An Empirical Analysis of Techniques for Constructing and Searching K-Dimensional Trees

Douglas A. Talbert
Vanderbilt University
Dept of Electrical Engineering & Computer Science
Box 1679-B, Nashville, TN 37235
+1-615-936-3805
dat@vuse.vanderbilt.edu

Doug Fisher
Vanderbilt University
Dept of Electrical Engineering & Computer Science
Box 1679-B, Nashville, TN 37235
+1-615-343-4111
dfisher@vuse.vanderbilt.edu

ABSTRACT
Affordable, fast computers with large memories have lessened the demand for program efficiency, but applications such as browsing and searching very large databases often have rate-limiting constraints and therefore benefit greatly from improvements in efficiency. This paper empirically evaluates several variants of a common k-dimensional tree technique to demonstrate how different algorithm options influence search cost for nearest neighbors.

Categories and Subject Descriptors
H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms
K-dimensional trees, nearest-neighbor search techniques.

1. INTRODUCTION
The memory and speed of today's computers seem to obviate the need for the peak efficiency that minimal memory and relatively slow speeds used to demand. However, there remain computing scenarios with significant rate-limiting components in which slight efficiency improvements provide significant performance gains.

Two types of searches with such rate-limiting components are *browsing* and *searching very large databases*. During browsing, each step of the search requires a human to analyze the state of the search and decide which step to take next, and when searching a very large database (one which cannot fit into main memory), each step in the search may require a disk access. In both examples, each step of the search is costly and should be avoided if possible.

A common organizational structure used for nearest-neighbor searching is a *k-dimensional tree* (kd-trees) [3]. Kd-trees are designed for efficient nearest-neighbor search. This paper presents an empirical analysis of kd-tree nearest-neighbor searches with respect to several independent variables. Our goal is to demonstrate the effect on search cost of various kd-tree construction and search options and to explain the exhibited relationships to enable researchers or automated agents to adapt a kd-tree algorithm to better address their needs.

The next section provides background information on kd-trees. Section 3 presents the algorithmic variants we tested, and section 4 explains the experimental results. The last section describes implications of the results and points to future work.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
KDD 2000, Boston, MA USA
© ACM 2000 1-58113-233-6/00/08 ...$5.00

2. BACKGROUND

2.1 Kd-Trees
Kd-trees are multi-dimensional binary search trees that organize vectors of numeric data to facilitate nearest-neighbor searches. Each internal node of the tree represents a branching decision in terms of a single attribute's value, called a *split value*. All data with values less than or equal to the split value along that dimension follow one branch and all data with values greater than the split value along that dimension follow the other branch. Each leaf is a collection of data items. The number of items in each leaf is bounded by a user-specified threshold.

Kd-trees have been used to assist in such automated learning tasks as clustering [7] and instance-based learning [4].

2.2 Search Algorithm
Given a kd-tree constructed over a set of numeric vectors, *D*, the search for the nearest neighbor of a numeric vector, **v**, in *D* proceeds as follows:

1. Starting at the root, use the values in **v** to follow the branches of the kd-tree to an initial leaf.
2. Compute the distance to each data item at the leaf; store the closest item and its distance.
3. Continue searching the tree in a depth-first fashion, using the values in **v** to determine which branch to search first at each internal node.
4. Whenever a leaf is encountered, compute the distances to each data item at the leaf and update the nearest-neighbor information when needed.

This algorithm does not address two issues raised by missing values.

1. how to decide which branch to follow when **v** has no value for the node's split attribute
2. how to compute a distance when either **v** or a data item from *D* is missing one or more values

When **v** is missing a value for a split attribute, there is no obvious advantage to picking one branch over the other. Thus, we pick the "less than or equal to" branch first and explore the "greater than" branch later in the depth-first search.

Attributes for which **v** has no value are considered irrelevant and can be ignored in distance calculations. If, however, **v** has a value for an attribute and a data item does not, we need to somehow include that attribute in the calculations.

We address this with a technique employed by Aha [1] for similarity computations during instance-based learning. We assume that the missing value is as distant from the known value in **v** as possible given the known range for that attribute. Assuming that the range exhibited for each attribute in the data set is accurate, this pessimistic approach guarantees that the data item identified as the nearest neighbor is correct even if that item has missing values for relevant attributes.

The algorithm above assumes interest in the nearest *single* neighbor. In general, one can search for as many "nearest" neighbors as desired.

2.3 Search Cost

Since we are interested in measuring search efficiency we must quantify search cost. Metrics used in previous kd-tree evaluations include *search time* [5][8], *number of nodes examined during search* [5][6][8], *number of distance calculations performed* [5][6], and *the number of dimensional comparisons performed during search* [9][10]. (A dimensional comparison occurs once at each internal node to decide which branch to search first and many times at each leaf during distance calculation.)

Our analysis uses *number of nodes* and *number of dimensional comparisons* to evaluate kd-trees along dimensions not explicitly compared in previous work. The number of dimensional comparisons is a true measure of cost under ideal circumstances, but the number of nodes may better reflect disk accesses when data must be stored on secondary memory.

2.4 Reducing Search Cost

The structure of kd-trees can be exploited to improve the search algorithm above. Because each split in the kd-tree is based on a single value of a single attribute, each node in the tree covers a hyperrectangular region of the search space. The root covers the entire search space, and the space covered by any other node in the tree is bounded by the hyperplanes defined by the split points in all its ancestor nodes.

To exploit this property, we must recognize that the search space in which a possible nearest neighbor might be found is always a *hypersphere*. The radius of this hypersphere is the distance to the current nearest neighbor. (Prior to finding a candidate nearest neighbor, the radius of the sphere is 8 .) If multiple nearest neighbors are sought, the radius of the hypersphere is the distance to the farthest item in the *set* of candidate nearest neighbors. The relationship between the hypersphere of relevant search space and the hyperrectangle covered by a node can be used to prevent unnecessary searching.

2.4.1 "Strong" Pruning

A more complete description of these aspects of kd-trees and kd-tree search can be found in Friedman, et al. [5] where they present a search pruning (not tree pruning) technique based on this knowledge. This pruning technique checks to see whether a node needs to be examined. If a node's hyperrectangle does not intersect the hypersphere of relevant search space, then neither that node nor the subtree beneath it can contain any data items closer than the current candidate nearest neighbors(s).

This is "strong" pruning because it only allows searching a node if the hyperrectangle and the hypersphere do indeed overlap. It is performed prior to searching the second child of any node.

(It is important to note that this and the other pruning techniques that follow involve distance calculations and therefore contribute to the number of dimensions examined during a search.)

2.4.2 "Weak" Pruning

Sproull [8] describes a weaker, heuristic pruning technique. He observed that the strong pruning check is relatively expensive and that it may be approximated by checking if the hypersphere of relevant search space intersects the hyperplane formed by the node's split point.

When the strong pruning check is true (i.e., if the node does need to be searched), then the weak check is also true, but the converse is not true. The weak check *is* less costly to perform but results in more nodes being searched. This is because it does not examine all the bounds on a hyperrectangle; it only examines one hyperplane boundary. Bounds from other ancestor nodes might indicate that there is no need to search that space, but this check can not know that.

2.4.3 Stopping Search Early

Friedman, et al. [5] also present search pruning technique that determines whether or not the search is completed. This is true if the hyperrectangle of a node that has been completely searched contains the entire hypersphere of relevant search space. Then, the search can stop because we are guaranteed to have the correct nearest neighbor(s). It is checked prior to backtracking out of each node.

2.4.4 Reduced Distance Calculations

Another cost reduction employed by Sproull [8] is to stop distance calculations as soon as possible. For example, if the data vectors are defined using 50 attributes, then normally each distance calculation would require 50 dimensional computations. If, however, after the first two dimensions, the distance is already greater than the distance to the current candidate nearest neighbor, no more useful calculations need to be performed. Thus, 48 dimensional computations would be eliminated.

2.5 Construction Algorithm

The basic kd-tree construction algorithm can be described as follows:

1. Select a split point.
2. Partition the data accordingly.
3. Repeat until the size of each node is less than or equal to a user-specified threshold. These nodes are the leaves of the kd-tree.

Various split-point algorithms have been proposed. Bentley's [3] original spilt-point algorithm simply cycled through the attributes. The actual value was randomly selected from the set of values in the subset of items at the node.

Friedman, et al. [5] showed that splitting on the median of the attribute with the broadest range at each node was an improvement. Moore [6] then argued that the arithmetic mean provided a better split point because it results in hyperrectangle dimensions with more uniform sizes.

All of the construction techniques above assume no knowledge of the query distribution. Talbert and Fisher [9] showed that by starting with a tree built using Friedman, et al.'s technique, a more efficient kd-tree could be learned by adapting to the observed queries. Also, Talbert and Fisher [10] showed that the distribution of the data set itself can be exploited to produce trees that are improvements over Friedman, et al.'s.

In addition to a split-point algorithm, the maximum size of a leaf must be chosen prior to tree construction. Furthermore, techniques need to be in place for handling missing values during tree construction and for normalizing the data prior to tree construction. The technique we use to handle partitioning when an element has no value along the split-point attribute is to always place that item down the "less than or equal to" branch. The search algorithm guarantees that if that item is the nearest neighbor, it will be found.

If the data's attributes use different scales of measurement, it may be necessary to normalize the data because those differences could skew the distance calculations. We normalized our data set by dividing each value along an attribute by that attribute's standard deviation [2].

3. ALGORITHMS

Kd-tree construction has at least two elements that can vary.

1. the split point selection algorithm
2. the maximum number of data items in a leaf

Kd-tree search has at least three variables.

1. the pruning technique that will be used (if any)
2. whether or not to check for early completion of the search
3. whether or not to perform reduced distance calculations

Presumably, the choices made on these options influence the cost of the search. The algorithms in our tests use many different combinations of choices along these axes.

For tree construction, the leaf sizes we tested are 1, 2, 5, 10, 20, and 50, and the split-point algorithms we tried were *median*, *arithmetic mean*, *harmonic mean*, and *inner quartile mean* of the attribute with the broadest range at each node. We consider more split point algorithms than previous literature because we know that the arithmetic mean is sensitive to outliers (whereas the harmonic and inner quartile mean as less so), and we are interested in the affect outliers have on kd-trees.

For searching, the pruning techniques we evaluate include *no pruning*, *strong pruning*, and *weak pruning*. We also consider a "hybrid" pruning method that relies on weak pruning as long as it indicates not to search a node but use strong pruning the verify that the other nodes do indeed need to be searched. Furthermore, we evaluate the utility of checking for early search completion and of performing reduced distance calculations.

Collectively, these experiments are intended to provide baselines against which data and query sensitive heuristics can be compared.

4. EXPERIMENTS

4.1 Methodology

For each experiment, we performed a 10-fold cross-validation. We randomly partitioned the data into ten groups and averaged the results of building a kd-tree using each set of nine of the groups and using the tenth group as the set of search vectors.

We performed 162 tests in which we varied our choices along all of the axes above, leaf size, split point algorithm, pruning technique, trying to stop the search early, and performing reduced distance calculations. The precise variants tests are described below.

4.2 Nutrition Data Set

The nutrition database was derived from the United States Department of Agriculture's Nutrient Database for Standard Reference [11] and consists of nutritional descriptions of 5,976 foods. These descriptions are vectors of values describing the content of various nutrients, such as protein, fat, and vitamins. The USDA database describes each food in terms of nutrients per 100 grams. For our database, we precomputed the appropriate vectors for the common serving sizes listed in the database. This process resulted in a database with 11,697 data items.

4.3 Results

4.3.1 Split Point Selection Algorithm

Figure 1a shows the average cost in nodes of searching for a single nearest neighbor across all four split point algorithms using the weak pruning technique in trees with a number of different maximum leaf sizes, and Figure 1b shows the data for searches using the strong pruning method. (We did not include the graph for the no-pruning option, but its results are similar with respect to split point algorithms.)

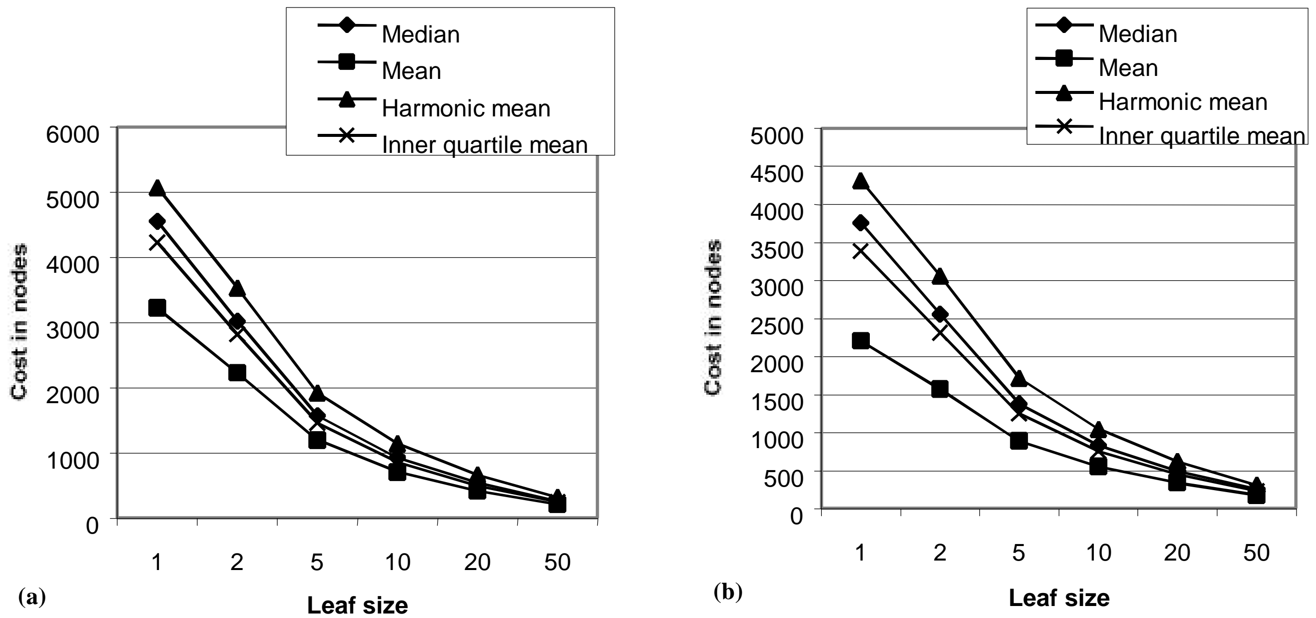


Figure 1. The arithmetic mean results in the lowest number of nodes examined during search for both weak (a) and strong (b) pruning.

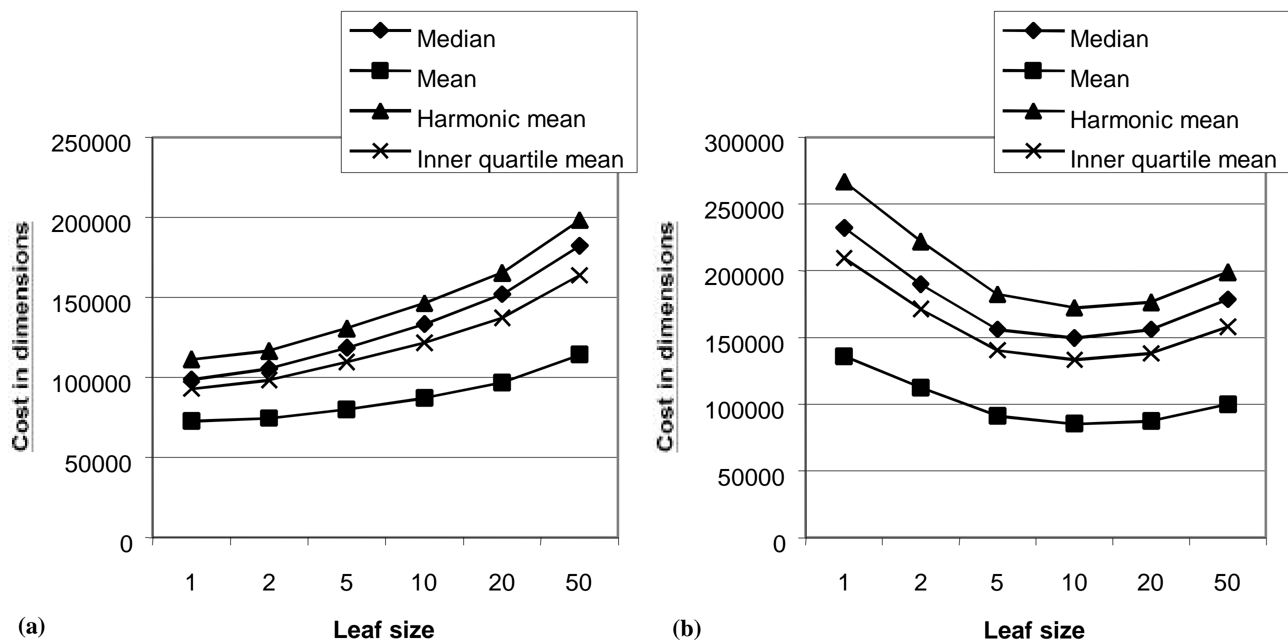
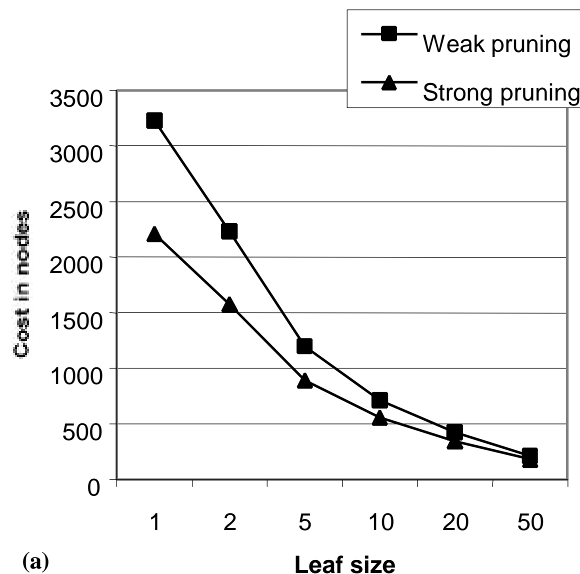


Figure 2. Curves representing cost in number of single dimension comparisons for both weak pruning (a) and strong pruning (b) show that using the arithmetic mean enables more efficient searches than the other split point algorithms.

The differences between the costs at each leaf size are significant at the 0.01 level by the sign test [12]. Of the split point algorithms examined, *the mean enables the most efficient search* (for this data set at least) across a variety of pruning methods and maximum leaf sizes.

Figure 1 shows that as leaf size increases, cost in nodes decreases and the gap between the split point algorithm closes. When we examine cost in dimensions (a more accurate reflection of the true computational cost), we see a different story. Figure 2 shows the costs in dimensions for all split point algorithms for searches using both weak and strong pruning techniques in trees with a variety of maximum leaf sizes.

The ranking of the split point algorithms remains the same and all differences at each leaf size are still significant, but the cost



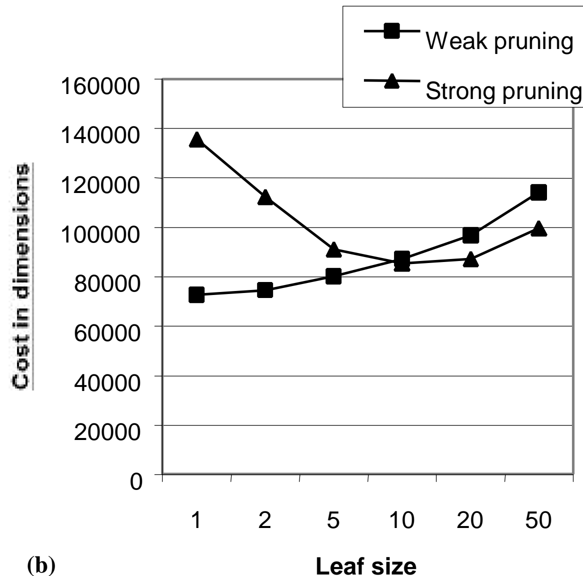


Figure 3. The cost in nodes (a) has an inverse relation to leaf size for all pruning techniques, but the relationship between leaf size and cost in dimensions (b) is dependent on the pruning technique.

gap does not close. In fact, when using weak pruning, the gap increases. The weak technique is more sensitive to inefficiencies in the tree because it prunes less than strong pruning. The rest of the experiments performed only for the mean split-point algorithm.

4.3.2 Leaf Size

We have already mentioned the effect leaf size has on cost in reference to Figures 1 and 2, but we will now take a closer look at the role of leaf size in cost. Curves shown in Figure 3a are taken from Figures 1a and 1b. Curves shown in Figure 3b are taken from Figures 2a and 2b. Figure 3 shows the cost in nodes and dimensions of search trees built using the arithmetic mean for the split point and having a variety of maximum leaf sizes. All differences at each leaf size for both graphs is significant (in the sense that strong pruning leads to a fewer number of nodes searched in a significant number of folds by the sign test)..

As would be expected, average cost in nodes decreases with leaf size, and Figure 3a shows the Sproull's pruning technique results in searching more nodes than Friedman, et al.'s. We have not shown the no-pruning option, but the cost in nodes for no-pruning gracefully decreases from 20,000+ nodes at a leaf size of 1 to virtually the same number of nodes as the pruning methods at a leaf size of 50.

Figure 3b shows the average cost in dimensions weak and strong pruning. The strong approach minimizes cost at an intermediate leaf size while weak pruning monotonically increases with leaf size. We have not shown the no-pruning option here either, but its cost starts at approximately 500,000 and decreases only slightly as leaf size increases.

To explain the relationship between leaf size and cost in dimensions for searches using strong pruning, we must understand that searching with this method has two different primary costs. It has the expensive distance calculations at the leaves that all kd-tree searches have, but it also has an expensive pruning check. When a tree has many very small leaves, the pruning check has to be performed frequently driving cost up, and when the leaves are significantly larger, the pruning cannot provide a fine-grained search focusing on only the most relevant parts of the search space. Between these extremes, there is optimal leaf size at which the pruning checks are not excessive and the leaves cover adequately small portions of the search space.

Weak pruning's monotonic increase is because, unlike the strong technique, its pruning check is inexpensive. Thus its relative cost across leaf sizes is influenced primarily by how precise the

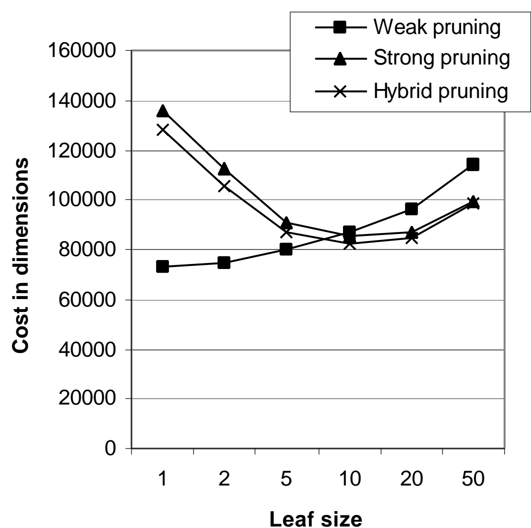
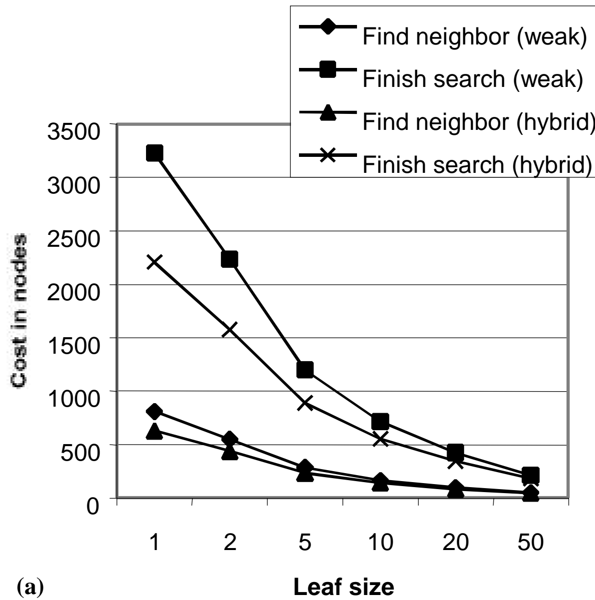


Figure 4. Hybrid pruning results in a lower cost in terms of dimensions than strong pruning alone.



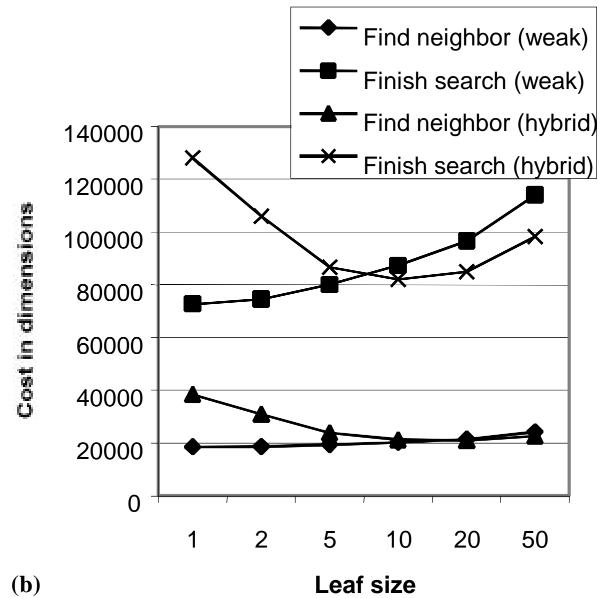


Figure 5. The cost of finding the single nearest neighbor and of completing the search for trees build using both the weak and the hybrid techniques in terms of nodes (a) and dimensions (b).

pruning can be. Larger leaves result in less pruning precision and higher costs.

4.3.3 Strong vs. Weak Pruning

Figure 3b shows the costs in number of dimensional comparisons for weak and strong pruning. We also tested a hybrid pruning approach in which strong pruning was only used when weak pruning indicated that a node needed to be searched. This guarantees the same number of nodes will be searched as when strong pruning is used by itself but should result in a lower cost in dimensions because some nodes will be pruned using the weak method alone.

We did not include the graph, but the number of nodes searched using the hybrid approach was indeed the same as for strong pruning. In contrast, Figure 4 (which superimposes the results of the hybrid approach on the curves of Figure 3b) shows that hybrid pruning is a modest improvement over strong pruning in terms of the number of dimensions. The differences between the hybrid approach and the strong method are statistically significant.

(For the rest of our experiments, we use the hybrid approach, rather than strong pruning, in comparisons to weak pruning.)

Another measure of how well a pruning technique works can be seen in the amount of "wasted" search. In kd-trees that would be a measure of how much search had to occur *after* finding the nearest neighbor. Figure 5 show this statistic for cost in nodes and in dimensions for weak and hybrid pruning.

As in Figure 1, looking only at cost in nodes is somewhat deceptive, whereas Figure 5b shows that there is considerable cost in terms of dimensional checks expended after finding (what an oracle would know to be) that nearest neighbor.

4.3.4 Stopping Search Early

In our experiments, trying to stop the search early by checking if the hypersphere of relevant search space in completely enclosed within a searched node's hyperrectangle never resulted in any savings in terms of nodes. It did, however, drive up the cost in dimensions.

One likely reason for this result is the number of missing values in the data sets. This check to stop always fails if there is an

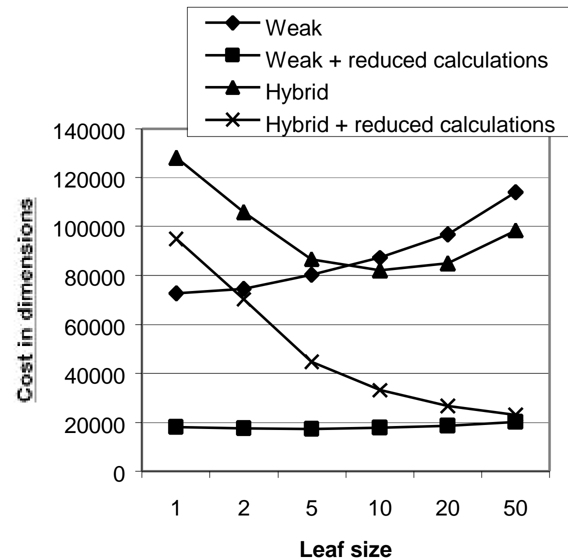


Figure 6. The cost in dimensions of searching with and without reduced distance calculations for both weak and hybrid pruning.

ancestor with a split-point attribute for which the query has no value.

4.3.5 Reduced Distance Calculations

Figure 6 shows the effect of reduced distance calculations on weak and hybrid pruning. Lower costs were expected, and the figure shows this was true. The change in shapes of the curves resulting from the reduced calculations indicates that, in such cases, the lower pruning precision caused by large leaves is less costly. We did not include cost in nodes because that is not affected by reduced distance calculations.

Figure 7 shows the change in "wasted" search (in terms of cost in dimensions) when reduced calculations are used. In comparing this to Figure 5b we see that gap between the "find" cost and the "finish" cost in greatly reduced especially at large leaf nodes.

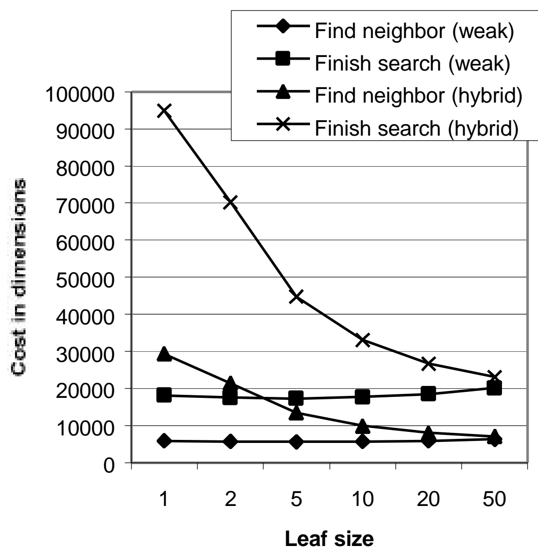


Figure 7. The cost of finding the single nearest neighbor and of completing the search for trees build using both the weak and the hybrid techniques in terms of nodes (a) and dimensions (b) when reduced distance calculations are used.

5. CONCLUSION

The implications of our study are severalfold. In terms of dimensions examined, the weak pruning method seems advantageous, for small leaf sizes. In a situations where we expect human users to be browsing a database (in addition to automated and semiautomated searches), small leaf sizes are desirable so as to limit the amount of undirected browsing of large lists of tuples, which would be required at large leaves.

In contrast, strong (or hybrid) pruning appears to be preferred in terms of the number of nodes visited, particularly at large leaf sizes. In a large database setting where most of the data must reside on disk (or distributed across machines), different nodes of a kd-tree (or other indexing structure) will likely implicate different disk sectors or machines. We would like to minimize the nodes visited in such a case, because secondary memory or remote-machine accesses will dominate cost. While our node counts combine internal nodes and leaves, results in terms of number of leaves only followed the same trends. This is an important observation, since a real database application might pack many indices (internal nodes) onto one disk sector, but place only one leaf on a disk sector.

Thus, our analysis can inform the development of kd-tree split heuristics and tree (not search) pruning criteria (in the sense of decision tree induction where tree expansion ceases) that are sensitive to the distribution of data and queries. By making split decisions that respect locality desirata for secondary-memory storage or distributed database environments, more informed split and pruning decisions minimize expected search cost.

Our tests show that using reduced distance calculations is better no matter what pruning method is being used, and our experiments indicate that a large amount of search is spent after the nearest neighbor(s) has been found. Thus, future work might include the development of a heuristic to allow search to stop with an "adequately" near neighborhood even if it is not guaranteed to be the absolute nearest one.

Future work should also include the application of these conclusions into further work on query sensitive kd-tree construction and search approaches. Other possible directions include examining the relationship between kd-tree variants the overall performance of larger tasks that have a kd-tree as a component.

6. REFERENCES

[1] Aha, D. W. "Tolerating Noisy, Irrelevant and Novel Attributes in Instance-Based Learning Algorithms." *International Journal of Man-Machine Studies*, *36*, 267–287 (1992).

[2] Bendat, J. S. & Piersol, A. G. *Random Data: Analysis and Measurement Procedures*. New York: John Wiley & Sons (1986).

[3] Bentley, J. L. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM*, *18*, 509–517 (1975).

[4] Deng, K. & Moore, A. W. "Multiresolution Instance-Based Learning," In G. Weiss & S. Sen (Eds.), *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, (pp. 1233–1239), Berlin: Springer-Verlag (1995).

[5] Friedman, J. H., Bentley, J. L., & Finkel, R. A. "An Algorithm for Finding Best Matches in Logarithmic Expected Time." *ACM Transactions on Mathematical Software*, *3*, 209–226 (1977).

[6] Moore, A. W. "An Introductory Tutorial on Kd-Trees." Extract from *Efficient Memory-based Learning for Robot Control* (Technical Report 209). Computer Laboratory, University of Cambridge (1991).

[7] Moore, A. W. "Very Fast EM-Based Mixture Model Clustering Using Multiresolution Kd-Trees." In M. S. Kearns, S. A. Solla, & D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, 2, Cambridge: MIT Press (1999).

[8] Sproull, R. F. "Refinements to Nearest-Neighbor Searching in k-Dimensional Trees." *Algorithmica*, *6*, 579–589 (1991).

[9] Talbert, D. A. & Fisher, D. "OPT-KD: An Algorithm for Optimizing Kd-Trees." In I. Bratko & S. Dzeroski (Eds.), *Machine Learning: Proceedings of the Sixteenth International Conference*, (pp. 398–405). San Francisco: Morgan Kaufmann (1999).

[10] Talbert, D. A. & Fisher, D. "Exploiting Sample-Data Distribution to Reduce the Cost of Nearest-Neighbor Searches with Kd-Trees." In D. J. Hand, J. N. Kok, & M. R. Berthold (Eds.), *Advances in Intelligent Data Analysis: Proceedings of the Third International Symposium, IDA-99*, *Lecture Notes in Computer Science*, *1642*, pp.407–414, Berlin: Springer-Verlag (1999).

[11] USDA "Nutrient Database for Standard Reference (Release 12)" [Electronic database]. Washington, DC: United States Department of Agriculture, Agricultural Research Service, Nutrient Data Laboratory [Producer and Distributor]. Available WWW: www.nal.usda.gov Directory: fnic/foodcomp/Data/SR12/dnload/ File: sr12asci.zip (1999).

[12] Woolson, R. F. "Statistical Methods for the Analysis of Biomedical Data." New York: John Wiley & Sons (1987).