2008-01-01

Adapting ADtrees for High Arity Features

Irene Langkilde-Geary

Robert Van Dam
rvandam00@gmail.com

See next page for additional authors

Original Publication Citation
Robert D. Van Dam, Irene Geary and Dan Ventura, "Adapting ADtrees for High Arity Features", Proceedings of the Association for the Advancement of Artificial Intelligence, pp. 78-713, 28.

Adapting ADtrees for High Arity Features

Robert Van Dam, Irene Langkilde-Geary and **Dan Ventura**
Computer Science Department
Brigham Young University
rvandam@cs.byu.edu, i.l.geary@gmail.com, dan@cs.byu.edu

Abstract

ADtrees, a data structure useful for caching sufficient statistics, have been successfully adapted to grow lazily when memory is limited and to update sequentially with an incrementally updated dataset. For low arity symbolic features, ADtrees trade a slight increase in query time for a reduction in overall tree size. Unfortunately, for high arity features, the same technique can often result in a very large increase in query time and a nearly negligible tree size reduction. In the dynamic (lazy) version of the tree, both query time and tree size can increase for some applications. Here we present two modifications to the ADtree which can be used separately or in combination to achieve the originally intended space-time tradeoff in the ADtree when applied to datasets containing very high arity features.

Introduction

The All-Dimensions tree (ADtree) was introduced as a generalization of the *k*d-tree designed to store sufficient statistics for symbolic datasets (Moore & Lee 1998). It was later adapted to grow lazily to meet the needs of a client's queries (Komarek & Moore 2000). More recently, the ADtree was modified to sequentially update for incremental environments (Roure & Moore 2006). This paper will focus on improving the static and dynamic versions of the tree but the results presented should be applicable to a sequentially updated tree as well.

The design of the ADtree permits it to scale efficiently to very large datasets (in terms of number of rows in the dataset). Previous results have been reported for datasets with as many as 3 million rows (Moore & Lee 1998; Komarek & Moore 2000). However, the size of the ADtree is determined by the number of feature-value combinations (and therefore the number of features and their arity). The number of rows in the dataset only influences the tree by limiting the number of possible combinations (most real world datasets have far fewer rows than possible feature-value combinations).

The full ADtree (see Figure 1) contains two types of nodes which alternate along every path in the tree. ADnodes (squares) store the count of one conjunctive query. The children of ADnodes are known as Vary nodes (circles). Vary nodes do not store counts but instead group ADnodes according to a single feature. The Vary node child of an ADnode for feature a_i has one child for each value v_j. These grandchildren ADnodes specialize the grandparent's query Q by storing the counts of $Q \wedge a_i = v_j$.

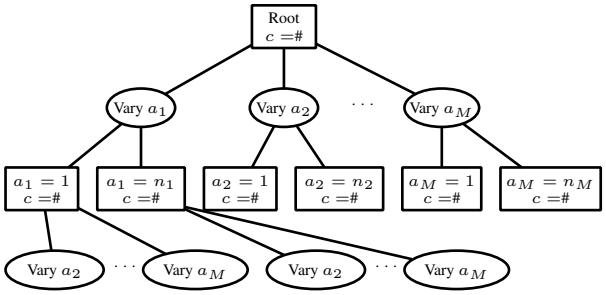

Figure 1: Top levels of a generic ADtree

This full ADtree contains every combination of feature-value pairs and is not yet efficient in its memory usage. The original ADtree included three approaches which can be combined to reduce its overall size. The tree can be made sparse by removing all zero counts. Additionally, the ADnodes near the bottom of the tree are not expanded. Instead they are replaced with "leaf lists" of indices into the dataset whenever the number of relevant rows (the count) drops below a pre-determined threshold.

These two modifications only afford minimal (if any) space savings. The last space-saving technique originally introduced reduces the tree size dramatically by removing counts which can be recovered from other counts already stored in the tree. Since the ADnode grandchildren of an ADnode for query Q represent all non-zero specializations of Q, the count of Q is equal to the sum of their counts. Therefore, the count of one of these ADnodes can be recovered by subtracting the sum of its siblings from the count of the grandparent (Moore & Lee 1998).

If k is chosen such that v_k is the Most Common Value (MCV) or the largest among its siblings, then its removal provides the largest expected space-savings without sacrificing the ability to recover any counts. For a tree built for

$a_1 a_2 a_3$	Count
0 0 0	1
0 0 1	2
0 2 1	4
1 0 1	8
1 1 1	16
1 2 0	32

Table 1: Sample dataset for the ADtree in Figure 2

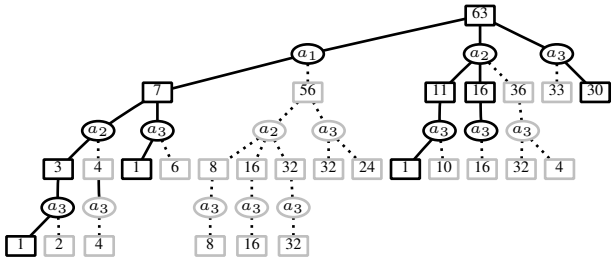

Figure 2: An example of a simple ADtree

M binary features, the worst case size of the tree is reduced from 3^M to 2^M upon removal of the MCV ADnodes. This technique assumes that the trade-off of increasing average query time is reasonable in comparison to the expected space savings.

The tree shown in Figure 2 illustrates what an ADtree generated according to the dataset in Table 1 would look like. The nodes shown in gray are those nodes which would have existed in a full tree but are left out according to the MCV based space savings rule. As can be seen in this tree, leaving out an MCV node can reduce the overall size of the tree by much more than just a single node. At the same time, the count corresponding to every one of the "missing" nodes can still be reconstructed from the remaining nodes.

The dynamic tree follows the same basic structure as described above. However, since it is built lazily, at any given point only a portion of the tree will have been expanded. The dynamic tree also contains some additional support info used to temporarily cache information needed for later expansion. Fully expanded portions of the tree no longer require these extra support nodes.

Modifications

It is noteworthy that the majority of previously published results have used datasets with relatively low arity features. Although there is no technical limit to the arity of features used in an ADtree, there are practical limitations. In particular, the space-saving technique of removing all MCV ADnodes makes the assumption of a reasonable space-time trade-off that is more easily violated with high arity features. Dynamic trees can suffer additional problems when removing MCV nodes even for low arity features. Below we will present two modifications to the ADtree which decrease the overall space usage when using high arity features while still mainintaing reasonable built/query times.

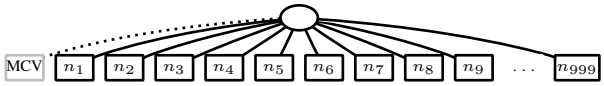

Figure 3: Vary node for a high arity feature

the worst-case query time when using high arity features while still maintaining reasonable space bounds.

The key problem is that removing the MCV ADnode for high arity features can dramatically increase query time and only slightly decreases space usage. If a query involves the MCV of feature a_i, it is necessary to sum at least $n_i - 1$ values (where n_i is the arity of a_i). Given a query Q of size q, the worst case scenario could require summing over $\prod_{i=1}^{q} n_i$ values due to recursive MCV "collisions". Although this worst case is rare (since data sparsity and/or correlation tend to cause Vary nodes lower in the tree to have fewer ADnodes), it helps to illustrate the potential for longer query times when the n_i are large. For instance, a query involving 3 features with 10 values each has a worst case of summing over 9^3 values. If the 3 features had $1,000$ values, this becomes 999^3.

Figure 3 illustrates the "best" case for a feature of $1,000$ values in which every sibling node is a leaf node (or at least a leaf list). If a query that encounters that MCV ADnode involves further features in the missing subtree then it will need to traverse the subtrees (if they exist) of all 999 siblings. This increases the potential for further MCV "collisions", eventually leading to the worst case scenario where every path involves one MCV ADnode for every feature in the query.

Although there are many more queries that don't require summing, higher arity increases the possibility for very slow queries. Additionally, since the more common values have a higher prior probability of being queried (at least for some client algorithms), the most expensive queries are the most likely to occur (or more likely to occur often).

The original justification for excluding MCV ADnodes was that it provided significant space savings in exchange for a minor increase in query time. For binary features, there is no significant increase in query time and each Vary node's subtree is cut at least in half. For a feature with 10 values, the Vary node's subtree is reduced by at least 10% in exchange for summing over at least 9 values. With $10,000$ values, the subtree is reduced by at least 0.01% in exchange for summing over at least 9999 values. The space savings becomes insignificant and the increased query time becomes significant with higher arity features.

There is a further complication when using a dynamic tree. If a query is made that involves at least one MCV then all $n_i - 1$ of its siblings must be expanded. If those nodes are never directly queried then they are simply wasted space. In the previous example of 3 features with $10,000$ values there are 9999^3 unique paths consisting of nearly 10^{12} nodes[1].

[1]There are $9999^3 + 2 \cdot 9999^2 + 2 \cdot 9999 = 999,900,009,999$ total nodes (AD and Vary) needed.

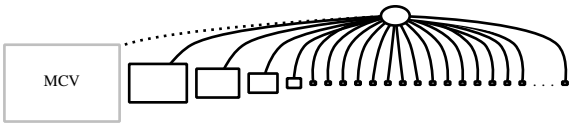

Figure 4: Vary node for a feature with skewed MCV

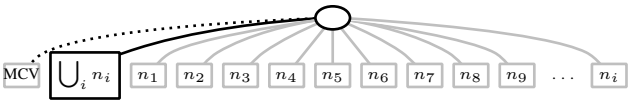

Figure 5: Vary node with a Clump node

Since this is true for any arity above 2^2, if the algorithm does not query all combinations exhaustively then excluding all MCV ADnodes can actually create a larger tree than if they had been included.

Solution 1: Complete Vary Nodes and RatioMCVs

The first solution is to establish a threshold for high arities, above which the MCV ADnode is included. Note that this does not entail making a list of features for which the MCV will always be included. The "arity" used to determine whether or not to include the MCV is the number of children of the parent Vary node. This means that the exclusion of MCVs becomes context sensitive and a given feature may have its MCV included in one part of the tree and excluded in another (just as the actual MCV is context dependent). The threshold, however, is kept constant throughout the tree. A Vary node that has an intact MCV ADnode child will be referred to as a Complete Vary Node.

Although an arity threshold should help provide an improved balance of space and time, there are still some cases where using a simple threshold might incorrectly indicate the need to include the MCV. In particular, if a feature is highly skewed (given the context in the tree) then the MCV may represent a significant portion of a Vary node's subtree. In such a case, the subtree of the MCV may be so large that including it would be too expensive, even if it has a large number of siblings. For instance, including the MCV ADnode shown in Figure 4 based solely on arity while ignoring its disproportionately large size could be worse than leaving it out (as it would have been by default).

Indeed, a strong skew towards the MCV implies smaller counts for its sibling nodes and therefore shorter subtrees. This eliminates the possibility of the worst case scenarios noted above and removes the need to include the MCV ADnode. We therefore introduce the concept of a RatioMCV which is an MCV that has a count above a parameterized threshold ratio of the total of the grandparent ADnode. Then, any MCV which is also a RatioMCV will be removed, regardless of the number of sibling values. The combination of arity threshold and RatioMCV should appropriately handle all possible situations.

Solution 2: Clump nodes

As was noted, excluding MCV ADnodes can interact negatively with dynamic expansion of the tree, counter-intuitively increasing its size. Although the consequences of this interaction are worse for higher arity features, the problem can occur with any number of values above 2. Since the usefulness of excluding the MCV ADnodes is well documented (except as noted here), some other solution is needed.

One feasible solution stems from the fact that this problem does not occur with binary features. A query involving the MCV of a binary feature follows the same path as a similar query switching only the MCV value to its complement. If all of the sibling values of the MCV were "clumped" together as a single "other" value, then any query involving the MCV could be found using this Clump node and performing the appropriate subtraction. This eliminates the need to calculate a sum and does not expand any nodes not explicitly queried (see Figure 5). Furthermore, Clump nodes can be built on demand just like any other node in the tree.

There is a slight catch when using the Clump node, however. A Clump node cannot be used to answer queries involving any of the "clumped" values. Therefore, a query involving such a value will require the expansion of the corresponding node, resulting in some redundancy in the tree. One expensive way to deal with this would be to delete the Clump node and create a new one without the newly expanded value. Using this method, the time needed to query the MCV will slowly increase until the Clump node is completely removed. The approach used in this paper takes the alternative approach of temporarily permitting the redundancy until some percentage of the "other" values are expanded. At that point, the Clump node is deleted.

Empirical Methods

The dataset used for the experiments in this paper was built from the Wall Street Journal (WSJ) corpus from the Penn Treebank. Four additional features were derived from the POS tags: category (simplified POS class), subcategory, tense, and number. Seven additional features were derived solely from the word itself: whether or not the word was "rare" in the dataset, capitalization style, orthographic representation (e.g. contractions), has-hyphen, and has-digit, the last 3 characters of the word, and if it contained one of a short list of derivational suffixes. Several of these features have been used in previous part-of-speech taggers (Toutanova & Manning 2000). Each of these features breaks what would be a single tagging decision into a set of smaller, related decisions, a task more befitting the constraint programming framework. In addition, three more features were added to the new dataset related to morphology. Those features were a base word form (stem) and the corresponding inflectional morphological rule (pattern) as well as whether the base stem was also "rare".

Although many datasets exist with more rows, this dataset is so large in terms of dimensionality and the arity of its fea-

[2]With binary features, it is always redundant to include both nodes so we keep the smaller (non-MCV) one.

Code	Feature	Values	MCV Skew
w	Original Word	42136	5%
b	Word Stem	35698	5%
3	Last 3 Characters	4591	5%
p	Inflectional Suffix	255	94%
c	Part of Speech	45	14%
0	Simplified POS	22	32%
1	POS subcategory	15	38%
s	Derivational Suffix	10	81%
t	Tense	5	86%
C	Capitalization Style	5	86%
n	Number	3	53%
z	Orthographic Style	3	98%
h	Has Hyphen	2	99%
8	Has Digit	2	97%
o	Rare Word	2	95%
a	Rare Stem	2	94%

Table 2: All available features, their arity (number of values) and how skewed is the Most Common Value (MCV)

Feature Sets	Event Space
wc	1.90×10^6
wcCh8	3.79×10^7
wc01tn	9.39×10^9
wc01tns3	4.31×10^{14}
wbcp	1.73×10^{13}
wbcpCh8	3.45×10^{14}
wbcps3	7.92×10^{17}
wbc01tnps3	3.92×10^{21}
wbc01tnzps3Ch8ao	9.41×10^{23}
wc+	6.82×10^{18}
wcCh8+	1.36×10^{20}
wc01tn+	6.75×10^{21}
wc01tns3+	3.10×10^{26}
wbcp+	6.21×10^{25}
wbc01tnps3+	3.07×10^{29}
wbc01tnzps3Ch8ao+	7.37×10^{31}

Table 3: Each code represents a subset of features (+ means that neighboring words were included)

tures that building a static ADtree is not feasible. Table 2 shows the number of unique values in the dataset for each feature. This corresponds to slightly less than 10^{23} possible events (or conjunctive queries). The "code" character for each feature is used as shorthand to represent different groupings of features (see Table 3).

In order to test the proposed modifications, we used a client algorithm that performs part-of-speech tagging on the modified WSJ data. The tagger was built in the Mozart/Oz constraint programming language. In general, the tagger takes as input a sequence of words making up a single sentence. For each word in the sentence, it then outputs a value for each of the features in the (trimmed) dataset. In order to isolate the timing and memory usage of just the ADtree building code, the tagger was used only to generate query logs for each used feature set. All of the experiments reported here were performed using only the query logs to generate the ADtrees.

In addition to the features in Table 2, the tagger can be parameterized to incorporate the context of a given word into its probabilistic model. In some of the following experiments, we used the words directly to the left and right of the word currently being tagged. Although there is generally a strong correlation among neighboring words, most words occur in such varied contexts that many paths in the tree involving a word and its neighbor will be above most reasonable values for the arity threshold. Even considering only the two neighboring words and their respective parts-of-speech (and ignoring their other corresponding features), this increases the event space to close to 2×10^{29} events.

Several different groupings of features were chosen, in part because they represent natural feature groupings for the tagger client application but also as a mechanism for distinguishing how the proposed modifications interact with features of different arities. Table 3 lists nine different feature sets by a shorthand "name". The name corresponds to one character for each feature in the set using the single character "codes" in Table 2. Seven of the nine feature sets were also combined with the word and POS tag for both the left and right words (adding four extra features to each group). Those feature sets are distinguished in the results using a '+' at the end of the name.

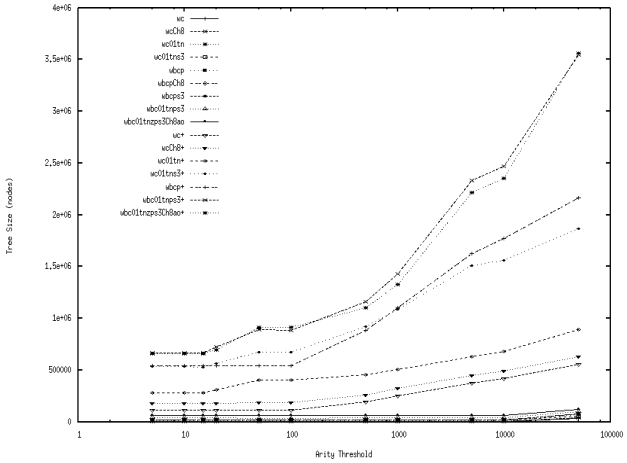

Figure 6: Results of varying the arity threshold from 5 to 50000 on tree size (number of nodes) for each of the 16 feature sets. Notice the x axis uses a logarithmic scale.

The first set of experiments was designed to determine how the tree size varied with respect to the arity threshold. Eleven threshold values were chosen between 5 and 50000. Figure 6 shows the tree size (number of nodes) for each feature set as the arity threshold is varied. Additionally, it is important to note that since the highest arity feature has only slightly more than 40,000 values, an arity threshold of 50,000 is actually equivalent to an unmodified ADtree. One detail obscured by the scale of the graph is that several of the feature sets exhibit a slight upturn at the low end of ar-

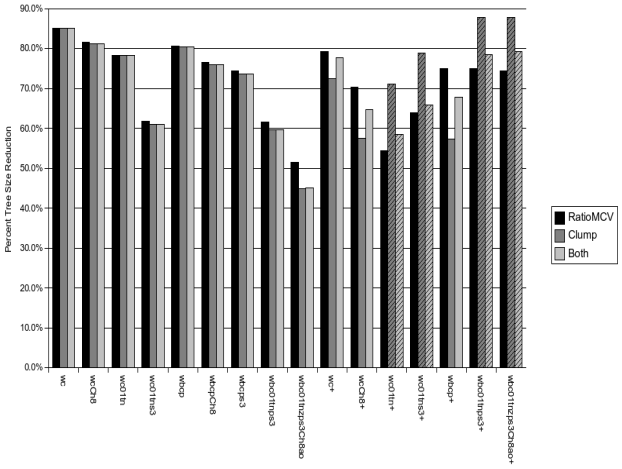

Figure 7: Percent tree size reduction relative to the baseline (higher is better). Bars with hash marks did not finish constructing the tree due to memory complications.

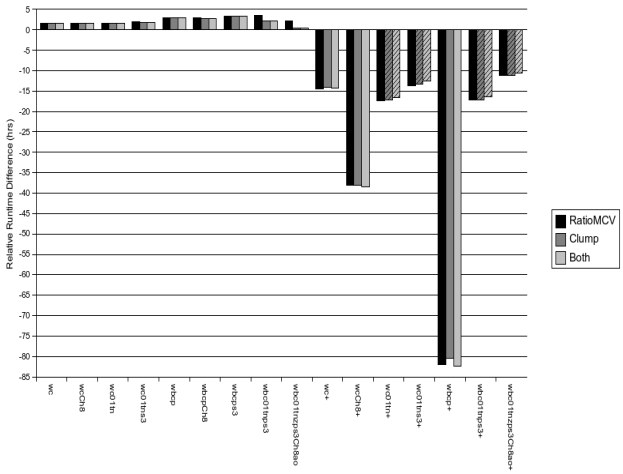

Figure 8: Relative difference in runtime (in hours) from the baseline (lower is better). Bars with hash marks did not finish constructing the tree due to memory complications.

ity threshold. In fact, the lowest point is usually between 20 - 50. This is of course a function of this dataset and the correlations among features and could vary greatly from one dataset to another. Additionally, the optimal tree size for a feature set does not always correspond to the lowest average query time. The remaining experiments use a threshold of 100 (when applicable) since most of the feature sets are still close to their optimal size at that threshold. It is however readily apparent from the graph that almost any choice of threshold will generally be an improvement over the baseline.

Once a reasonable value for the arity threshold was established, both the MCV ratio threshold and the Clump node deletion threshold were investigated. Each of these two thresholds correspond to a ratio that can vary between 50-99%. The MCV ratio threshold establishes how skewed the MCV of a high arity feature has to be before it will be included despite the feature's arity. The Clump node deletion threshold establishes how much (partial) redundancy is permitted before the Clump node is removed. Interestingly, it was not possible to establish a correlation between the exact value for either of these thresholds and their effectiveness. Both the size of the tree and average query time were found to be very robust with respect to these two ratio thresholds.

For this reason, it was decided to arbitrarily use a ratio threshold of 50%, meaning that the MCV is at least as large as the sum of all the other possible values. Similarly, the Clump node deletion threshold was also set at 50%. Three variations of the ADtree were built: using just the Complete Vary Node + RatioMCV modification, using just the Clump node modification, and using both modifications together. Then ADtrees were built for each combination of parameters (including the feature set) and the tree size (node count), elapsed runtime and cumulative memory usage were all measured. Elapsed runtime is a fairly reasonable estimate for average query time because each experiment represents only the time required to build the tree using the query logs recorded from the POS tagger. Cumulative memory usage is primarily dominated by the amount of memory used in temporary data structures needed to perform calculations while dynamically generating the tree. The results of these experiments are summarized in Figures 7-9.

Discussion

In general, it is clear that both modifications achieve the desired results of reducing tree size without substantially increasing build/query time. In particular, the smaller feature sets (those not including neighboring words) obtained on average better reduction in tree size but take a few hours longer to construct the tree and use more cumulative memory. The feature sets that included the neighboring words on the other hand generally performed faster while still obtaining reasonable tree size reductions. Although there is some variation among feature sets, it is not unreasonable to conclude that either modification successfully mitigates the unintended consequences of using MCVs in an ADtree with high arity features.

Even though each modification reduces the tree size, the use of Complete Vary Nodes (and RatioMCVs) appears to achieve equal or better results in terms of tree size but takes on average slightly longer to do so. This is most evident in Figures 10 and 11 as the performance corresponding to Complete Vary Node and RatioMCV with respect to the number of sentences completed slowly separates from that of the Clump node. However, there is a potential advantage to be gained by this difference in tree size versus elapsed time. Even though the original motivation for using both of these modifications was to improve performance on a dynamically generated ADtree, the RatioMCV modification can be applied just as easily to a statically built ADtree, resulting in the same tree size reduction. In that case, the cost of building the tree is a onetime cost that can be amortized

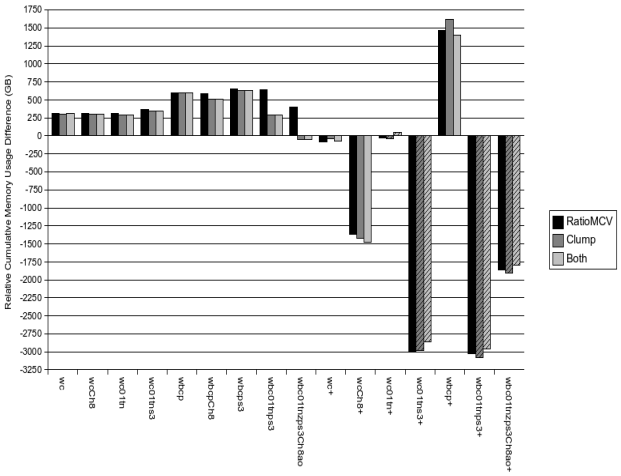

Figure 9: Difference in cumulative memory usage (in GB) from the baseline (lower is better). Bars with hash marks did not finish constructing the tree due to memory complications.

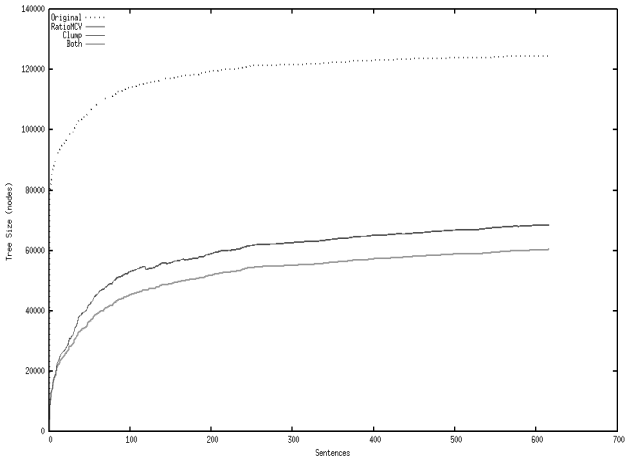

Figure 10: Example of how tree size varies as sentences are tagged using the feature set wbc01tnzps3Ch8ao.

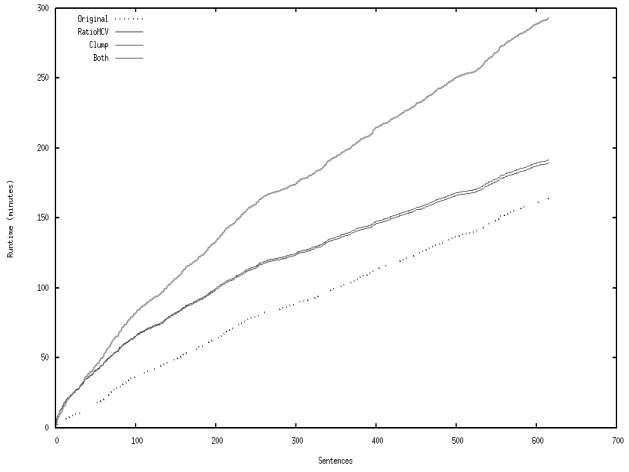

Figure 11: Example of how much runtime elapses as sentences are tagged using the feature set wbc01tnzps3Ch8ao.

over all future uses of that tree. Furthermore, generating a static tree can be optimized in ways that can't be applied to a dynamic tree, potentially eliminating the extra time cost of using Complete Vary Nodes with RatioMCVs. Because the Clump node always represents 100% redundant information in a static tree, it is of no use for static trees.

It is also apparent from the results that combining both modifications together in the same ADtree only rarely gives any advantage over using whichever modification proved most effective for the given feature set. However, since it is not necessarily obvious before hand which modification will provide the best performance, using both seems to be an effective way to "hedge your bet" as to which modification will perform best (assuming that a dynamic tree will be used).

From the cumulative memory usage metric, it is clear that the implementation of ADtrees used here is not optimal in memory utilization, particularly with respect to the temporary data structures which are built and then discarded very frequently in the course of dynamically building the tree. It may be possible to achieve nearly universal build/query time improvements (rather than the mixed results achieved here) by minimizing the amount of temporary memory used and therefore the amount of time spent doing garbage collection.

References

Anderson, B., and Moore, A. 1998. Adtrees for fast counting and for fast learning of association rules. In *Proceedings of the Fourth International Conference on Knowledge Discovery in Data Mining*, 134–138. AAAI Press.

Komarek, P., and Moore, A. 2000. A dynamic adaptation of ad-trees for efficient machine learning on large data sets. In *Proceedings of the International Conference on Machine Learning (ICML)*, 495–502.

Moore, A., and Lee, M. S. 1998. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research* 8:67–91.

Moraleda, J., and Miller, T. 2002. Ad+tree: A compact adaptation of dynamic ad-trees for efficient machine learning on large data sets. In *Proceedings of the 4th International Conference on Intelligent Data Engineering and Automated Learning*, 313–320.

Roure, J., and Moore, A. 2006. Sequential update of adtrees. In *Proceedings of the 23rd International Conference on Machine Learning*, 769–776.

Toutanova, K., and Manning, C. 2000. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of Joint Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, 63–70.