A Dynamic Adaptation of AD-trees for Efficient Machine Learning on Large Data Sets

Paul Komarek KOMAREK@ANDREW.CMU.EDU
Department of Mathematical Sciences, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213 USA

Andrew Moore AWM@CS.CMU.EDU
School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213 USA

Abstract

This paper has no novel learning or statistics: it is concerned with making a wide class of pre-existing statistics and learning algorithms computationally tractable when faced with data sets with massive numbers of records or attributes. It briefly reviews the static AD-tree structure of Moore and Lee (1998), and offers a new structure with more attractive properties: (1) the new structure scales better with the number of attributes in the data set; (2) it has zero initial build time; (3) it adaptively caches only statistics relevant to the current task; and (4) it can be used incrementally in cases where new data is frequently being appended to the data set. We provide a careful explanation of the data structure, and then empirically evaluate the performance under varying access patterns induced by different learning algorithms such as association rules, decision trees and Bayes net structures. We conclude by discussing the longer term benefits of the new structure: the eventual ability to apply AD-trees to data sets with real-valued attributes.

Table 1. Sample data set with three attributes and six records.

ATTRIBUTES:	a_1	a_2	a_3
RECORD$_1$	1	1	1
RECORD$_2$	2	3	1
RECORD$_3$	2	4	2
RECORD$_4$	1	1	1
RECORD$_5$	2	3	1
RECORD$_6$	2	3	1

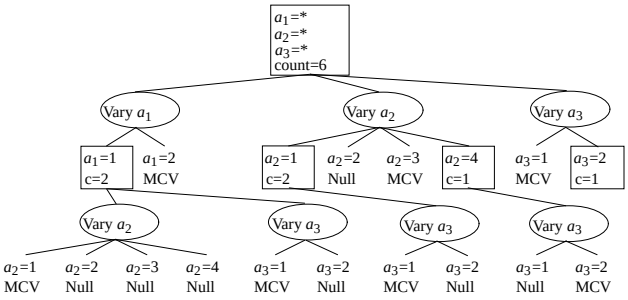

Figure 1. Sample static AD-tree for data set in Table 1, without leaf lists.

1. Description of AD-trees

1.1 What is an AD-tree?

Table 1 shows a tiny data set with $M = 3$ symbolic (i.e., categorical) attributes (the columns), and $R = 6$ records (the rows). A counting query has the form C($a_1 = 2 \wedge a_2 = * \wedge a_3 = 1$), and is a request to count the number of records matching the query, with asterisks interpreted as "don't cares". C($a_1 = 2 \wedge a_2 = * \wedge a_3 = 1$)=3 in our example.

Moore and Lee (1998) and Anderson and Moore (1998) introduced a new data structure for representing the cached counting statistics for a categorical data set, called an All-Dimensions tree (AD-tree). The AD-tree corresponding to the dataset in Table 1 is shown in Figure 1.

Each rectangular node in the figure stores the value of one conjunctive counting query. These rectangles are called AD-nodes. Let N represent an AD-node, and let Q be the query whose count C(Q) is stored in N. Let k be the greatest index such that the value of attribute a_k is specified in Q. Node N has $M-k$ children called Vary nodes, one for each attribute with index greater than k. These children are displayed as ovals with labels "Vary a_{k+1}, Vary a_{k+2}, ..., Vary a_M", and serve to specialize the query Q. Vary a_i chooses attribute a_i for specialization, and has one AD-node child for each value of a_i. The j^{th} AD-node child of Vary a_i stores the value of the counting query C($Q \wedge a_i = j$).

A tree built in this way would be enormous for non-trivially sized data sets. To save space, all AD-nodes storing zero counts are omitted from the AD-tree. In practice however, this does not afford enough of a space-savings to make such a tree practical. Consider the BIRTH data set of Moore and Lee (1998). This data set has 9,672 rows of 97 attributes, most of which are binary, Seventy of the binary attributes are very sparse, with 95% of the values being FALSE. After omitting all zero-count AD-nodes, it would require 10^{38} nodes. The trivial data set of Table 1 would require twelve nodes. Another small reduction in size, and increase in speed, can be obtained by not expanding AD-nodes near the bottom of the AD-tree. Instead, when the number of rows relevant to an AD-node is below the global "leaf list size", the indices of the relevant rows are stored in a "leaf list". Queries which require a more specialized count than this AD-node can provide, search only the data set rows mentioned in the leaf list.

More important is an innovation in Moore and Lee (1998) that allows us to never expand any child of a Vary node in which that child has a higher count than its immediate siblings. Such a child is called a Most Common Value child, or simply an `MCV`. Neither it nor any of its descendants are stored in the tree, nor are they even temporarily computed during its construction. The same paper shows how the answer to any query, not merely those remaining in the AD-tree, can be reconstructed efficiently despite this pruning. More importantly, this pruning has a dramatic effect on the memory used by the AD-tree. The BIRTH data set requires only 10^5 nodes, for example.

AD-trees thus allow constant-time counting, independent of the number of records in the data set. This in turn allows contingency tables, probability tables, entropies, mutual information and chi-squared statistics to be computed almost as rapidly for a data set with 10^9 records as for a data set with 10^5 *once the AD-tree has been built*. As a result, many learning algorithms such as decision trees, association rules and Bayes net structures are dramatically accelerated when faced with large data sets.

1.2 Description of Static AD-trees and Dynamic AD-trees

1.2.1 Motivation and Structure

A static AD-tree is a straightforward implementation of the AD-tree structure described above. A client which uses a static AD-tree must wait for the entire AD-tree to be built before making any conjunctive counting queries. static AD-trees are built in an optimized manner which reduces the number of queries made to the data set. This is roughly a depth-first construction which allows smaller and smaller portions of the data set to be read for each node created. Coupled with the amount of queries eliminated by `MCV`s and other space-saving techniques, the cost of building a static AD-tree is often smaller than the cost of answering the client's queries without an AD-tree. Since the cost of answering any counting query with an AD-tree is constant in the number of data set rows, the cost of answering the client's queries becomes negligible. Therefore if a client will make many unique or repeated queries, static AD-trees have excellent amortized performance (Moore & Lee, 1998). Note that a static AD-tree cannot be changed after it is built.

A dynamic AD-tree is an implementation of the AD-tree structure which attempts to keep all the useful features, while avoiding the long build time and inherent rigidity of static AD-trees. Building an entire AD-tree is seldom necessary for a particular client, thus static AD-trees often waste time and space. Furthermore, if the data set gains new rows, a static AD-tree must be rebuilt from scratch.

dynamic AD-trees contain only a root AD-node at first, growing to reflect the queries made by the client. Doing this while maintaining a static AD-tree's speed is difficult. If a client asks queries covering a small area of the tree, dynamic AD-trees are easily superior. As more of the tree is needed, matching the static AD-tree's amortized speed is more challenging.

If a data set grows after an AD-tree is built, updating all of the nodes in the tree is potentially an exponential operation in the number of attributes. For static AD-trees, rebuilding from scratch is the only option. However, dynamic AD-trees add state at each node to maintain consistency with the data set on a node-by-node basis. Nodes are only updated if and when accessed, achieving consistency with constant amortized complexity.

1.2.2 Implementation of Dynamic AD-trees

A naive implementation of dynamic AD-trees would build the AD-tree nodes as queries were answered, as well as record data set state to maintain consistency when the data set changed. This simple implementation would save space and allow the data set to grow. However, it would be far slower than a static AD-tree due to reading the entire data set for each Vary node and AD-node creation.

In our implementation of dynamic AD-trees, node creation is accelerated using structures we call skinny AD-nodes and row caches. We illustrate their use with an example, depicted in Figure 2. Suppose we are going to query the six row data set of Table 1, using a dynamic AD-tree to improve performance. Before our first query, the dynamic AD-tree contains one AD-node, the root. The root AD-node is equivalent to the most generic query, namely one which specifies no attributes. The count it stores is six, since every data set row matches the most generic query.

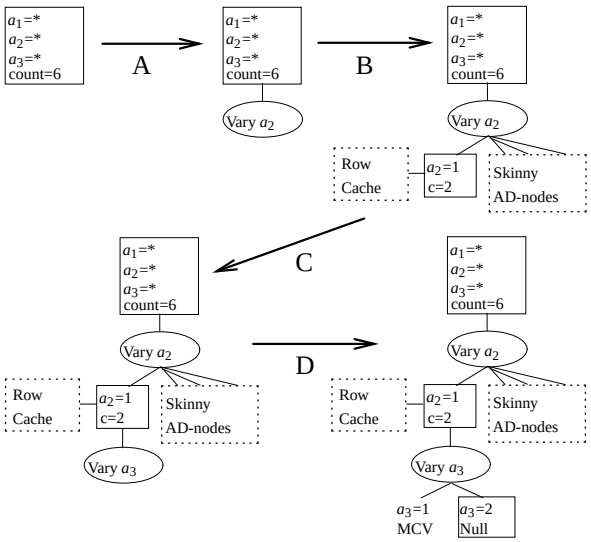

Figure 2. Sequence of dynamic AD-trees on the data set in Table 1, without leaf lists.

We make our first query, C($a_2 = 1 \wedge a_3 = 2$). To answer this query, we create the Vary a_2 beneath the root. This process is represented by arrow A in the figure. This Vary node partitions the data set according to each row's value for the second attribute, and chooses its MCV as the value with the most rows. In this case the row indices are partitioned as $(\{1,4\},\{\},\{2,5,6\},\{3\})$ for values 1, 2, 3, and 4, respectively, and the MCV is 3. The Vary node is now ready to create any children necessary to answer the query. It pulls the first attribute-value pair from our query, $\{a_2 = 1\}$, and makes the corresponding AD-node. This is arrow B in the figure. Since it has already partitioned the data set for attribute two, and since value 1 is not its MCV, the Vary node simply counts the rows in the first partition section and stores this count in the new AD-node. Although there are only two relevant rows, we will neglect leaf lists for purposes of this discussion. In practice, dynamic AD-trees do use leaf lists.

The new AD-node cannot answer the remaining query itself, and hence it will root a new subtree. This subtree represents specializations of the counting query C($a_2 = 1$), and as such it can restrict its scope to the rows matching this query. These rows are exactly the first section of the partition created by our Vary node. Therefore our new AD-node stores this partition section in a row cache, which will be used later when growing the subtree. Without this row cache, future specializations of C($a_2 = 1$) would require reading irrelevant rows of the data set.

The Vary node is not finished yet. Since we may later ask it for a different AD-node child, the Vary node stores the needed partition sections in skinny AD-nodes. A skinny AD-node contains two items: the number of rows in the data set when it was created, and the relevant partition section. In this case the second and fourth partition sections are stored, since the MCV is 3. These skinny AD-nodes are placed underneath the Vary node, exactly where full AD-nodes might later appear. If we ask a Vary node for an AD-node child, and there is a skinny AD-node where the child should be, the Vary node promotes the skinny AD-node to a full AD-node.

Our Vary node has finished its work, and the new AD-node services the remainder $\{a_3 = 2\}$ of the original query. The AD-node creates a child Vary node, namely Vary a_3, following arrow C in the figure. This Vary node partitions the data set rows listed in the AD-node's row cache, and finds that its MCV is value 1. Following arrow D in the figure, the Vary node builds a child AD-node representing $\{a_3 = 2\}$ in order to service the query. Since the second partition section is empty, the Vary node stores a count of zero in the new AD-node. If the count weren't zero, the new AD-node would receive a row cache from the Vary node. The new AD-node is able to answer the query, and no new nodes are built. Note that if a_3 could take any values other than 1 and 2, the Vary node would make skinny AD-nodes for these values. This completes our explanation of skinny AD-nodes and row caches.

static AD-trees never store AD-nodes with a count of zero, resulting in space savings depending upon the sparseness of the data. For dynamic AD-trees, doing this would cause a problem. Suppose we didn't create an AD-node for counts of zero. Since the data set might later grow to include new relevant rows, we would always recheck the entire data set when a query needed the missing AD-node. Therefore we create another specialization of an AD-node, called an 'empty AD-node', which stores only the size of the data set. This size is updated whenever the empty AD-node is accessed. Future queries involving the empty AD-node only need to check for newly arrived relevant rows beyond this stored value. If any are found, the empty AD-node is replaced by a regular AD-node.

The space used by skinny AD-nodes and row caches could quickly grow unacceptably large. Since both structures are used for localized acceleration of growth, any skinny AD-node or row cache may be deleted at any time. We derive optimal benefit from these structures when they exist in areas of the dynamic AD-tree with rapid growth. Therefore we track all skinny AD-nodes in a skinny AD-node queue and all row caches in a row cache queue. Both queues are ordered according to a simple least-recently-used scheme. When the cumulative size of all skinny AD-nodes exceeds a fixed upper bound, the least-recently-used skinny AD-nodes are deleted from the tree and ejected from the queue. The same is true for row caches, using a separate upper bound.

Good sizes for the skinny AD-node and row cache upper bounds depend upon the number of rows and attributes of the data set, correlations between the data set attributes, and the pattern of queries made on the dynamic AD-tree. Note that if a Vary node's `MCV` is dereferenced, all of its AD-node children must exist or be created. Thus this Vary node will have skinny AD-node children for a trivial amount of time. Also, if many similar queries are made within a short space of time, for example trying to find the 'best' value for some attribute while fixing other attributes, the situation is approximately the same. This implies that skinny AD-nodes often have short lives, and the upper bound on the cumulative size of all skinny AD-nodes can typically be made small without a significant performance penalty. However, row caches typically have longer useful lives, and the row cache upper bound should be made larger. Commonly used ratios between the skinny AD-node upper bound and the row cache upper bound are 6:1 and 10:1.

The least-recently-used policy for managing the dynamic AD-tree's queues is not optimal. Sensible policies are defined as any which increase the likelihood of skinny AD-nodes and especially row caches, existing when and where needed. To measure the need, one may consider how many complete reads of the data set are avoided, or how few rows of the data set need to be read as a result of skinny AD-node or row cache existence.

2. Analysis

2.1 Algorithm Descriptions

The results of empirical tests are presented below, in which dynamic AD-tree performance is contrasted with static AD-tree performance. The tests were conducted using two data sets, `e29.fds` and `e46.fds`. These data sets come from the Sloan Digital Sky Survey (SDSS, 1998). Both have 3 million rows, `e29.fds` has 29 attributes, and `e46.fds` has 46 attributes. The first 29 attributes of `e46.fds` are the same as those from `e29.fds`. All attributes used in these tests have a small arity, typically two. One should not take this to mean dynamic AD-trees cannot handle larger arities. We have successfully used dynamic AD-trees on a data set containing 3.5 million rows and 49 attributes, where queries were made on attributes with arities between two and more than two-hundred-thousand. For this latter data set, we were unable to build a static AD-tree within a two gigabyte memory limit.

Three learning algorithms were used: an exhaustive rule learner, a decision tree learner, and a Bayes net structure finder. These algorithms make many queries to the AD-trees, each with a different query pattern. The various patterns have performance implications for dynamic AD-trees due to query-locality effects on skinny AD-nodes and row caches. We give a brief description of each algorithm's query pattern below.

Algorithm	*Query Pattern*
rules	very many highly localized queries to fixed depth, specializing previous queries
decision tree	less localized queries than rule learner, still specializing previous queries
Bayes net	stochastic algorithm making widely varying queries

For static AD-trees the main difference is the number of queries made by the algorithm, and hence they are a good baseline for judging dynamic AD-tree performance. Details of the learning algorithms and their interaction with dynamic AD-trees are presented in the following sections. We also use algorithms accelerated by means of specialized data structures, as opposed to the general purpose AD-trees, that read directly from the data set. These algorithms are referred to below by the term `standard`.

For each learning algorithm, three learning problems were chosen to demonstrate empirical behavior under reasonable real-world use. We will categorize these problems as small `single`, large `single`, and large `multi`. The problems with the tag `single` create a static AD-tree or grow a dynamic AD-tree to solve a single problem. Those with the tag `multi` consist of solving several subproblems of increasing size. A single static AD-tree or dynamic AD-tree is created for the smallest subproblem and reused for the remaining subproblems. In the case of a dynamic AD-tree, the tree grows to accommodate the needs of each subproblem. The `standard` learners cannot perform `multi` runs. Table 2 shows per-algorithm definitions of small and large problems. The terms used in the table are defined later, in sections which describe the learning algorithms in greater detail.

Each `single` or `multi` problem is solved multiple times using different memory constraints. The four constraints used in this paper are 32 MB, 64 MB, 128 MB, and unlimited. These constraints do not affect static AD-trees or the `standard` learners, since we cannot limit their memory use. In our current implementation of dynamic AD-trees, only the skinny AD-node queue and row cache queue sizes have upper bounds. There is no upper bound on the AD-tree size. Therefore we save 30% of the available memory for the tree, assigning 10% to the skinny AD-nodes and 60% to the row caches. Most often this is over-generous for the tree.

Each run was given a maximum allowable time. For `single` runs on the `e29.fds` data set, this was 1200 seconds, and for `single` runs on the `e46.fds` data set it was

4500 seconds. For `multi` runs, the time limit for each subproblem was chosen as if it were a `single` run, and the letters "DNF" indicate that the `multi` run did not finish because some subproblem was aborted.

As a final note, all times reported are system time plus user time, not wall clock time. The machine used for the experiments is a shared resource, and as such there are times reported which show anomalies due to load.

2.1.1 Exhaustive Rule Learner

The exhaustive rule learner discovers good *rules* for the data set, objects which might best be described as approximate implications. Given an output attribute a_{output} and a legal value $\text{val}_{\text{output}}$ for a_{output}, the rule learner searches among conjunctive queries of the form

$$q = \{a_{i_1} = \text{val}_{i_1} \wedge \cdots \wedge a_{i_n} = \text{val}_{i_n}\}$$

to maximize the estimated value

$$P(a_{output} = \text{val}_{output} \mid q) = \frac{C(a_{output} = \text{val}_{output} \wedge q)}{C(q)}$$

To avoid queries without significant support, we insist C(q), the number of records matching query q, must be greater than `minsupport`. Agrawal et al. (1996) and Clark and Niblett (1989) describe examples of rule learners with specialized data structures.

In practice, when searching for the best rule we restrict which attributes may appear on the left-hand-side using an *input list*. Below, we will refer to the input list length as `subset`. We place an upper bound, `numatts`, on the number of attributes in the left-hand-side of a rule.

In all rule learner experiments the AD-tree's leaf list size was 1000, `minsupport` was 100, and the output attribute and value were fixed. We set `subset` to 16 and varied `numatts` while making both `single` and `multi` runs. The results of varying `subset` will be available in a longer version of this paper. The values of `numatts` for the small and large problems are shown in Table 2. Time and memory results for these problems are shown in Tables 3 through 6.

2.1.2 Decision-Tree Induction

The decision tree learner finds a good decision tree for predicting the value of an output attribute, using a greedy C4.5-like strategy (Quinlan, 1993). The decision tree's nodes are restricted to those attributes present in an input list, whose length is denoted `subset`. The number of nodes in the decision tree must be no more than the value of parameter `size`.

To grow the decision tree, the decision learner must choose at each point which attribute will be associated with a new node. Our decision tree learner queries the data set to discover which attribute has the best information gain among the possible attributes. The queries made are very similar to queries previously made when the parent of the new node was created, but have one additional attribute-value pair. When running on an AD-tree, this means that for any new query made, the parent of the AD-node needed to answer that query will already exist. Therefore we expect dynamic AD-trees to perform well when supporting our decision tree learner.

In all decision tree learner experiments, the AD-tree's leaf list size was 1000 and the output attribute was fixed. We varied the `size` parameter, making both `single` and `multi` runs. The definitions of small and large problems may be found in Table 2. The test results are shown in Tables 3 through 6. These results are discussed briefly in section 2.3

2.1.3 Bayes Net Structure Finder

Given a data set, we search the space of Bayes net structures using a backtracking greedy algorithm with random restarting (Pearl, 1996; Heckerman, 1991; Spirtes et al., 1993). The number of random restarts is limited by the `iters` parameter. Our structure finder searches for a Bayes net which describes the entire data set, and is not restricted to considering a subset of the data set attributes.

The structure finder moves greedily between Bayes net structures by changing one dependency edge at a time, trying to improve the current net's score. When using dynamic AD-trees, the scoring function and random restarts often require queries in disparate, long forgotten sections of the AD-tree, allowing row caches and skinny AD-nodes to lapse. We do not expect dynamic AD-trees to be an appropriate choice for the structure finder, since dynamic AD-trees are designed for more localized query patterns and contexts in which making the whole AD-tree is wasteful.

In all Bayes net structure finder experiments, the output attribute was fixed. We varied the parameter `iters`, and conducted both `single` and `multi` runs. The AD-tree's leaf list size was 1000. The definitions of small and large problems may be found in Table 2, and the test results are shown in Tables 3 through 6. These results are discussed briefly in section 2.3

2.2 Exhaustive Rule Learner Performance

2.2.1 Unlimited Physical Memory

Note in Table 5, which summarizes `multi` run performance, that the times for dynamic AD-trees and static AD-trees are similar. That dynamic AD-trees are this competitive on `multi`-type runs can be explained by observing the amount of memory used, also shown in Table 5. Though not shown in Table 3, the cumulative time for a dynamic

Table 3. Performance of dynamic AD-trees and static AD-trees for `single` runs, when memory is unlimited. Note that static AD-trees always use 55MB for the `e29.fds` data set and 489MB for the `e46.fds` data set, and the algorithms themselves use almost no memory. In the table, `STD` refers to the `standard` learner.

		TIME						MEMORY			
		SMALL PROBLEM			LARGE PROBLEM			SMALL		LARGE	
ALGORITHM	DATA SET	DYN	STAT	STD	DYN	STAT	STD	DYN	STD	DYN	STD
RULES	e29.fds	56s	469s	203s	216s	582s	>1200s	84M	23M	137M	NA
	e46.fds	55s	3529s	203s	209s	3688s	>4500s	84M	23M	138M	NA
DECISION TREE	e29.fds	38s	467s	135s	87s	468s	349s	67M	77M	134M	117M
	e46.fds	39s	2927s	136s	89s	2932s	340s	67M	77M	134M	117M
BAYES NETS	e29.fds	63s	465s	105s	449s	485s	>1200s	100M	23M	715M	NA
(STOCHASTIC)	e46.fds	70s	3021s	110s	1229s	3039s	>4500s	86M	23M	1293M	NA

Table 4. Performance of dynamic AD-trees for `single` runs, when memory is limited.

		TIME					
		SMALL PROBLEM			LARGE PROBLEM		
ALGORITHM	DATA SET	DYN128	DYN64	DYN32	DYN128	DYN64	DYN32
RULES	e29.fds	99s	145s	154s	445s	983s	>1200s
	e46.fds	94s	142s	153s	434s	1001s	1868s
DECISION TREE	e29.fds	38s	74s	119s	370s	611s	1181s
	e46.fds	39s	74s	119s	352s	568s	1176s
BAYES NETS	e29.fds	74s	143s	160s	>1200s	>1200s	>1200s
(STOCHASTIC)	e46.fds	88s	132s	149s	>4500s	>4500s	>4500s

Table 2. Parameter Descriptions for various problem types. Note that `subset` is 16 for the exhaustive rule learner and decision tree learner. The Bayes net structure finder always considers all attributes of the data set.

	Rules `numatts`	Deci. Trees `size`	Nets `iters`
Small `single`	2	10	50
Large `single`	8	210	12800
Large `multi`	1 ... 15	10 ... 210, `size+=20`	50 ... 12800, `iters*=4`

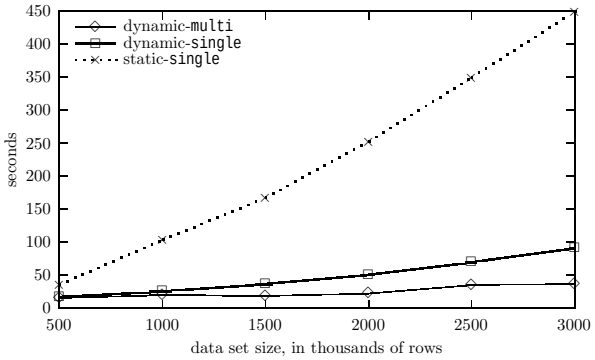

Figure 3. Rule learner times by the size of the data set, which is grown between runs.

AD-tree to complete all subproblems of the large `multi` run in `single` mode is nearly the same as the `multi` run time. This indicates that dynamic AD-tree growth is very efficient when driven by our rule learner, and a client can perform `single` runs with almost no penalty. It is easy to deduce that AD-trees are superior for `multi` runs compared to the `standard` learner. Table 3 shows AD-trees outperforming the `standard` learner for `single` large problems, and dynamic AD-trees as the fastest overall.

2.2.2 LIMITED MEMORY

The limited memory model was explained above. Detailed analyses of dynamic AD-tree performance with the rule learner will be presented in a longer version of this paper. However, times for the three limited memory models are available in Tables 4 and 6.

2.2.3 GROWING THE DATA SET USING `e29.fds`

Figure 3 shows repeated rule learner runs while growing the data set. The times shown are for running the rule learner with `numatts` and `subset` fixed to 3 and 16. The independent variable is the data set size, in thousands of rows. The rule learner is run on the first *n* rows of `e29.fds`, then *n* is increased and the run is repeated. This continues

Table 5. Performance of dynamic AD-trees and static AD-trees for `multi` runs, when memory is unlimited.

		TIME, LARGE multiRun		MEMORY, LARGE multiRun	
ALGORITHM	DATA SET	DYN	STAT	DYN	STAT
RULES	e29.fds	4069s	4142s	137M	55M
	e46.fds	3983s	7200s	138M	489M
DECISION TREE	e29.fds	104s	489s	133M	55M
	e46.fds	107s	3455s	133M	489M
BAYES NETS	e29.fds	472s	490s	707M	55M
(STOCHASTIC)	e46.fds	1163s	3826s	1285M	489M

Table 6. Performance of dynamic AD-trees for `multi` runs, when memory is limited. The entry "DNF" means that one subproblem of the `multi` run was aborted after the preset maximum time.

		TIME, LARGE multi RUN		
ALGORITHM	DATA SET	DYN128	DYN64	DYN32
RULES	e29.fds	4338s	4956s	5733s
	e46.fds	4169s	4806s	5629s
DECISION TREE	e29.fds	389s	628s	1201s
	e46.fds	373s	603s	1197s
BAYES NETS	e29.fds	DNF	DNF	DNF
(STOCHASTIC)	e46.fds	DNF	DNF	DNF

until $n = 3000000$. The static AD-tree times include rebuilding the tree for each run since this is unavoidable, and hence the curve represents a series of `single` runs. There are two curves for dynamic AD-trees. The slower curve represents a collection of `single` runs, and the faster curve represents dynamic AD-trees' ability to make a `multi` run in this context.

From figure 3, one may conclude that dynamic AD-trees are the best choice when data sets are periodically updated and amortized complexity is important. Similar results for the decision tree learner and Bayes net structure finder have been obtained.

2.2.4 RULE LEARNER SUMMARY

Dynamic AD-trees perform competitively with static AD-trees, and frequently are substantially faster. In `multi` problems, both AD-trees thoroughly outperform optimized conventional code. Furthermore, static AD-trees and dynamic AD-trees surpass the `standard` learner on larger problems here and in Moore and Lee (1998), with dynamic AD-trees reporting the fastest times for any `single` run.

2.3 Decision-Tree Learner and Bayes Net Structure Finder Performance

Detailed analyses of dynamic AD-tree performance with the decision tree learner and Bayes net structure finder will be presented in a longer version of this paper. However, time and memory use for all the learners are available in Tables 3 through 6.

Though not presented in this paper, finding a good decision tree on a few hundred nodes is trivial using AD-trees. With dynamic AD-trees, only the necessary parts of the AD-tree are built, saving a vast amount of time over static AD-trees. As seen in the tables, dynamic AD-trees may be both faster and smaller than the specialized `standard` decision tree learner.

On the other hand, the Bayes net structure finder does not benefit from dynamic AD-trees as much as the other learners. It reveals critical weaknesses in dynamic AD-trees, causing them to use large amounts of memory as reported in Tables 3 through 6. That said, the scaling behavior of dynamic AD-trees allows them to take over where static AD-trees falter, also shown in these tables. As before, the `standard` learners are unable to handle large problems. Finally, note that dynamic AD-trees can be used to provide better performance than static AD-trees and conventional optimized code, when memory is plentiful.

3. Future Work

It may be useful to support the dynamic addition of attributes to the underlying data set, as well as dynamic reordering of existing attributes. Both of these features are easily implemented using dynamic AD-trees in a manner which amortizes the possibly exponential complexity.

Currently AD-trees only allow attributes to have discrete values, which clouds their statistical usefulness for many real-world applications. One solution is to dynamically add artificial attributes, representing inequality tests, to more finely discriminate between real values in data set rows when needed. Another solution uses dynamic attribute resolution, whereby an existing attribute's arity can be dynamically increased. An attribute's new values would represent additional decimal places of accuracy as requested by a query. Both of these may be implemented simply using dynamic AD-trees.

It may be desirable to allow some parameters to vary within the tree, such as leaf list size. More interesting is to locally vary which attributes appear, and which values they take, within an AD-tree. For instance both ideas above for handling real values may lead to an undesirable explosion of attributes or attribute arities in an AD-tree, unless local variation of attributes and attribute arity is allowed. This feature could also reduce the size of purely symbolic AD-trees, if a client conditionally discriminates on an attribute's values. Dynamic AD-trees are particularly suited to local variation because inadequate or excessive specialization can be automatically and transparently repaired later.

Initial work for transparently compressing leaf lists has commenced, and indicates that an approximate compression of 2:1 is possible without significant performance degradation.

A further extension of dynamic AD-trees would allow pruning. This pruning could be managed by a client, or handled internally. For instance, if an algorithm knew it was finished using a specific subtree, it could instruct the dynamic AD-tree to remove it. The subtree would be regrown automatically if it were needed later. Internally, a dynamic AD-tree could use a priority queue to prune infrequently used AD-nodes. Such a mechanism would obviate skinny AD-nodes.

4. Conclusion and Acknowledgements

The AD-tree structure has proven itself useful for accelerating conjunctive counting queries, here and in Moore and Lee (1998). It derives its advantage over traditional specialized data structures by caching the query results, thus reducing the number of data set passes. In effect, AD-tree performance is largely unaffected by the number of rows in the data set. The dynamic AD-tree's client-driven nature implies that its performance is virtually independent of the number of attributes in the data set. This allows dynamic AD-trees to scale up to larger data sets than static AD-trees can handle, and scale down to provide strong performance for clients which make few or very specialized queries. The extra state maintained by dynamic AD-trees provides greater flexibility as well, allowing new rows to be added to the data set between queries. For many learning problems, these properties make dynamic AD-trees the preferred choice over both existing specialized data structures and static AD-trees. These problems include rule learning, decision-tree induction, and discovering good Bayes net structures on large data sets.

This work was sponsored by an NSF KDI grant to Andrew Moore. The award number is DMS-9873442.

References

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., & Verkamo, A. I. (1996). Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*. AAAI Press.

Anderson, B., & Moore, A. W. (1998). AD-trees for fast counting and rule learning. *Proceedings of the Fourth International Conference on Knowledge Discovery in Data Mining* (pp. 134–138). AAAI Press.

Clark, P., & Niblett, R. (1989). The CN2 induction algorithm. *Machine Learning, 3,* 261–284.

Heckerman, D. (1991). *Probabilistic similarity networks*. MIT Press.

Moore, A. W., & Lee, M. S. (1998). Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research, 8*.

Pearl, J. (1996). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. San Mateo: Morgan Kaufmann.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. San Mateo: Morgan Kaufmann.

SDSS (1998). The Sloan digital sky survey. `www.sdss.org`.

Spirtes, P., Glymour, C., & Scheines, R. (1993). *Causation, prediction, and search*. New York: Springer-Verlag.