ADtrees for Fast Counting and for Fast Learning of Association Rules

Brigham Anderson
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
brigham@andrew.cmu.edu

Andrew Moore
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
awm@cs.cmu.edu

Abstract: The problem of discovering association rules in large databases has received considerable research attention. Much research has examined the exhaustive discovery of all association rules involving positive binary literals (e.g. Agrawal et al. 1996). Other research has concerned finding complex association rules for high-arity attributes such as CN2 (Clark and Niblett 1989). Complex association rules are capable of representing concepts such as "PurchasedChips=True and PurchasedSoda=False and Area=NorthEast and CustomerType=Occasional ⇒ AgeRange=Young", but their generality comes with severe computational penalties (intractable numbers of preconditions can have large support). Here, we introduce new algorithms by which a sparse data structure called the ADtree, introduced in (Moore and Lee 1997), can accelerate the finding of complex association rules from large datasets. The ADtree uses the algebra of probability tables to cache a dataset's sufficient statistics within a tractable amount of memory. We first introduce a new ADtree algorithm for quickly counting the number of records that match a precondition. We then show how this can be used in accelerating exhaustive search for rules, and for accelerating CN2-type algorithms. Results are presented on a variety of datasets involving many records and attributes.

Keywords: Data Mining, Association Rules, CN2, Condensed Representations.

Introduction

Sets of if-then rules are expressive and human readable representations of learned hypotheses. Finding association rules in databases is an interesting and profitable undertaking. The rules one might search for could be of the form "if workclass=private and education=12+ and maritalstatus=married and capitalloss=1600+, then income $\Rightarrow$ 50K+ with 96% confidence." Association rules can be useful in industry. For instance, the above example could help target income brackets.

However, learning these rules is often difficult. There are at least two approaches taken in learning association rules:

1. Only use positive binary literals in the rule. This approach is often used for supermarket "basket" data (Agrawal et al, 1996). Limiting the elements of the rule to only positive binary literals allows the algorithm to efficiently search the space of hypotheses using frequent sets.
2. If the database or rules one is interested in contain attributes of higher arities, then exhaustive search is prohibitive. CN2-type algorithms (Clark and Niblett, 1989) can perform a general-to-specific beam search that allows the program to heuristically search the hypothesis space and thus dramatically reduce the number of rules it considers.

For datasets with symbolic attributes, both exhaustive search and CN2 search can be helped by ADtrees (Moore and Lee, 1997). This paper introduces a new algorithm that permits fast counting queries, often several magnitudes faster than iterating through the dataset. Faster queries mean faster search because the bulk of work in these domains is usually in the rule evaluation step. The paper then benchmarks the speedup that the ADtree can produce. Finally, a CN2-type algorithm is augmented by use of the new counting algorithm and gains in speed are measured.

Problem definition

Consider a database of *R* records with symbolic attributes. A database could be, for example, a list of loan applicants where each entry has a list of attributes such as type of loan, marital status, education level, and income range, and each attribute has a value. A record thus has *M* attributes, and is represented as a single vector of size *M*, each element of which is symbolic. The attributes are called a_1, a_2, … a_M. The value of attribute a_i in a record is represented as a small integer lying in the range $\{1,2,\ldots n_i\}$ where n_i is called the *arity* of attribute *i*.

In our definition of association we followed Agrawal, et al. (1996). Their definition of an association rule is a conjunction of attributes implies a conjunction of other attributes. Here, the terminology is slightly different; we define a *literal* as an attribute-value pair such as "education = masters". Let L be the set of all possible literals for a database. An *association rule* is an implication of the form S1 $\Rightarrow$ S2, where S1, S2 $\subset$ L, and S1 $\cap$ S2 = $\varnothing$. S1 is called the *antecedent* of the rule, and S2 is called the *consequent* of the rule. We thus denote association rules as an implication of sets of literals. An example of an association rule is "gender=male and education=doctorate $\Rightarrow$ maritalstatus=married and occupation=prof-specialty".

Each rule has a measure of statistical significance called *support.* For a set of literals S $\subset$ L, the *support* of S is the number of records in the database that match all the attribute-value pairs in S. Denote by *supp*(S) the *support* of S. The support of the rule S1 $\Rightarrow$ S2 is defined as *supp*(S1 $\cap$ S2). Support is a measure of the statistical significance of a rule. A measure of its strength is called *confidence*, and is defined as the percentage of records that match S1 and S2 out of all records that

match S1. In other words, the support of a rule is the number of records that both the antecedent and consequent literals match. The confidence is the percentage that the supporting records represent out of all records in which the antecedent is true.

This paper considers the problem of mining association rules to predict a user-supplied target set of literals S2. The objective is to find rules of the form S1⇒S2 that maximize confidence while keeping support above some user-specified minimum (*minsupp*). One version of this generation procedure is to return the best *n* rules encountered, or perhaps all rules above a certain confidence.

Generation of such rules requires calculating large numbers of rule confidences and supports. Rule evaluation thus requires two calculations, *supp*(S1) and *supp*(S1 ∩ S2). These two numbers give both the support and the confidence of the rule S1⇒S2. One method for calculating a *supp*(S) is to run through every relevant record and count the number of matches. Another method is to use some way of

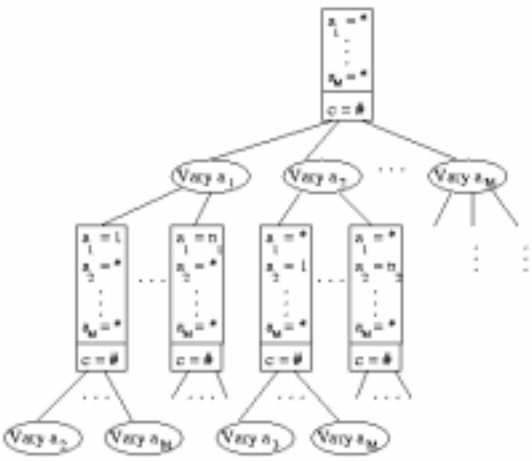

Figure 1: The top ADnodes of an ADtree, described in the text

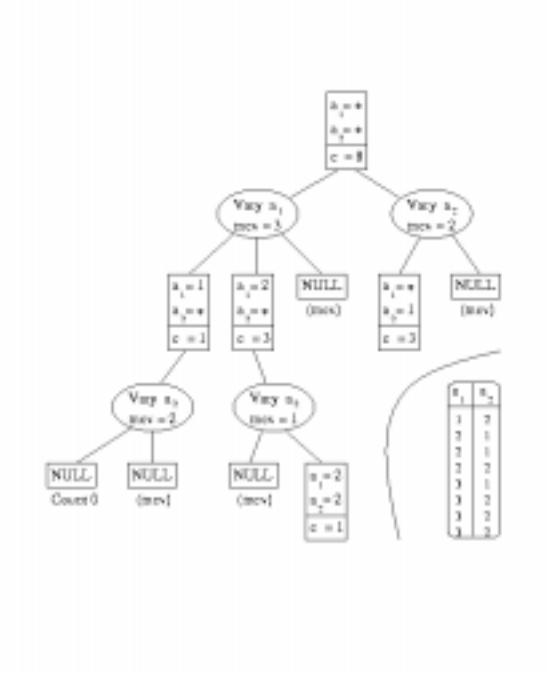

Figure 2: A sparse ADtree built from the dataset in the bottom right. The most common value for a_1 is 3, and so the $a_1 = 3$ subtree of the Vary a_1 child of the root node is NULL. At each of the Vary a_2 nodes the most common child is also set to NULL (which child is most common depends on the context.)

caching statistics that allows calculating these numbers directly, such as an ADtree (described below). A third possibility is to build all queries having adequate support with sequential passes through the dataset (Agrawal et al. 1996). This is very effective if only positive binary literals are being found, but if negative literals are also required the number of rule sets found will be intractably large: $O(2^M)$.

ADtree Data Structure

If we are prepared to pay a one-time cost for building a caching data structure, then it is easy to suggest a mechanism for doing counting in constant time. For each possible query, we precompute the count. The total amount of numbers stored in memory for such a data structure would be

$$(arity_1 + 1)(arity_2 + 1) \ldots (arity_M + 1)$$

For a real dataset with more than ten attributes of medium arity, or fifteen binary attributes, this is far too large to fit in main memory.

We would like to retain the speed of precomputed counts without incurring an intractable memory demand. That is the purpose of ADtrees. An ADNODE (shown as a rectangle in Figure 1) has child nodes called "Vary nodes" (shown as ovals).

Each ADNODE represents a query and stores the number of records that match the query. The "Vary a_j" child of an ADNODE has one child for each of the $arity_j$ values of attribute a_j. The kth child represents the same query as "Vary a_j"'s parent, with the additional constraint that $a_j = k$.

Although drawn on the diagram, the description of the query (e.g., $a_1 = 1$, $a_2 = * \ldots a_M = *$) on the leftmost ADNODE of the second level) is not explicitly recorded in the ADNODE. The contents of an ADNODE are simply a count and a set of pointers to the "Vary a_j" children. The contents of

a "Vary a_j" node are a set of pointers to ADNODEs. Notice that if a node ADN has "Vary a_i" as its parent, then ADN's children are

$$\text{"Vary } a_{i+1}\text{"}, \text{"Vary } a_{i+2}\text{"}, \ldots \text{"Vary } a_M\text{"}$$

It is not necessary to store Vary nodes with indices below i+1 because that information can be obtained from another path in the tree.

As described so far, the tree is not sparse and contains every possible count. Sparseness is easily achieved by storing a NULL instead of a node for any query that matches no records. All of the specializations of such a query also have a count of zero and they will not appear anywhere in the tree. This helps, but not significantly enough to be able to cope with large numbers of attributes.

To greatly reduce the tree size further, we will take advantage of the observation that very many of the counts stored in the above tree are redundant. For each vary node, we will find the most common of the values of a_j (call it MCV) among records that match the node and we will store a NULL in place of the MCVth subtree. The remaining ($arity_j$-1) subtrees will be represented as before. An example for a simple dataset is given in Figure 2. Each "Vary a_j" node now records which of its values is most common in a MCV field. (Moore and Lee 1997) describes the straightforward algorithm for building such an AdTree.

Removing most-common-values has a dramatic effect on the amount of memory needed. If we don't remove most-common-values then we must store one count for every possible query for which the count is greater than zero. But when we've removed most-common-values the memory used is (empirically, on large datasets) reduced by factors greater than a billion. This is because we only store counts for queries in which every component of the query is a "surprise". In order to appear in the ADtree, a counting query

$$a_{i(1)} = v_1 \text{ , } a_{i(2)} = v_2 \ldots a_{i(m)} = v_m$$

must be such that v_1 is a "surprising" (i.e. not the most common value) for $a_{i(1)}$ AND v_2 is a "surprising" (i.e. not the most common value) for $a_{i(2)}$ among those records in which $a_{i(1)} = v_1$ AND . . . . . .v_k is a "surprising" (i.e. not the most common value) for $a_{i(k)}$ among those records in which $a_{i(1)} = v_1$, $a_{i(2)} = v_2$, ... $a_{i(k-1)} = v_{k-1}$..... AND the query must have a non-zero count.

The consequence of this is that for datasets with M binary attributes:

- If we do not use the "most-common-value" memory reduction, the worst case number of counts that need to be stored is 3^M while the best possible case (e.g. if there were only one record) is 2^M.
- If we do use the "most-common-value" reduction, the worst case is 2^M, and the best case is M. Furthermore, (Moore and Lee 1997) show that if the number of records is less than 2^M, or if there are correlations, or non-uniformities among the attributes then the number is very much less than 2^M. As we shall see shortly, this is borne out empirically.

Notice in Figure 2 that the MCV value is context dependent. Depending on constraints on parent nodes, a_2's MCV is sometimes 1 and sometimes 2. This context dependency can provide dramatic savings if (as is frequently the case) there are correlations among the attributes. This point is critical for reducing memory, and is the primary difference between the use of ADtree versus the use of Frequent Sets (Agrawal et al, 1996) for representing counts. (Mannila and Toivonen, 1996).

Usage in Fast Counting

In Moore & Lee (1997) an algorithm was presented and discussed for quickly building contingency tables for subsets of variables in constant time. Contingency tables are very closely related to probability tables in the Bayes net community or DataCubes (Harinarayan et al, 1996) in the database community.

Here we show how the ADtree can also be used to produce counts for specific queries in the form of a set of literals. For example, one can calculate the number of records having {a1=12, a4=0, a7=3, a8=22} directly from the ADtree. The following algorithm returns these types of counts:

```
Preconditions:
Query_list ← list of attribute-value pairs sorted by attribute
Index ← 0
Current_ADnode ← root ADNODE of ADtree

AD_COUNT(ADnode, Query_list, index)
If index equals the size of Query_list then
   Return ADnode's count

Varynode ← Vary node child of ADnode that corresponds to indexth attribute in Query_list
Next_ADnode ← ADNODE child of Varynode that corresponds to indexth value in Query_list

If Next_ADnode's count is 0 then
   Return 0

If Next_ADnode is a MCV then
   Count ← AD_COUNT(ADdnode, Query_list, index + 1)
   For each s in siblings of Next_ADnode do
      Count ← Count - AD_COUNT(s, Query_list, index+1)
   Return Count

Return AD_COUNT(Next_ADnode, Query_list, index+1)
```

Figure 3: Pseudocode for AD_COUNT, an algorithm that returns the number of records matching a given list of literals.

The important step in this recursive algorithm is the point at which we manage to survive the absence of the MCV of an attribute. We simply use the fact that

$$\text{Count}(a_{i(k+1)} = v_{(k+1)} \ldots a_{i(n)} = v_n) = \sum_{j=1}^{arity_{i(k)}} \text{Count}(a_{i(k)} = j, a_{i(k+1)} = v_{(k+1)}, \ldots a_{i(n)} = v_n)$$

And so

$$\text{Count}(a_{i(k)} = \text{MCV}, a_{i(k+1)} = v_{(k+1)}, \ldots a_{i(n)} = v_n) = \text{Count}(a_{i(k+1)} = v_{(k+1)}, \ldots a_{i(n)} = v_n) - \sum_{j \neq \text{MCV}}^{arity_{i(k)}} \text{Count}(a_{i(k)} = j, \ldots a_{i(n)} = v_n)$$

This flavor of subtraction trick is also used in the contingency table construction method of Moore and Lee and is the "condensed representation" use of frequent sets for counting described by (Mannila and Toivonen, 1996).

Comparison of Rule Evaluation Speed

Evaluation of a rule S1 $\Rightarrow$ S2 only requires calculating *supp*(S1) and *supp*(S1 $\cap$ S2). A simple use of the ADtree is to return numbers of records matching simple queries which are conjunctions of literals, such as "in the ADULT1 dataset, how many records match {income=50K+, sex=male, education=HS}?" The answer can be returned by a simple examination of the tree, usually several orders of magnitude faster than going through the entire dataset. What that means in this particular application is that rules can be evaluated more quickly, and thus can be learned faster.

Since counting is so important to rule learning, we compare here the performance of ADtree counting against straightforward searching through the database. The comparison results are in Table 2. To generate the results, we generate many random queries, count them, and time the procedure. Each randomly generated query consists of a random set of attribute-value pairs from a randomly selected record from the database. Generating queries this way ensures that the corresponding count is at least one matching record. The numbers in Table 2 represent the ratio of the mean time to return a count on a set of literals by running through the entire dataset versus the mean time to return a count using an ADtree.

Why create the random queries as subsets of literals of existing records? Would it not have been simpler to generate entirely random queries? The reason is that completely random queries usually have a count of zero. The ADtree can discover this extremely quickly, giving an even larger advantage over direct counting For instance, for size 20 randomly generated queries on the BIRTH dataset, the ADtree's performance was 111 times better than regular searching of the entire dataset. Compare this to Table 2, where the speedup is only 2.5.

Name	R=Num Records	M=Num Attributes	Description	Tree Size (nodes)	Tree Size (MB)	Build Time (sec)
ADULT1	15060	15	The small "Adult Income" dataset placed in the UCI repository by Ron Kohavi. Contains census data related to job, wealth, and nationality. Attribute arities range from 2 to 41. In the UCI repository this is called the Test Set. Rows with missing values were removed.	58200	7.0	6
ADULT2	30162	15	The same kinds of records as above but with different data. The Training Set.	94900	10.9	10
ADULT3	45222	15	ADULT1 and ADULT2 concatenated.	162900	15.5	15
BIRTH	9672	97	Records concerning a very wide number of readings and factors recorded at various stages during pregnancy. Most attributes are binary, and 70 of the attributes are very sparse, with over 95% of the values being FALSE.	87400	7.9	14
MUSHRM	8124	22	A database of wild mushroom attributes compiled from the Audubon Field Guide to Mushrooms by Jeff Schlimmer and taken from the UCI repository. Attribute arities range from 2 to 12.	45218	6.7	8
CENSUS	142521	13	A larger dataset than ADULT3, based on a different census. Also provided by Ron Kohavi. Arity ranges from 2 to 15.	24007	1.5	17

Table 1: Datasets used to produce experimental results. The size of the ADtrees used with each dataset is included both in the number of nodes in the ADtree and in the amount of memory the tree used. The preprocessing time cost is given also.

Dataset\Rulesize limit	2	4	6	8	10	15	20
ADULT1	1019.2	208.3	76.2	36.9	24.6		
ADULT2	1980.6	361.7	130.5	58.8	36.1		
ADULT3	2782.7	508.4	166.8	71.0	46.1		
BIRTH	1494.3	272.9	86.5	37.3	19.2	5.9	2.5
MUSHRM	881.5	319.7	179.3	110.4	82.2	46.0	27.7
CENSUS	10320.3	1139.8	261.3	105.3	60.8		

Table 2: Speedup ratio of average time spent counting a query of a given size not using ADtree vs. when using ADtree.

Rule-Learning Algorithm

We now look at how the fast counting method of the previous section can accelerate rule-finding algorithms. We look at CN2 (Clarke and Niblett, 1989), an algorithm that finds rules involving arbitrary literals, not merely positive binary literals. In our experiments, only the most confident antecedents for S2 are sought instead of attempting to cover the entire dataset. The learning algorithm is given S2 and begins search at S1 = {}. It then progressively adds literals one at a time, retaining the best k rules from each generation and remembering the best-ever performers. This process continues until the *minsupp* condition can no longer be satisfied. In this way, the search considers increasingly specific rules using a breadth-first beam search with beam size *k*.

```
LEARN-ONE-RULE(Target-attribute, Attributes, Examples, k)
Initialize Best_hypothesis to the most general hypothesis, h = {}
Initialize Candidate_hypotheses to the set {Best_hypothesis}
While Candidate_hypotheses is not empty, Do
        Generate the next more specific candidate_hypotheses
                New_candidate_hypotheses ←
                        for each h in Candidate_hypotheses,
                                for each c in All_literals,
                                        Create a specialization of h by adding the literal c
                Remove from New_candidate_hypotheses any hypotheses that are duplicates
        Update Best_hypothesis
                For  each h in New_candidate_hypotheses do
                        If (PERFORMANCE(h, Examples, Target_literal) > PERFORMANCE(Best_hypothesis,
                        Examples, Target_literal))
                        Then Best_hypothesis ← h
        Update Candidate_hypotheses
                Candidate_hypotheses ← the k best members of New_candidate_hypotheses, according to the
                PERFORMANCE measure
Return a rule
        "IF Best_hypothesis THEN Target_literal"

PERFORMANCE(h, Examples, Target_literal)
        Return supp(h ∩ Target_literal) / supp(Target_literal)
```

Figure 4: Pseudocode for the CN2 algorithm variant used.

Comparison of Rule Learning Speed

The following is a comparison of ADtree-assisted CN2 beam search rule learning and regular beam search. The parameters for the CN2 search were a beam size of 4, a minsupp of 200. Rules

learned were of the form S1⇒S2. The average time to learn a best rule for a randomly generated target literal, S2, was regarded as the "rule-learning time". The target literal S2 was restricted to being a single literal, where that literal's attribute was first selected randomly from all possible attributes for the dataset, then a value was randomly assigned from the set of values that the attribute could take on. Rule-learning times for normal CN2 and for CN2-with-ADtree were both recorded. Table 3 reports the ratio of these two averages for different rule size limits. Rule size is defined as the number of literals in S1 plus the number of literals in S2.

Rule Size	4			8			16		
	Regular Time (sec)	ADtree Time (sec)	Speedup	Regular Time (sec)	ADtree Time (sec)	Speedup	Regular Time (sec)	ADtree Time (sec)	Speedup
ADULT1	2.8	0.041	68.1	2.9	0.09	31.8	2.9	0.12	23.8
ADULT2	5.4	0.041	132.7	5.5	0.16	34.0	5.7	0.22	26.3
ADULT3	8.7	0.049	178.2	8.5	0.10	84.9	8.4	0.12	69.3
BIRTH	4.3	0.16	26.9	5.1	1.4	3.6	6.3	13.1	0.5
MUSHRM	1.8	0.039	46.6	1.8	0.064	28.5	1.9	0.094	19.9
CENSUS	16.6	.058	286.8	15.8	0.22	71.1	16.1	0.26	61.3

Table 3: CN2 speedup when ADtree is used with different rule size limits.

As can be seen, there is a general and large speedup achieved from using ADtree evaluation on these datasets.

Advantages and Disadvantages

Our implementation of the plain counting version of CN2 was as efficient as possible. In particular, we exploited a major speedup in the original CN2 algorithm: the fact that, as candidate rules are made more specific, they are relevant to only a subset of the records relevant to their parent rule. Since it is the case that no more specific descendent of a rule can ever match records that the parent rule did not, a running list of relevant examples is kept for each rule. This list is pruned each time that the rule is made more specific, and can drastically reduce the number of records that the algorithm needs to look at in order to evaluate a rule as it grows. Thus search speeds up dramatically near the end of the search.

The ADtree, on the other hand, cannot use this information; it will always return a count for the entire dataset. The ADtree is at an additional disadvantage when evaluating specific rules, simply because specific rules tend to have more literals. For ADtrees, rules with more literals take more time to evaluate. So, the CN2 algorithm using an ADtree becomes less advantageous as rules grow more specific.

Noticeably, the BIRTH dataset performance is the poorest. Although this dataset has the largest number of attributes per record (97), the speed of lookup in ADtrees is only slightly affected by this quantity, so why the huge difference? Primarily, the difference is due to the types of random target (S2) literals generated to test rule learning speed. The random target literal for the BIRTH dataset was (a_x=0) 50% of the time, so the algorithm was attempting to predict literals of the type "diabetes=no". These types of non-events can usually be best predicted by rules with large numbers of other non-events in the antecedent, like "hemorhage-in-3rd-trimester=no" and "overweight=no". Moreover, the sparseness of the BIRTH dataset causes searches of the ADtree to en-

counter more MCVs and thereby spawn longer searches. It is thus the prediction of non-events and the sparseness of the dataset that bias the relative performance of ADtree downwards. As demonstration of this, when S2 is restricted to be an actual "event" literal, learning rules of size less than or equal to 16 from BIRTH is 2.9 times faster with an ADtree.

Flexibility

It is the general-to-specific search that allows the CN2 algorithm to prune the example set. One advantage of the ADtree stems from its lack of reliance on the general-to-specific nature of the search to aid rule evaluation. One can even use exhaustive search if one wants relatively short rules (see Table 4 and **Error! Reference source not found.** for empirical results).

Comparison of Coverage

One major drawback of the use of ADtrees for rule evaluation is their inability to perform the kind of tilings of rules that the original CN2 method could perform. In CN2, once a rule is found that covers a certain subpopulation of the examples, those examples are removed from consideration and another rule is found. This is rarely practical with an ADtree. One would have to either re-build the ADtree from scratch without the covered examples, or one would have to modify the query sent to the ADtree which excludes all examples matching a certain description

On other hand, tiling is not always needed. Targeted rule finding can be more appropriate when one has specific literals (like breastcancer=true) one is interested in predicting or learning more about. This type of usage would be more often seen in interactive mining of databases.

	ADULT2				BIRTH			
Rule Size	**Number Rules**	**ADtree Time**	**Regular Time**	**Speedup Factor**	**Number Rules**	**ADtree Time**	**Regular Time**	**Speedup Factor**
1	116	.000019	.0056	295.0	194	.000025	.0072	286
2	4251	.000019	.0014	75.3	17738	.000021	.0055	259
3	56775	.000024	.00058	23.8	987134	.000022	.0040	186
4	378984	.000031	.00030	9.8	37824734	.000024	.0030	127
5	1505763	.000042	.00019	4.7	1077672005	.000026		

Table 4: ADULT2 and BIRTH exhaustive search times

Discussion

The current implementation is restricted to all symbolic attributes. Furthermore, The current implementation assumes that the dataset can be stored in main memory. This is frequently not true. Work in progress (Davies & Moore 1998) introduces algorithms for building ADtrees from sequential passes through the data instead of by random access.

A disadvantage of ADtrees for rule learning is that they cannot be easily used to do "tiling" of datasets. Exclusive use of ADtree rule evaluation has benefits, though. Mainly, one is not restricted to general-to-specific search.

Combining the flexibility of use of ADtrees with their speed could make rule learning of databases a process that can take place at interactive speeds. Relevant rules can be located quickly and perhaps enhanced by the operator's knowledge, then resubmitted to the program for further polishing of the hypothesis. The ADtree is a tool for creating a fast and interactive rule learning programs.

References

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

Clark, P., & Niblett, R. 1989. The CN2 induction algorithm. *Machine Learning* 3:261-284.

Davies, S. and Moore, A. W.. 1998, Lazy and sequential ADtree construction. In preparation.

Harinarayan, V, Rajaraman, A. and Ullman, J. D., 1996, Implementing Data Cubes Efficiently. In Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems : (PODS 1996), Assn for Computing Machinery. Pages 205-216.

John, G. H., and Lent, B., 1997, SIPping from the data firehose. In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, AAAI Press, 1997

Mannila, H., and Toivonen, H., 1996, Multiple uses of frequent sets and condensed representations. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, edited by Simoudis, E., and Han, J., and Fayyad, U. AAAI Press.

Mitchell, T. 1997. *Machine Learning*. McGraw-Hill.

Moore, A.W., and Lee, M.S.,1997, Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. CMU Robotics Institute Tech Report TR CMU-RI-TR-97-27. *(Accepted for publication in Journal of Artificial Intelligence Research)*