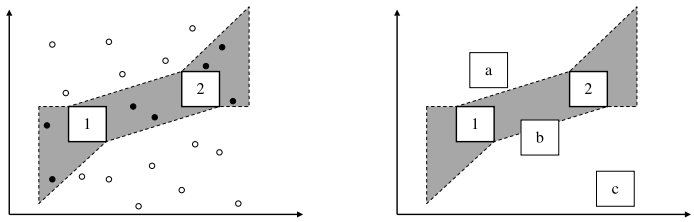
We denote this approach as the "standard multiple tree approach" or simply "multi-tree" algorithms to differentiate it from the type of new multiple tree algorithms presented below. It is important to note that although we refer to these approaches as multiple tree methods, they are not constrained to use different trees. More precisely these approaches should be called multiple tree iterator approaches, because we are iterating over sets of tree nodes (from the same or different trees).

There are several potential drawbacks to using multiple trees. First, additional trees introduce a higher branching factor in the search and increase the potential for taking "wrong turns." Second, care must be taken in order to deal with missing or a variable number of support points. Kubica *et. al.* discuss the use of an additional "missing" tree node to handle these cases [8]. However, this approach can effectively make repeated searches over subsets of trees, making it more expensive both in theory and practice.

4 Variable Tree Algorithms

In general we would like to exploit structural information from all aspects of our search problem, but do so while concentrating the actual branching of the search on just the parameters of interest. To this end we propose a new type of multiple tree algorithm that uses a *variable* number of trees during the search. Like a standard multiple tree algorithm, we search combinations of tree nodes to find valid sets of points. However, we limit this search to just those points required to define, and thus bound, the models currently under consideration. In addition, throughout the search we maintain information about other potential *supporting* points that can be used to confirm the final track.

Consider the asteroid linking problem. Each line is defined by only 2 points, thus we can efficiently search through the space of models using a dual tree search with 2 *model trees*. As shown in Figure 1, at each level of the search the bounds of our current model tree nodes immediately limit the set of feasible support points for *all* possible line segments that contain one point in each model tree node. If we track which support points are feasible, we can use this information during our search. Further, this allows us to prune infeasible support points for many possible combinations of model points at once.



(A) (B)

Figure 1: The bounding boxes of the two model tree nodes (1 and 2) define a region of feasible support points (shaded) for any potential combination of model points from those nodes. As shown in (A), the points in our current support point list can be classified as feasible (solid) or infeasible (open). As shown in (B), we can also classify entire support tree nodes as feasible (node b) or infeasible (nodes a and c).

4.1 Support List Algorithm

This leaves the questions of exactly how we search over the model points and how we handle the support points. We want a technique that both efficiently searches over the possible combinations of model points while also efficiently determining which support points are valid. One instantiation of this approach is the *support list* algorithm, which combines a multiple tree search with an explicit check for valid support points. While the support list algorithm is rarely the most efficient approach, it provides both motivation for and a good introduction to the concepts and techniques used in the full variable tree algorithms presented below.

The *support list* algorithm uses a multiple tree search to efficiently find valid combinations of model points while also maintaining a list of compatible support points. Formally, we define the search as a multiple tree search over the model points, using c' *model tree* nodes. As with the multi-tree algorithm, these tree nodes can reference the same actual tree or multiple different trees. In addition, we maintain a full list of the support points that are compatible with the current set of model tree nodes. At each level of the search we can scan the list of support points, removing the points that are no longer compatible. Figure 1.A shows an example of feasibility testing for linear tracks and two model tree nodes.

Maintaining a list of valid support points provides two important advantages during the search. First, we can use the updated list of support points to help prune the search. For example, if we need at least $k-2$ support points from unique time steps, we can prune the search whenever our list contains points from less than $k-2$ time steps. We know that no matter which pair of model points we choose, we will not have enough support points. Second, we are able to test the validity of support points for entire sets of models. By the time we reach the model tree leaf nodes, and thus start explicitly checking sets of model points, we will have pruned away a significant number of support points. Since this pruning is done with respect to entire sets of model points,

we can remove redundant or similar feasibility queries that would arise from testing a support point against several similar models.

The approach of recursively narrowing in on model parameters while maintaining a list of valid support points is similar to the fast Hough transform [9]. The fast Hough transform searches for points along a line by recursively partitioning the parameter space of all linear models and maintaining a list of all points compatible with the current set of parameters. The support list algorithm uses a more general feasibility criteria and uses the actual data points to focus the search through model space. However, it is worth noting that the techniques described below can also be directly applied to parameter space searches such as the fast Hough transform.

4.2 Variable Support Tree Algorithm

The support list algorithm has the disadvantage that we are required to test all of the current support points at each level of the search. Thus we are ignoring potential structure within the support points themselves. One simple remedy is to place the support points in a tree, bringing us back to the full multiple tree algorithms and the dangers of a high branching factor. *Is it possible to use structure from the support points without having to include them in our branching?* The answer is yes and we call the resulting technique a *variable tree* algorithm.

The variable tree algorithm provides a compromise between the full multiple tree algorithm and the support list algorithm. We place the support points in trees, but do not treat them as part of the direct search. Instead we maintain a dynamic *list* of currently valid support nodes. By considering support tree nodes instead of individual points, we can remove the expense of testing each support point at each step in the search. And by maintaining a list of support tree nodes, we can consider multiple tree nodes from the same time step as potential support, removing the need to branch over the support points. The big advantage though is that by using both a list and a tree, we can refine our support representation on the fly. If we reach a point in the search where a support tree node is no longer valid, we can simply drop it off the list. Further, if we reach a point where a support tree node provides too coarse a representation, we can simply split it and add both of its children to the list.

The variable tree algorithm, shown in Figure 2, works by performing a multiple tree search to find valid sets of model points. We use c' model tree nodes, which guide the recursion and thus the search. At each level we look for pruning opportunities based on the mutual compatibility of the model tree nodes. At the same time, the variable tree algorithm also maintains a list of compatible support tree nodes. As the multiple tree search over model points progresses, we use these nodes to determine if we have enough support and constantly refine the granularity of our support representation. Since we are not guiding the search with the support trees, the children of a given node are not mutually exclusive and we can add the right child, the left child, both children, *or* neither child to our list of support tree nodes.

A example of a simple search for lines is illustrated in Figure 3. The first column shows tree nodes that are currently part of the search. The second and third columns show the search's position on the two model trees and the current set of valid support tree nodes respectively. Unlike the pure multiple tree search, the variable tree search

	Variable Tree Model Detection
	Input: c' model tree nodes and a list of valid support tree nodes
	Output: A list of feasible sets of points
1.	Determine if we can prune based on the validity of the model tree nodes.
2.	IF we cannot prune due to the model tree nodes:
3.	FOR each support tree node S in the list
4.	IF S is compatible with the current model tree nodes:
5.	IF S is too wide or either of S's children can be pruned:
6.	Split S and add the child that is not pruned (if any) to the list.
7.	ELSE
8.	Remove S from the list of valid support tree nodes.
9.	IF we still have enough valid support nodes:
10.	IF all of the model nodes are leaves:
11.	Explicitly test all combinations of points owned by the model tree nodes, using the support nodes' points as potential support points.
12.	ELSE
13.	Let X be the non-leaf model tree node that owns the most points.
14.	Recursively search using X's left child in place of X.
15.	Recursively search using X's right child in place of X.

Figure 2: The recursive algorithm for variable tree model detection.

does not "branch off" on the support trees, allowing us to consider multiple support tree nodes from the same time step at any point in the search.

This leaves the question of when to split support tree nodes. If we split them too soon, we may end up with many support nodes in our list and mitigate the benefits of the nodes' spatial coherence. If we wait too long to split them, then we may have a few large support nodes that cannot efficiently be pruned. Although we are still investigating splitting strategies, the experiments in this paper use a heuristic that seeks to provide a small number of support nodes that are a reasonable fit to the feasible region. We split a support node if it is wider than some constant factor of the smallest model node's width or if doing so would allow one of its two children to be pruned.

5 Results on the Asteroid Linking Domain

The goal of the single-night asteroid linkage problem is to find sets of 2-dimensional point detections that correspond to a roughly linear motion model. In the below experiments we are interested in finding sets of at least 7 detections from a sequence of 8 images. The movements were constrained to have a speed between 0.05 and 0.5 degrees per day and were allowed an observational error threshold of 0.0003 degrees.

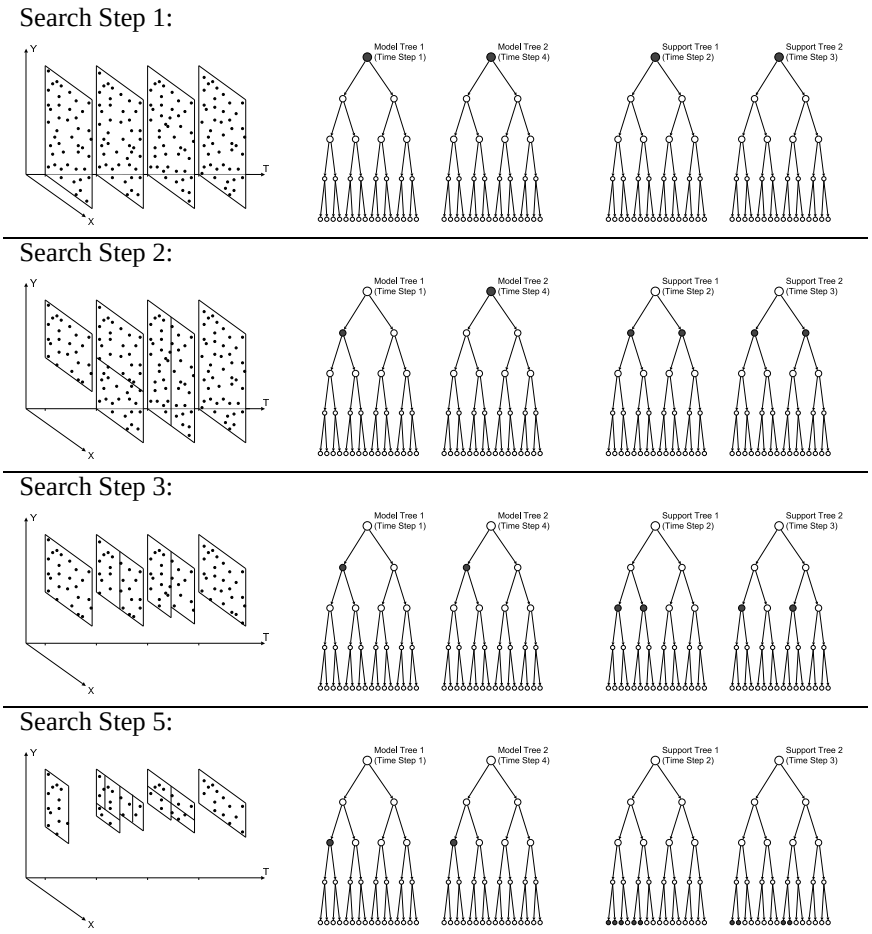


Figure 3: The variable tree algorithm looks for valid tracks by performing a depth first search over the model trees' nodes. At each level of the search the model tree nodes are checked for compatibility with each other and the search is pruned if they are not compatible. In addition, the algorithm maintains a list of compatible support tree nodes. Since we are not guiding the search with the support trees we can split the support trees and add: the right child, the left child, both children, *or* neither child to our list of support tree nodes. This figure shows a simple rule where the support tree nodes are split exactly once at each level of the search. Support tree nodes are only added if they are compatible with the entire set of model tree nodes.

5.1 Detection Data

The asteroid detection data consists of detections from 8 images of the night sky separated by half-hour intervals. The images were obtained with the MegaCam instrument

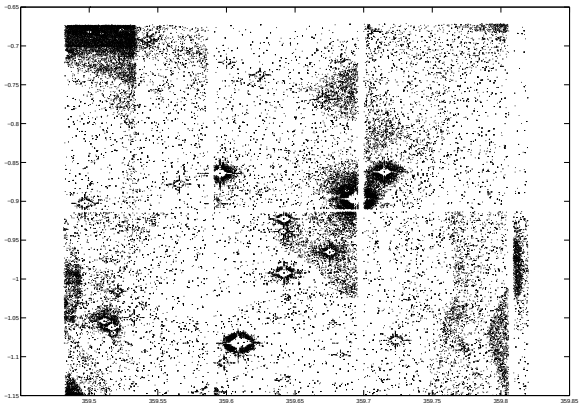


Figure 4: 6 of the 36 cells that make up a full image. The figure shows detections from all eight time steps.

Table 1: The running times (in seconds) for the asteroid linkers with different detection thresholds (and thus different numbers/density of observations).

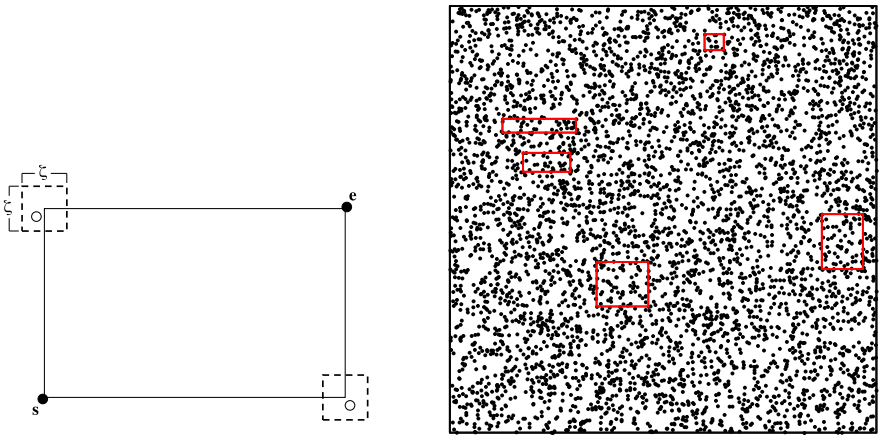
σ	10.0	8.0	6.0	5.0	4.0
N	3531	5818	12911	24068	48646
Single Tree	2	7	61	488	2442
Multi-Tree	1	3	30	607	4306
Support List	4	10	64	498	2399
Variable-Tree	< 1	1	4	40	205

on the 3.6-meter Canada-France-Hawaii Telescope. The detections, along with confidence levels, were automatically extracted from the images. We can pre-filter the data to pull out only those observations above a given confidence threshold σ. This allows us to examine how the algorithms perform as we begin to look for increasingly faint asteroids. Figure 4 shows example data from eight MegaCam images.

It should be noted that only limited preprocessing was done to the data, resulting in a very high level of false detections. While future data sets will contain significantly reduced noise, it is interesting to examine the performance of the algorithms on this real-world high noise, high density data.

5.2 Results

The results on the asteroid tracking domain, shown in Table 1, illustrate a clear advantage to using a variable tree approach. As detection threshold σ decreases, the number and density of detections increases, allowing the support tree nodes to capture feasibility information for a large number of support points. In contrast, neither the



(A) (B)

Figure 5: The goal of the rectangle problem is to find sets of points that form the corners of a rectangle (A). An example random data set with 5000 points and 5 known rectangles is shown (B).

full multi-tree algorithm nor the single-tree algorithm performed well. For the multi-tree algorithm, this decrease in performance is likely due to a combination of the high number of time steps, the allowance of a missing observation, and the high density. In particular, the increased density can reduce opportunities for pruning, causing the algorithm to explore deeper before backtracking.

6 Experiments on the Simulated Rectangle Domain

In addition to the problem of finding linear tracks, we can apply the above techniques to other model-based spatial search problems. In this section we consider one such problem, finding rectangles in a dense set of noisy points. This problem is illustrated in Figure 5.A and an example data set with 5 rectangles is shown in Figure 5.B. We use this simple, albeit artificial, problem to analyze the behavior of the algorithms as we vary the properties of the data. This problem also serves as a simple example to demonstrate potential object detection and pattern recognition applications of this approach.

6.1 Problem Definition

Formally, the rectangle finding problem consists of searching for hyper-rectangles in D-dimensional space by finding k or more *corners* that fit a rectangle. We restrict the model to use the upper and lower corners as the two model points. Potential support points are those points that fall within some threshold of the other $2^d - 2$ corners. In

Table 2: Average running times (in seconds) for a 2-dimensional rectangle search with different numbers of points. The brute force algorithm was only run up through $N = 2500$.

N	500	1000	2000	2500	5000	10000	25000	50000
Brute Force	0.37	2.73	21.12	41.03	n/a	n/a	n/a	n/a
Single Tree	0.02	0.07	0.30	0.51	2.15	10.05	66.24	293.10
Multi-Tree	0.01	0.02	0.06	0.09	0.30	1.11	6.61	27.79
Support List	0.01	0.03	0.09	0.14	0.43	1.47	7.92	29.65
Variable-Tree	0.01	0.02	0.05	0.07	0.22	0.80	4.27	16.30

addition, we restrict the allowable bounds of the rectangles by providing a minimum and maximum width.

6.2 Experiments and Results

The first factor that we examined was how each algorithm scales with the number of points. We generated random data with 5 known rectangles and N additional random points and computed the average wall-clock running time (over ten trials) for each algorithm. The results, shown in Table 2, show a graceful scaling of all of the multiple tree algorithms. In contrast, the brute force and single tree algorithms run into trouble as the number of points becomes moderately large. The variable tree algorithm consistently performs the best, as it is able to avoid significant amounts of redundant computation.

One potential drawback to the multiple tree algorithms is that because they use support point information during the search, they may become inefficient as the allowable number of missing support points grows. To test this we looked at a 3-dimensional rectangle problem and varied the minimum number of required support points. The results are shown in Table 3. As shown, the multiple tree methods become *more* expensive as the number of required support points decreases. This is especially the case for the multi-tree algorithm, which has to perform several almost identical searches to account for missing points. It should be noted however that the variable-tree algorithm's performance degrades gracefully and is the best for all trials.

7 Conclusions

Tree-based spatial algorithms provide the potential for significant computational savings with multiple tree algorithms providing further opportunities to exploit structure in the data. However, a distinct trade-off exists between ignoring structure from all aspects of the problem and increasing the combinatorics of the search. We presented a variable tree approach that exploits the advantages of both single tree and multiple tree algorithms. A combinatorial search is carried out over just the minimum number of model points, while still tracking the feasibility of the various support points.

Table 3: Average running times (in seconds) for a rectangle search with different numbers of allowable missing points. For this experiment $N = 10000$ and $D = 3$

Number of Missing Corners Allowed	0	1	2	3	4
Single Tree	4.71	4.72	4.71	4.71	4.71
Multi-Tree	3.96	19.45	45.02	67.50	78.81
Support List	1.79	1.99	2.29	2.46	2.77
Variable-Tree	0.65	0.75	0.85	0.92	1.02

As shown in the above experiments, this approach provides significant computational savings over both the traditional single tree and and multiple tree searches.

Acknowledgments

Jeremy Kubica is supported by a grant from the Fannie and John Hertz Foundation. Andrew Moore and Andrew Connolly are supported by a National Science Foundation ITR grant (CCF-0121671).

References

[1] A.J. Storkey, N.C. Hambly, C.K.I. Williams, and R.G. Mann. Renewal Strings for Cleaning Astronomical Databases. In *Uncertainty in Artificial Intelligence 19*, 559-566, 2003.

[2] M.A. Fischler and R.C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Comm. of the ACM*, 24:381–395, 1981.

[3] S. Blackman and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House, 1999.

[4] J. K. Uhlmann. Algorithms for multiple-target tracking. *American Scientist*, 80(2):128–141, 1992.

[5] P. V. C. Hough. Machine analysis of bubble chamber pictures. In *International Conference on High Energy Accelerators and Instrumentation*. CERN, 1959.

[6] A. Gray and A. Moore. N-body problems in statistical learning. In T. K. Leen and T. G. Dietterich, editors, *Advances in Neural Information Processing Systems*. MIT Press, 2001.

[7] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. of the 1998 ACM-SIGMOD Conference*, 237–248, 1998.

[8] J. Kubica, A. Moore, A. Connolly, and R. Jedicke. A Multiple Tree Algorithm for the Efficient Association of Asteroid Observations. In *The Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 138–146, 2005.

[9] H. Li, M.A. Lavin, and R.J. Le Master. Fast Hough Transform: A Hierarchical Approach. In *Computer Vision, Graphics, and Image Processing*, 36(2-3):139–161, November 1986.