2008-07-10

Adapting ADTrees for Improved Performance on Large Datasets with High Arity Features

Robert D. Van Dam
Brigham Young University - Provo

ADAPTING ADTREES FOR IMPROVED PERFORMANCE ON LARGE DATASETS WITH HIGH ARITY FEATURES

by

Robert Van Dam

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Robert Van Dam

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Dan Ventura, Chair
Date	Irene Langkilde-Geary
Date	Scott Woodfield

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Robert Van Dam in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Dan Ventura Chair, Graduate Committee

Accepted for the Department

Date	Kent Seamons Graduate Coordinator

Accepted for the College

Date	Thomas W. Sederberg Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

ADAPTING ADTREES FOR IMPROVED PERFORMANCE ON LARGE DATASETS WITH HIGH ARITY FEATURES

Robert Van Dam
Department of Computer Science
Master of Science

The ADtree, a data structure useful for caching sufficient statistics, has been successfully adapted to grow lazily when memory is limited and to update sequentially with an incrementally updated dataset. However, even these modified forms of the ADtree still exhibit inefficiencies in terms of both space usage and query time, particularly on datasets with very high dimensionality and with high arity features. We propose five modifications to the ADtree, each of which can be used to improve size and query time under specific types of datasets and features. These modifications also provide an increased ability to precisely control how an ADtree is built and to tune its size given external memory or speed requirements.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As machine learning matures and is applied to new problems, the need arises to model datasets that are substantially larger than those that have traditionally been attempted. The higher dimensionality of many of these datasets exceeds the capacity of current techniques for storing model parameters, but higher dimensionality is correlated with more relevant, meaningful, and interesting problems. Unfortunately, most existing models suffer from practical (if not theoretical) problems of tractability with high dimensional data. This is particularly true of statistical models which can make use of a full joint probability distribution, because these generative models require the ability to model the probability of any potential event or arbitrary tuple of feature/value combinations.

A general model that makes no assumptions of independence among its features implies an exponential number of events that must be counted in order to obtain the statistics necessary for the model. Not only are there an exponential number of possible events when considering all of the features, there are an exponential number of subsets of the features representing events where the values of the missing features are unimportant. In general, if there are m features in a dataset and each feature has n_i possible values plus an option to ignore that feature's value, then there are

$$\Pi_{i=1}^{m}(n_i + 1) \tag{1.1}$$

possible events. This gets overwhelmingly large even for what would otherwise be considered moderately sized datasets. For example, a set with twenty binary features (such as the SPECT dataset from the UCI repository) would have $3^{20} \approx 3.4\,billion$ unique events and a set with ten features with ten values each would have $11^{10} \approx 25.9\,billion$ unique events.

To simplify the process of learning and/or storing a model, many algorithms make assumptions about the data such as independence or conditional independence between certain features. For example, in an mth order Hidden Markov Model (HMM), the current state depends on only the previous m states, assuming independence from all others. HMMs are used repeatedly in language modeling, language identification, speech recognition and other problems that are inherently sequential. As another example, the Naive Bayes model makes the assumption that all variables are conditionally independent of each other given the class variable. Naive Bayes has often proven useful for classifying objects into distinct categories. Models that make use of these and other assumptions are generally simpler (and therefore more computationally feasible) than models that do not, but since the assumptions may not actually hold, they can harm the accuracy of the model.

Typically, machine learning tasks focus on predicting a single output variable. Models are trained on a specific set of input variables, and it is assumed that any novel data to which the model is later applied will provide values for all input variables (although those inputs may indicate an 'unknown' value). As a result, many models can be simplified by treating the output variable differently from the input variables. However, multitask learning [4] permits the more general possibility of multiple inputs and outputs in a single model. Although these learning techniques will not be addressed here directly, we are interested in the need for a general purpose mechanism in which to store probabilistic models that will work for both single and multiple task prediction and will avoid the assumption that there is only one output.

Whether or not these or other assumptions are made, probabilistic machine learning techniques require fast access to the counts of events. Usually such statistics are stored in a straightforward data structure like an array. However, as the size and/or dimensionality of the event space increases, this becomes impossible to do in RAM given common memory limitations. Even an extremely efficient array that needs only 4 bytes per count (with no additional overhead) could only store 1 billion events in the 4 gigabyte limit of a 32-bit machine (ignoring other OS-specific limitations). As described earlier, even moderately sized problems can greatly exceed this limit. The SPECT dataset with 20 binary features would need 13 gigabytes, the dataset with 10 features of 10 values each would require 96 gigabytes and the enhanced WSJ corpus that will be introduced later would require 56 yottabytes (trillion terabytes). In many cases, even the additional space afforded by using disk storage may not provide sufficient space and is often too slow for many applications.

An alternative would be to lazily collect count statistics on demand, since often only a small fraction of events is needed. However, lazy counting generally implies iterating over the entire dataset every time a count is required. Given datasets with a large number of rows, this is prohibitively slow.

Several solutions exist for storing data, particularly real-valued data, in ways that allow efficient access to the subsets of the data that are of most interest to specific algorithms. These solutions are indexing data structures. Structures of this nature include kd-trees and BSP-trees (described later), both of which group together real-valued points which are closest to each other in a dataset. However, less work has been done on lazily caching the results of counting conjunctive discrete-valued events in a dataset. Since the dimensionality of the event space grows exponentially with the number of features, the previously mentioned memory problems will not go away with advances in RAM memory size or computer speed. To use large datasets, it will be necessary to devise a more effective way to index the data and count the matching

rows. An effective solution will make the best possible time and space tradeoff.

Chapters 2 and 3 will discuss some previous approaches to the problem of efficiently storing datasets and/or counts of events built from those datasets. In particular, Chapter 3 will introduce the ADtree which will serve as the primary foundation for the work presented here. Chapter 4 details the dataset and client application which serve both as motivation (by identifying some existing problems with the ADtree) and as the test-bed for further improvements to the ADtree. Chapter 5 will include explanations for five modifications to the original ADtree which can be used, particularly (but not exclusively) in the context of high arity features. This will be followed in Chapter 6 with a detailed examination of each of these modifications when used by the application introduced in Chapter 4. Much of Chapters 3-4 as well as Sections 5.1 and 6.1 have previously been published in a more condensed form. [5]

Chapter 2

Related Work

One widely applicable structure for representing high-dimensional datasets is a Binary Space Partitioning tree or BSP tree. Each level of a BSP trees subdivides a space into two smaller subspaces until the leaves of the tree all meet some predefined criteria (which can vary depending on the purpose of the tree). The resulting tree tends to group subparts of a dataset so that points which are close together (and therefore presumably similar) are in the same space defined at a leaf node. If that group of points are meaningful to a machine learning algorithm, the BSP tree significantly reduces the number of redundant passes through the entire dataset [7] [8].

The kd-tree (short for k-dimensional tree) is a type of BSP tree which restricts the partitions to splitting the space only at data points and only along orthogonal hyperplanes. This partitioning criterion leads the kd-tree to contain only k-dimensional hypercubes at the leaf level. Neither of these structures directly stores the counts of events in the dataset, they store only those points which may represent all instances of a certain subset of events (depending on the dividing criteria). This type of structure can be very useful for some algorithms, particularly ones which use a data point's nearest neighbors for calculations, but it does not generalize to permit fast counting of arbitrary events [2] [8].

Andrew Moore and Mary Soon Lee [12] proposed a caching data structure which overcomes many of the time and space problems noted previously. Their structure, the ADtree, is a generalization of the kd-tree which provides access to all events

while still improving memory usage. Additionally, the ADtree can benefit from both independence and correlations among its features in order to create a structure which is orders of magnitude smaller than its theoretical upper bound and capable of meeting memory requirements for higher dimensional datasets.

The ADtree provides a substantially improved mechanism over naive methods for algorithms that require frequent counting over a dataset. Unfortunately, the problem of counting is fundamentally an exponential problem and many datasets are still much too large even for an ADtree. The structure that Moore and Lee presented was designed to be built in its entirety and then used by the algorithm, after which it could either be stored for future use or simply discarded. However, Komarek and Moore [9] later noted that what they termed static ADtrees (those which are completely built before being used) often waste space (and therefore waste time in generation) on parts of the tree that will not be used by the algorithm for which the tree was built. If these parts of the tree were easily identifiable *a priori*, then a simple modification could create the tree without including the unnecessary subtrees.

However, the unnecessary parts of the tree can often be the result of a complicated interaction between the specific machine learning algorithm and dataset (that is, changing either the algorithm or the dataset would change which parts of the tree are extraneous). Komarek and Moore's solution was to create a dynamic adaptation of the ADtree which functions as a lazy caching structure that grows the tree only when presented with specific requests (queries) which are not already cached. Their adaptation again makes use of several clever techniques to avoid repeatedly iterating over the dataset as might otherwise seem necessary.

Static ADtrees provide a memory-efficient mechanism for utilizing large datasets with high dimensionality in machine learning algorithms. Dynamic ADtrees extend those capabilities even further. However, real-world sized datasets can still be difficult or impossible to manage. For instance, the extended WSJ corpus mentioned previ-

ously generates an ADtree that does not fit into 4 Gigabytes of memory. Furthermore, although dynamic ADtrees can be applied to high arity features, they give extremely poor average access times and very minimal space savings on these attributes since one of the space-time tradeoffs used to reduce the size of the tree is no longer applicable for high arity features. Additionally, the trade-offs between the amortized runtime and re-usability of static ADtrees and the space-savings and flexibility of dynamic ADtrees remain relatively unexplored.

Chapter 3

Description of ADtrees

The full ADtree (see Figure 3.1) 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 $Q \wedge a_i = v_j$.

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 afford reasonable 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 [12] using Eq. 3.1.

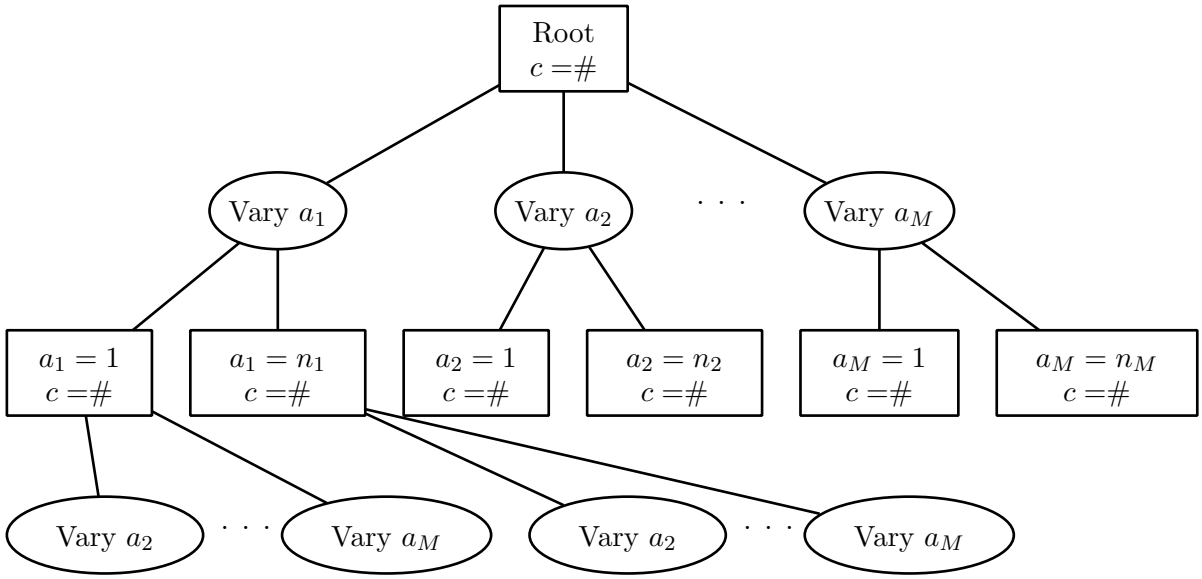

Figure 3.1: Incomplete top levels of a generic ADtree showing the alternating pattern of ADnodes and Vary nodes. In a real ADtree, the ADnode for the MCV and any nodes with a count of zero would be removed

$$count(Q \wedge a_i = v_k) = count(Q) - \sum_{j \neq k} count(Q \wedge a_i = v_j) \tag{3.1}$$

If k is chosen such that v_k is the Most Common Value (MCV) that is, it has the largest count 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 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 (due to calculating MCV counts) is reasonable in comparison to the expected space savings.

The tree shown in Figure 3.2 illustrates what an ADtree generated according to the dataset in Table 3.1 would look like. The nodes shown in gray are those nodes which would have existed in a full tree but which 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.

Although Eq. 3.1 appears fairly simple in form, in practice many layers of MCVs can create a rather complicated sequence of queries. For example, the following set of expressions walks through the process of refining the query $count(a_1 = 1, a_2 = 2, a_3 = 0)$. Each refinement either "unrolls" a summation, is caused by a "collision" with an MCV (thereby generating a summation) or substitutes a value when the query is explicitly stored in the tree. In order to save space, the queries are condensed such that features are made implicit based on order. As a result, $count(a_1 = 1, a_2 = 2, a_3 = 0)$ becomes $count(1, 2, 0)$ and $count(a_3 = 0)$ becomes $count(*, *, 0)$. $count(*, *, *)$ is therefore the empty query which is equal to the total number of rows (63).

$$
\begin{aligned}
& count(1,2,0) \\
& count(*,2,0) - \sum_{v_1 \neq 1} count(v_1,2,0) \\
& count(*,2,0) - count(0,2,0) \\
& \left[count(*,*,0) - \sum_{v_2 \neq 2} count(*,v_2,0) \right] - count(0,2,0) \\
& [count(*,*,0) - count(*,0,0) - count(*,1,0)] - count(0,2,0) \\
& \left[\left[count(*,*,*) - \sum_{v_3 \neq 0} count(*,*,v_3) \right] - count(*,0,0) - count(*,1,0) \right] - count(0,2,0) \\
& [[count(*,*,*) - count(*,*,1)] - count(*,0,0) - count(*,1,0)] - count(0,2,0) \\
& [[63 - 30] - count(*,0,0) - count(*,1,0)] - count(0,2,0) \\
& [33 - count(*,0,0) - count(*,1,0)] - count(0,2,0) \\
& [33 - 1 - count(*,1,0)] - count(0,2,0) \\
& [32 - count(*,1,0)] - count(0,2,0) \\
& [32 - 0] - count(0,2,0) \\
& 32 - count(0,2,0)
\end{aligned}
$$

$$32 - \left[count(0, *, 0) - \sum_{v_2 \neq 2} count(0, v_2, 0) \right]$$

$$32 - [count(0, *, 0) - count(0, 0, 0)]$$

$$32 - [1 - count(0, 0, 0)]$$

$$32 - [1 - 1]$$

$$32$$

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.

$a_1 a_2 a_3$	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 3.1: Sample dataset for the ADtree in Figure 3.2 from [13]. This is a condensed representation of a 63 row dataset.

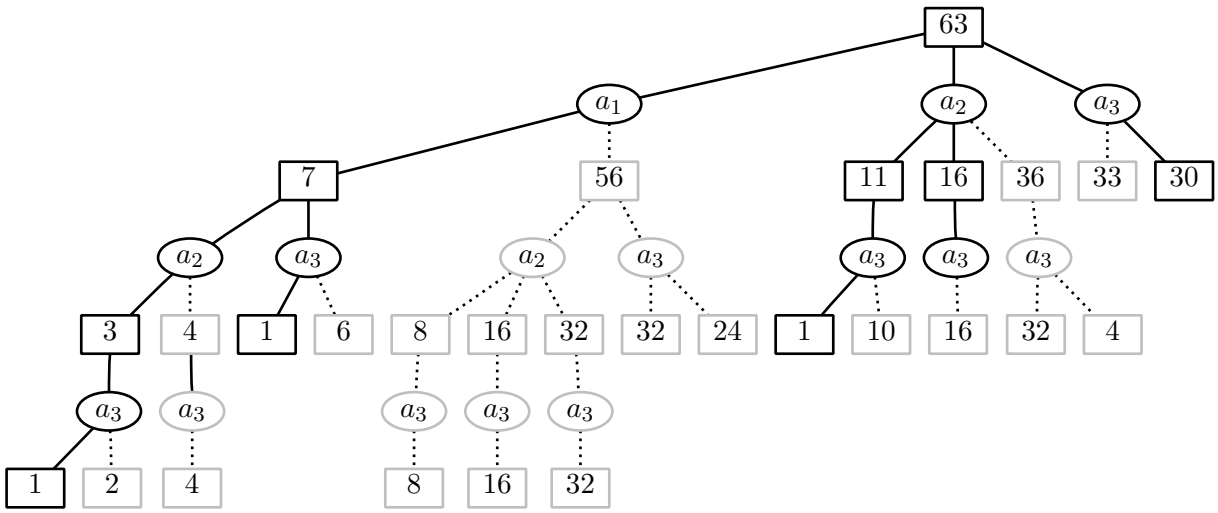

Figure 3.2: An example of a simple ADtree generated from the data in Table 3.1. MCV nodes and their corresponding subtrees are marked with gray boxes and dotted edges. Nodes with a count of 0 have been left out entirely. For instance, there is no node for $a_1 = 0$ and $a_2 = 1$ since this combination never occurs in the dataset.

Chapter 4

WSJ Dataset and Constraint-based Part-of-Speech Tagger

In order to test the proposed modifications, we use a client algorithm that performs part-of-speech tagging on a modified version of the Wall Street Journal (WSJ) corpus. The tagger is built in the Mozart/Oz constraint programming language. In general, the tagger takes as input a sequence of words making up a single sentence, and for each word in the sentence, it then outputs a value for each of the features in the dataset. The client algorithm uses a number of constraint relationships between the features such that determining the value of one feature for a particular word can automatically narrow the set of possible values for other features. Furthermore, the algorithm does not have a fixed order in which it determines the values of each feature. Instead, it searches for the overall most probable assignment of tags/feature-values to words based on a best-first methodology, and conditions later feature assignments on earlier ones. Thus, typically, punctuation and words like "th" and "to" are among the first to be tagged, and they are then used to help dismbiguate the tags of the remaining words. Sometimes a tag is used to help determine the morpheme (eg. "housing" in "September housing starts due Wednesday":tag=noun–¿morph=none) and sometimes vice-versa (eg. "demonstrating":morph=ing–¿tag=gerund verb). As a result, each sentence can result in a completely different sequence of queries (although this sequence is still deterministic). Additionally, the client algorithm attempts to calculate independence on the fly, meaning as a sentence is tagged, each subsequent query is not simply a refinement of the previous query. This high variability of queries is an

ideal platform for testing the ADtree because the algorithm in general requires fast access to all possible queries.

However, the client algorithm is not the central focus of the work presented here. Rather, it is a tool used to demonstrate the effectiveness of the proposed modifications to the ADtree. As long as the modifications to the ADtree do not compromise the validity of the query responses (e.g. by approximating some queries in order to save space) the details of the client algorithm are less important than the ADtree. 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 feature set. All of the experiments reported here were performed using only the query logs to generate the ADtrees. Using these query logs means that none of the proposed modifications can tag more sentences than were tagged by the initial run. Although this can be seen as a negative consequence, it does make it easier to compare the various modifications because the results don't have to be normalized by the number of sentences completed. As a result, except where noted, the number of sentences tagged and all other aspects of the tagger will be ignored in all subsequent chapters.

The dataset used for the experiments in this paper was built from the Wall Street Journal (WSJ) corpus from the Penn Treebank. The original dataset included each word along with its Part-of-Speech (POS) tag as well as the syntactic structure of each sentence. The dataset described here maintains the word and POS tag as features but ignores the syntactic sentence structure. 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 [16]. Each of these features breaks what

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 4.1: All available features, their arity (number of values) and how skewed is the Most Common Value (MCV). 'Original Word' ('w') and 'Part of Speech' ('c') come from the original WSJ corpus, the remaining features are derived from one or both of these (except for the two 'Rare' features which are derived from the corpus as a whole). "MCV skew" measures the independent skew of each feature across the entire dataset. When calculated given a conditional context of other feature/value assignments, this measure can change dramatically (as can the MCV itself)

would be a single tagging decision into a set of smaller, related decisions. In addition, three more features were added to the new dataset related to morphology. Those features were a base word form and the corresponding inflectional morphological rule (pattern) as well as whether the base stem was also "rare".

This enhanced WSJ dataset has over 1 million rows (word tokens). Although many datasets exist with more rows, this dataset is so large in terms of dimensionality and the arity of its features that building a static ADtree is not feasible. Table 4.1 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 4.2).

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 4.2: Each code represents a subset of features (+ means that neighboring words were included). There is not a '+' feature set corresponding to each plain feature set because the client algorithm could not complete enough sentences to create usable query logs.

In addition to the features in Table 4.1, 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 can still be considered "high arity". 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 4.2 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 4.1. 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. The other two feature sets were not used, when combined with neighboring words, because the client algorithm could not finish tagging enough sentences to provide usable query logs.

Chapter 5

Modifications

This chapter will introduce five different modifications to the ADtree. These modifications are generally more situation-specific than the techniques used in the original ADtree. This is not to say that they are not widely applicable outside of the dataset and client application used here. Rather, each modification depends upon slightly varying data and usage characteristics. The situations in which each modification can be applied are detailed both in this chapter and in Chapter 6. It is also worth noting that the ADtree used in most of these cases is a dynamic version of the tree. However, many of these improvements can still be used in a static tree (although that sometimes implies a change in the expected space-time tradeoff). The applicability of each modification to a static ADtree will be discussed as part of each of the following sections.

Section 5.1 presents together the first two modifications, both of which provide improvements intended for datasets with high arity features (partially applicable to static trees). Section 5.2 details a modification that leverages some of the benefits of the static ADtree that are lost when using a dynamic tree. Section 5.3 attempts to ascertain the effects of different methods for ordering the dataset features within the tree (static and dynamic). Section 5.4 discusses pruning nodes from the tree based on usage patterns if/when the tree grows too large to be held in memory (dynamic trees only).

5.1 Improved Space-Time Tradeoffs for High Arity Features

It is noteworthy that the majority of previously published results using ADtrees 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 build/query times.

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 which specifies values for q features, the worst case scenario could require summing over $\prod_{i=1}^{q}(n_i - 1) + 1$ 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 $10,000$ values, this becomes 9999^3.

Figure 5.1 illustrates the "best" case for a feature of $10,000$ 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 9999 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

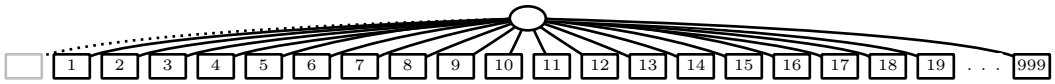

Figure 5.1: Iconic representation of a vary node for a high arity feature ignoring any further subtrees. This effectively illustrates the "best case" scenario with respect to the base problem of summing over so many counts in order to retrieve a single number. A real-world situation could be far worse if the subqueries had to descend subtrees and potentially encounter more missing MCV nodes. (Although node size is not represented graphically, the only assumption is that the MCV is not highly skewed).

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 increasingly insignificant and conversely the query time becomes increasingly 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]). Although not as dramatic for lower arity features, this is true for any arity above two[2]. Therefore, if the algorithm does not query all combinations

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

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

exhaustively then excluding all MCV ADnodes can actually create a larger tree than if they had been included.

5.1.1 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 due to a large MCV (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 5.2 based solely on arity while ignoring its disproportionately large count 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 mitigates 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,

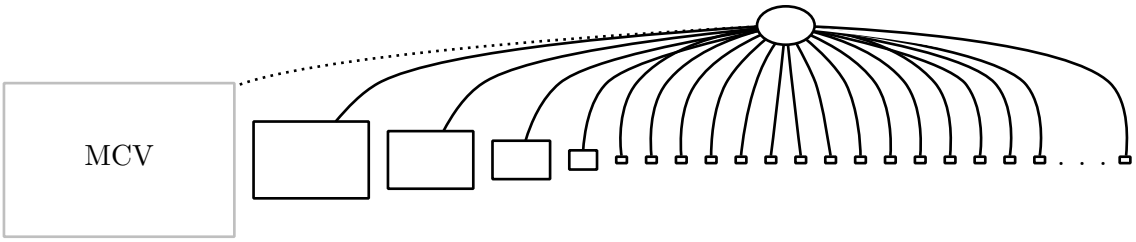

Figure 5.2: Vary node for a feature with skewed MCV (high proportional count). Relative node size represents the relative counts at each node (as well as the *a priori* potential size of the corresponding subtrees). Although it is still necessary to sum over many nodes to recover some queries, the MCV subtree is so large and the potential for further MCV collisions in the smaller subtrees so small that the trade-off is more reasonable (contrast Figure 5.1).

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 since the thresholds can be adjusted to accomodate different datasets and feature arities.

5.1.2 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 5.3). Furthermore,

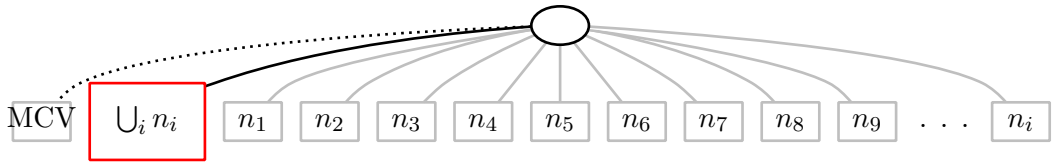

Figure 5.3: Here the issue of skew is ignored and so the nodes are represented as uniform in size. However, the Clump node represents both the sum of the counts of the other nodes as well as the union of the rows (represented here) for that node's partition of the datasest.

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.

5.2 Static and Dynamic ADtree Hybridization

Static trees provide excellent amortized conjunctive counting query retrieval time, particularly if they are reused multiple times. However, if a dynamic tree is required due to memory limitations, the tree must be regenerated on each run of the client algorithm.

Even if the series of queries produced by different runs of an application are not identical, it is likely that the ADtrees from each run share some structure. One reasonable solution is to hybridize the benefits of the static and dynamic trees by generating some of the shared portions of the tree in advance. This can reduce the

time needed to dynamically generate the tree without wasting space on unneeded portions of the tree. In general, the first and second levels of the tree are the most likely to be needed by an arbitrary algorithm. These levels represent all possible single and paired feature queries and constitute partial computation of higher-order queries. So it would generally be beneficial to statically precompute these portions of the tree. Additionally, this hybridization could be parameterized to statically generate only a subset of features or values.

In many respects, this hybridization technique is a way to memoize a query-based analog to "currying", a function generation technique used in some higher-order functional languages. The simplest form of currying takes an n-argument function and a value for one of those arguments and returns a new, $(n-1)$-argument function in which all occurrences of the first argument have been replaced with the given value. This can be generalized to replacing k of the original function's arguments, where $k < n$, by repeatedly currying the result of the previous currying. The hybrid ADtree, essentially "curries" the query function for all values of each feature independently. The second level can then be generated by taking the results of the first level and currying each query/node with all possible values of a second, distinct feature[3]. The result is an efficiently stored, memoized table of all (or a parameterized subset of) 2-feature full and partial queries.

5.3 Feature Ordering Strategies

The original ADtrees maintain a predetermined fixed order on features that determines the path that a query should follow through the tree. Any conjunctive query is first sorted according to this ordering and then the tree is traversed in that same order. This fixed order is generally a naive ordering such as the original order from the

[3]Fortunately, the actual 2-level tree generation is more efficient than the currying analogy implies since all ADnodes (values) of a particular feature can be generated together in a single step.

dataset or even a simple alphabetical ordering (assuming descriptive feature names exist). A more tailored ordering (to be used with either static or dynamic ADtrees) would determine the relative ordering of features that minimizes the overall size of the tree. Maximum benefit would come from a context-specific ordering because of the increased flexibility to use different relative orderings of features that best minimize the tree given the context.

A context specific ordering requires the use of a heuristic to determine, at each level of the tree, which feature is next along a particular path. A well-defined heuristic is necessary that can be used both for generating the tree as well as for determining the path that should be followed for any given query. Unfortunately, it is not known whether there is a generally optimal heuristic or even whether the same heuristic would work equally well for any two distinct datasets or applications. In particular, a metric that selects for strongest correlation may miss an opportunity to exploit independence and vice versa.

Besides the canonical and alphabetical orderings, several simple heuristics can be designed based on some of the characteristics of the ADtree. The two simplest characteristics on which to base an ordering are arity of the features and size (count) of the MCV of each feature. A third, and somewhat less obvious measure on which to base an ordering is the overall entropy of the values of a feature. Entropy in this context can be defined as

$$H(a_i) = -\Sigma_{j=1}^{n_i} p(v_j) \cdot log_2(p(v_j)) \tag{5.1}$$

where the a_i is the i-th feature in the dataset, n_i is the arity of a_i and $v_1 \cdots v_{n_i}$ are the possible values of a_i. The value of $p(v_j)$, the conditional probability that v_j occurs given the current context in the tree, is estimated using the observed counts such that

$$p(v_j) = count(Q \wedge a_i = v_j)/count(Q) \tag{5.2}$$

where Q is the grandparent query (in the case that Q is the empty query, $count(Q)$ is defined to be the total number of rows in the dataset.) In essence, entropy is a measure of the context-specific distribution of values of a_i given Q. A uniform distribution will have maximal entropy, while a single value (and therefore terminating MCV) will have an entropy of 0.

In general, these three heuristics are not uncorrelated. For instance, (relatively) large MCV counts imply smaller sibling counts and therefore lower overall entropy. High arity features automatically have a higher potential maximum entropy (since the entropy of a perfectly uniform distribution is $log_2(n_i)$) and so an ordering based on largest arity will be identical to one based on highest entropy given a set of independently and uniformly distributed features. However, such a dataset is not likely to be considered sufficiently interesting to perform an analysis on, and so it is reasonable to assume that an entropy-based ordering will be more tailored to the specific distributions of a given dataset. That does not, however, imply that such a tailored ordering is inherently "better" in terms of the resulting size of the tree. In fact, it is not readily obvious in which "direction" to use each ordering (that is, will a lower or higher entropy, a larger or smaller MCV result in a smaller tree). The usefulness of each ordering will generally be very data dependent.

Superficially, ordering based strictly on arity essentially trades depth for width. If the largest arity features are ordered first, the levels of the tree after the first (which is always essentially fixed in size) will likely be very wide because so many feature/value combinations will occur in the first few levels. This has the fortunate side-effect that it partitionsthe fixed rows of the dataset quickly and therefore presumably, the tree will not grow very deep as strong correlations lead to terminal MCVs and smaller counts lead to terminal leaf-lists. Furthermore, strong correlations among the high arity features can greatly reduce the width of the tree because fewer combinations of the first few feature/value pairs are encountered. In the reverse ordering,

the top levels of the tree are smaller because there are many fewer value combinations. However, if the reverse ordering is used and there is a strong correlation between the lowest arity features and the highest, there will be many occurrences at the lower levels of features with a single possible value determined by those correlations. As a result, the tree could be much smaller since it started out "narrow" and many paths terminated prematurely.

Similar analysis of the orderings based on MCV counts and entropy reveal that it could be very difficult (likely impossible) to establish a single, universally-optimal ordering. It may not even be possible to determine an optimal ordering for a given dataset unless the correlations among all features is known *a priori* (which implies a prior analysis of the dataset not unlike that accomplished by the process of building the ADtree). As a result, it may be necessary in many cases to try some or all of the orderings empirically. We will provide results of all five orderings (canonical, alphabetical, by MCV count, by feature arity, and by entropy) performed forward and reversed on the different subsets of features of the enhanced WSJ dataset with the expectation that varying degrees of correlation among the features in each set satisfactorily gauges each ordering's effectiveness on different types of datasets.

5.4 Pruning in Limitated Space Scenarios

Komarek and Moore noted that dynamic ADtrees could be extended beyond their implementation to provide both internal and external pruning mechanisms. External pruning could be as simple as providing an interface to permit the client application to specify to the ADtree which subtrees to prune. Internal pruning could be based on the raw internally stored counts[4] or on a priority queue based on frequency of use as suggested by Komarek and Moore. It is not unreasonable to imagine a system

[4]Without additional knowledge of the client algorithm, the most reasonable prior probability that a node will be used is simply the Maximum Likelihood Estimate taken directly from the counts which are readily available.

that utilizes both mechanisms by pruning using default rules while also permitting the client to submit requests to prune specific subtrees.

The fundamental problem of pruning nodes from the ADtree is to simultaneously maximize the freed memory gained from the pruning of a subtree while minimizing the risk of pruning some portion of the tree which will need to be regrown. It is also possible that the low-level implementation of the tree could associate a time cost with pruning (due to extra garbage collection for instance). This formulation of maximizing payoff versus minimizing risk provides some immediate insight into certain classes of subtrees which it would generally be disadvantageous to prune. Leaf nodes/lists provide a very minimal reward for freed space (assuming an implementation using only indices, not duplicated data rows). Nodes at the top level which have high counts would generally provide the largest reward but also imply the biggest risk since they (or a subtree) have a high probability of being needed again later.

On the other hand, if a high arity feature occurs early in the feature ordering then its subtrees will often be larger than subtrees of other features at that level because they have more features on which to split. However, the high arity feature's lower count values have a lower likelihood of being needed again (and less penalty for regeneration). Therefore, any reasonable internal pruning mechanism would generally expect to avoid pruning the most expensive subtrees and would instead focus on infrequently used mid-level subtrees or top-level subtrees of high arity features with lower counts.

The results given in Chapter 6 represent a simple compromise between internal and external pruning. Given the above specifications for what kind of pruning is likely to be generally applicable, implementing the pruning itself is very straightforward. In this case, the pruning algorithm chosen is reasonably general but also noticably biased by the particular dataset being used. The algorithm removes all ADnodes (values) with a count less than a pre-determined threshold (1000 was chosen somewhat arbi-

trarily) in the first level for high arity features. Since the majority of the high arity features for the enhanced WSJ are based on words, their distributions are generally similar to a power law and so the algorithm basically prunes the "long tail". The difficulty in using this or any pruning algorithm really lies in determining an opportune moment in the processing to perform the pruning. If used as a purely internal mechanism, the easiest and most reasonable approach is to prune if and when memory usage exceeds a predetermined (or possibly parameterized) threshold, resulting in essentially an inverted-greedy pruning algorithm (greedy in the sense of using as much memory as possible). However, this is not likely to correspond to a "convenient" time with respect to the client application. The aforementioned compromise used in this case was to provide the client application with the ability to initiate pruning externally (although not with any external parameters as suggested above). Since the tagger application treats sentences as a whole, while a sentence is being tagged, the words of the sentence are repeatedly used in queries. Therefore, the completion of a sentence (or a set of sentences) seems the most natural event to trigger pruning in the tree. Other client algorithms might also have natural boundaries where pruning is least likely to remove needed counts. The results discussed here were obtained by pruning after every 100 sentences were finished tagging. With feature sets for which the tree grows only gradually, this is likely to provide a reasonable tradeoff between the relatively expensive operation of pruning and risk of running out of memory. With feature sets which cause the tree to grow more rapidly, this 100 sentence gap between prunings may not happen frequently enough to avoid known memory limitations.

Chapter 6

Validation and Analysis of Results

This Chapter analyzes the results of running the client tagger application on each of the datasets discussed in Chapter 4. Structurely, this chapter and its sections mirror those of Chapter 5 in order to make it easier to find and compare the motivations with the results. The raw data that was used to generate most of the graphs shown here can be found in Appendix A (except those graphs showing growth with respect to sentences tagged because of the large number of sentences involved).

When considering the datasets in order of increasing number of features, the amount of memory used by the original ADtree increases very quickly. Unfortunately, since Mozart/Oz is still a relatively new programminging language, there are still some bugs in its memory management and garbage collection systems. Although some bugs have been discovered and fixed as a result of this work and the large amount of memory usage it entails, some bugs in memory management still exist that lead to apparently non-deterministic failures[1]. Due to this, there are several instances throughout the reported results where the results for specific datasets are either marked or excluded. Where the results are marked (such as with hash marks), this can be understood to mean that experiment terminated early (usually about 5-20% fewer sentences than the respective baseline). These results are still valid but are considered to have a slightly larger margin of error than the rest. On the other hand, some experiments

[1]Unfortunately, the non-determinism only relates to 'when' certain experiments fail, not 'whether'.

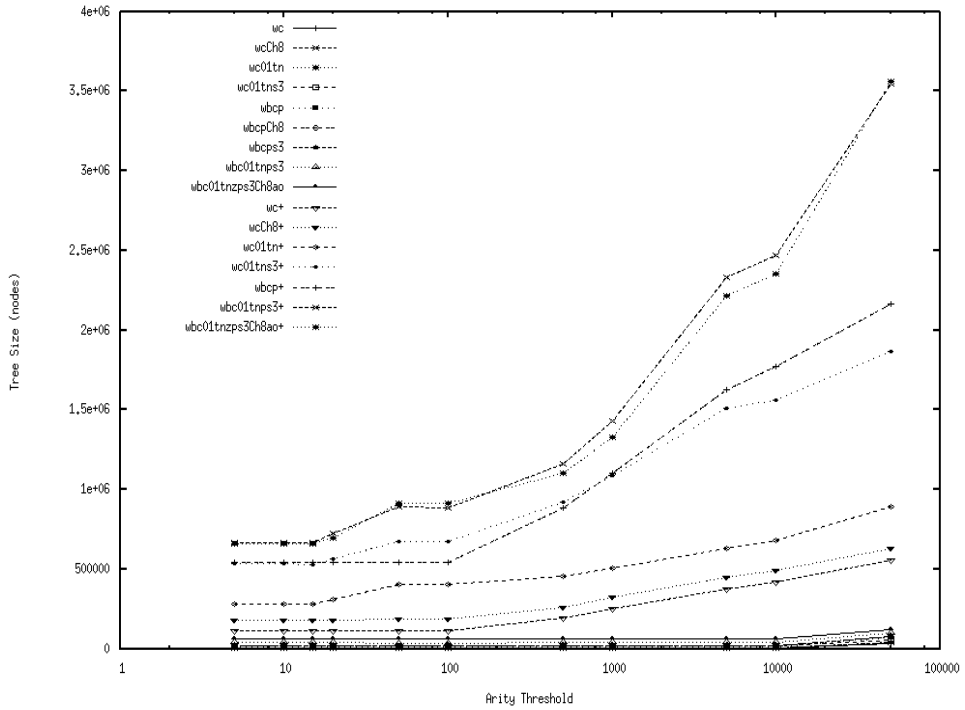

Figure 6.1: Results of varying the arity threshold from 5 to 50000 on tree size (number of nodes) for each of the 16 feature sets. Notice the x-axis uses a logarithmic scale. The individual lines are less important than the overall trend. The large scale of the y-axis disguises the fact that most of the feature sets show a small upward curve at the low end of the x-axis.

completed so few sentences (if any) that the margin of error was considered to be too large to report. In those cases, there may be a gap in a line graph or a missing bar in a bar graph. This does not nececessarily imply that the missing dataset would always be too large or too difficult to use in conjunction with ADtrees and/or the specific modification being analyzed.

6.1 High Arity Features

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 6.1 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 arity threshold. In fact, the lowest point is usually between threshold values of 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

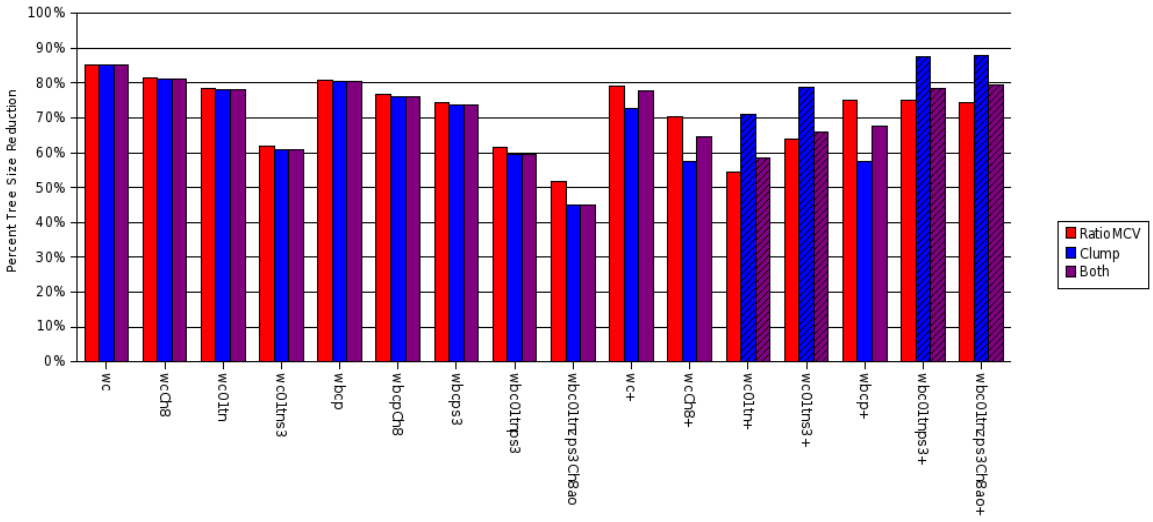

Figure 6.2: Percent tree size reduction relative to the baseline (higher is better). Bars with hash marks did not finish constructing the tree due to memory complications.

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 6.2-6.4.

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.

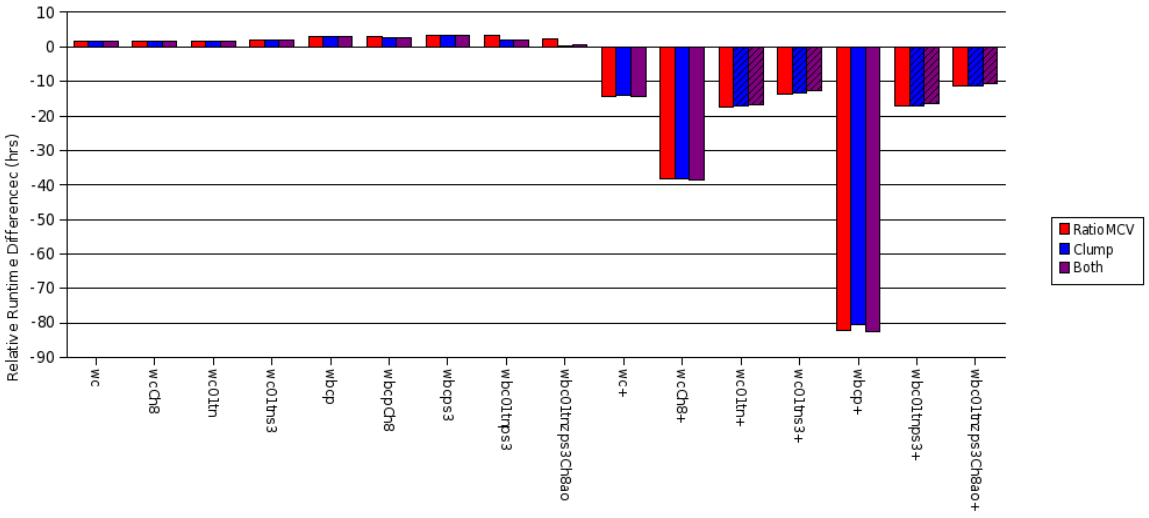

Figure 6.3: Relative difference in runtime (in hours) from the baseline (lower is better). Bars with hash marks did not finish constructing the tree due to memory complications.

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 6.5 and 6.6 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 one time cost that can be amortized 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 less easily justified (though still potentially useful if space is

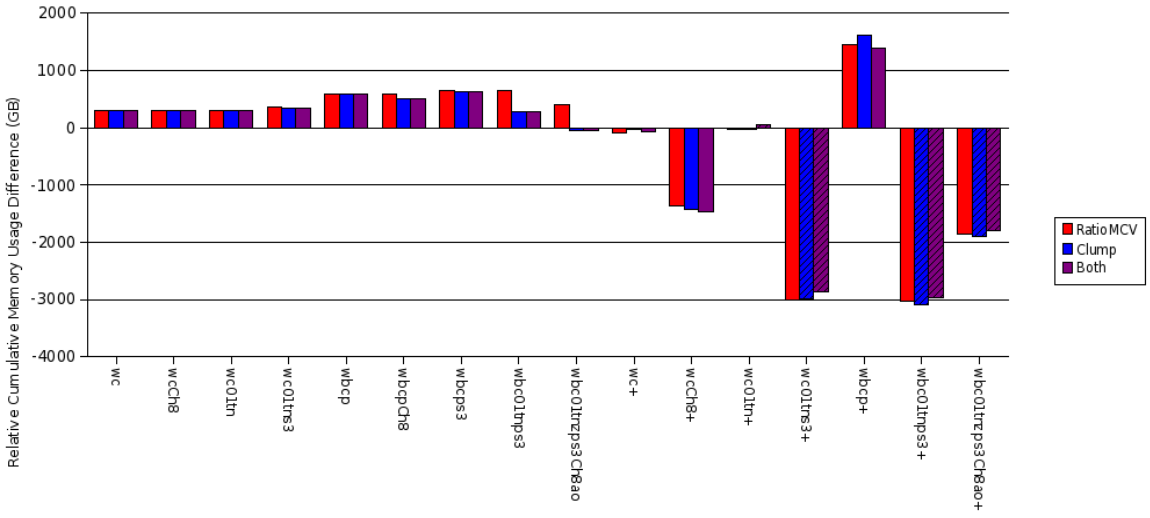

Figure 6.4: Difference in cumulative memory usage (in GB) from the baseline (lower is better). Bars with hash marks did not finish constructing the tree due to memory complications.

available).

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.

7

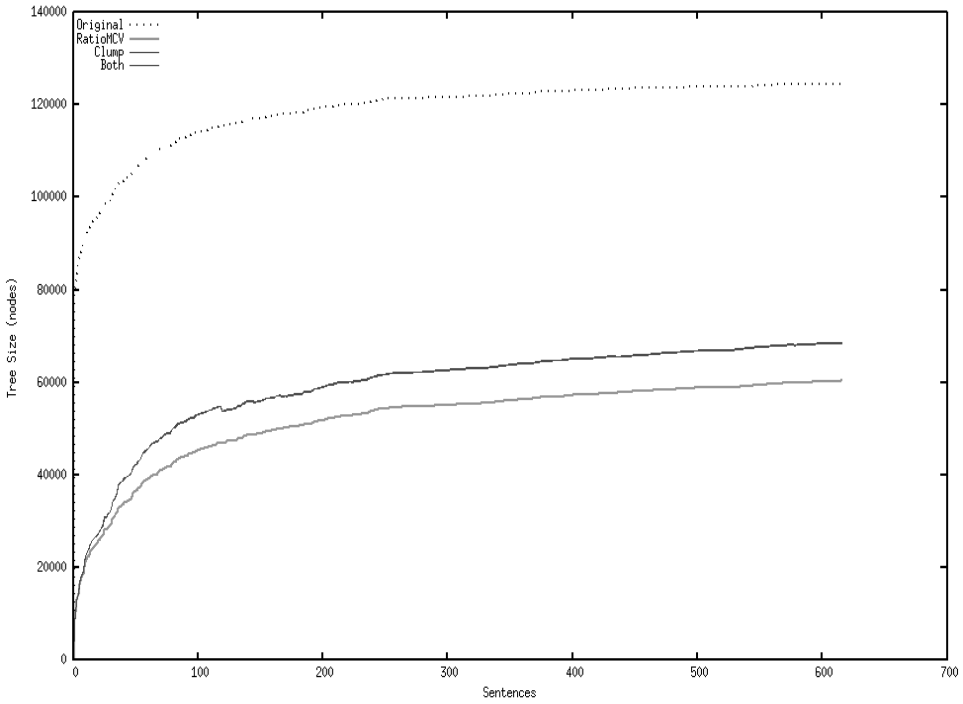

Figure 6.5: Example of how tree size varies as sentences are tagged using the feature set wbc01tnzps3Ch8ao. Tree growth tapers off towards the right as fewer unique words (and correspondingly unique value assignments) are encountered. Although using a clump node does slightly increase tree size, it is still significantly less that the unmodified tree.

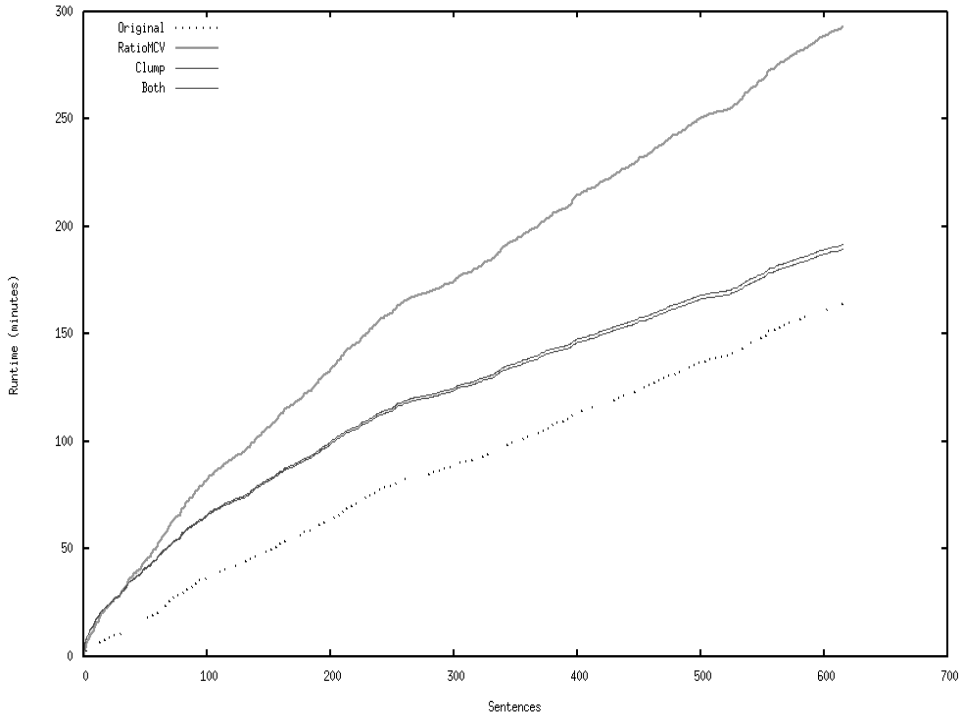

Figure 6.6: Example of how much runtime elapses as sentences are tagged using the feature set wbc01tnzps3Ch8ao. The extra computation time needed for each modification is clear, particularly for the Complete Vary Node and RatioMCV.

6.2 Hybridization

Because of the improvements achieved by the use of Complete Vary Nodes and RatioMCVs, the experiments using the hybrid version of the tree used Complete Vary Nodes and a RatioMCV threshold of 0.5. Correspondingly, Figures 6.7, 6.9, and 6.11 compare the results of creating a hybrid tree in combination with a Complete Vary Node/RatioMCV tree to the results in Figures 6.2-6.4 for just the Complete Vary Node/RatioMCV. As in the previous graphs, the results are relative to a baseline that uses none of these modifications. Additionally, Figures 6.8, 6.10, and 6.12 show just the improvement (or lack thereof) of just the hybrid tree using the Complete Vary Node/RatioMCV results as a new baseline.

In order to understand these results, it is important to understand that a single static tree was built in advance. This tree consists of the first two levels of a tree using all of the available features (including the neighboring features). In other words, it is the first two levels of the 'wbc01tnzps3Ch8ao+' tree. This means that for all other feature sets the tree starts out containing paths that will never be used. The result is that the hybrid, although still greatly outperforming the baseline in terms of tree size, is actually larger than the RatioMCV version of the tree. This difference is most pronounced for the smallest feature subset (wc) and reduces to near zero for larger subsets (particularly those involving the neighboring words).

On the other hand, the runtime of the hybrid trees is unaffected by the excess tree nodes and all but one of the feature sets improved over both the original and the RatioMCV baselines (with the one outlier performing the same as the RatioMCV baseline). Additionally, the cumulative memory usage substantially improves over the RatioMCV baseline in almost every case, in most cases at least canceling the extra memory used by the RatioMCV tree. It is also interesting to add back in the one-time cost of building the static portion of the hybrid tree. Building a static tree is generally more efficient in terms of both time and temporary data structures than

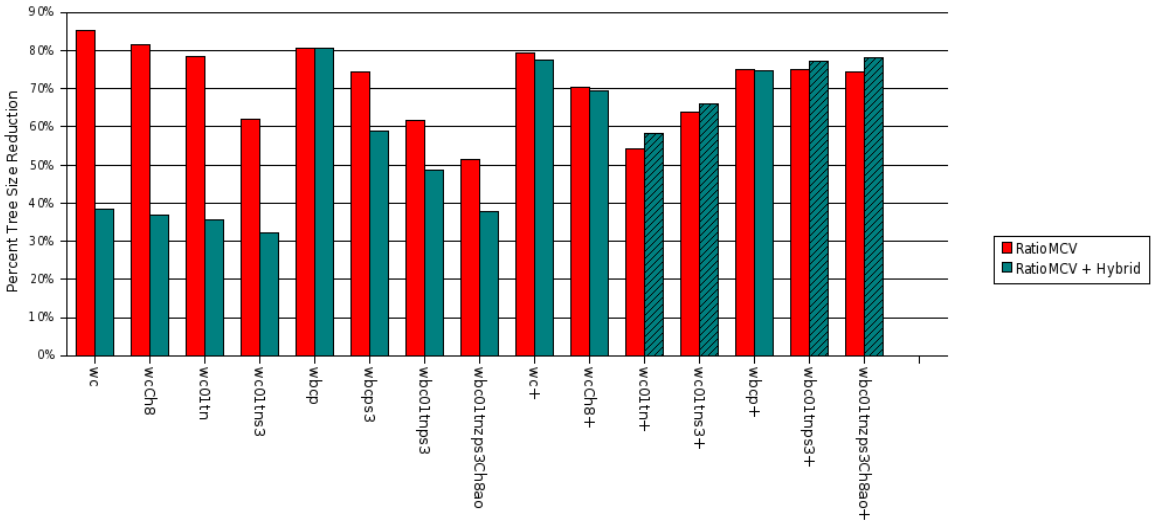

Figure 6.7: Percent tree size reduction relative to the baseline (higher is better). Because the same partial static tree is used for each feature set, the smaller feature sets actually perform much worse. Bars with hash marks did not finish all of the queries in the respective query logs due to memory complications.

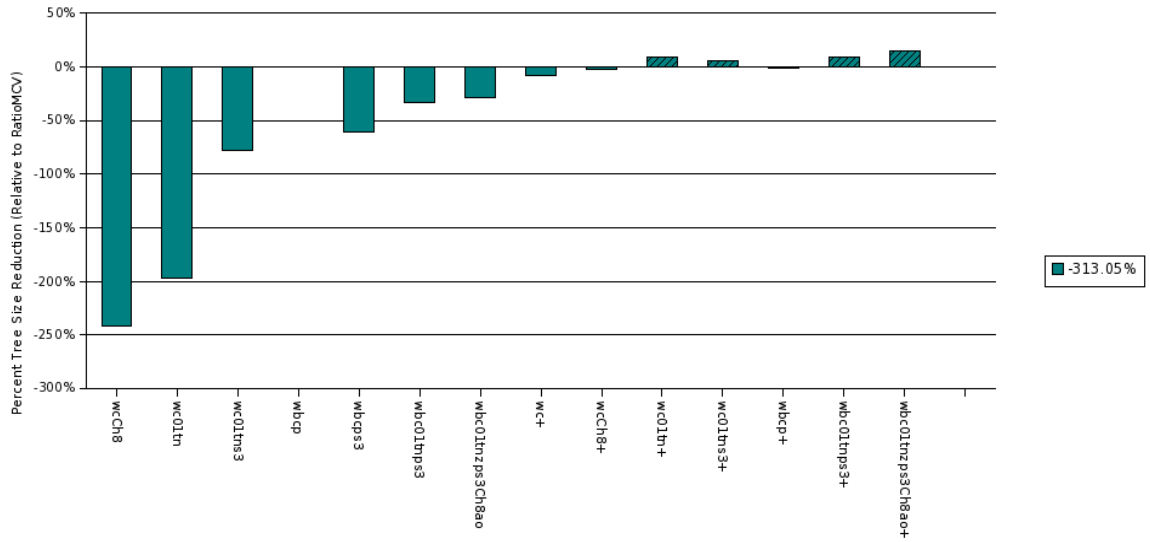

Figure 6.8: Percent tree size reduction relative to the RatioMCV results in Figure 6.7 (higher is better). Here the poorer performance (larger tree size) for smaller feature sets is very clear. Note that since this graph does not use the baseline as a comparison point, the scale of this graph is not relevant to that of Figure 6.7.

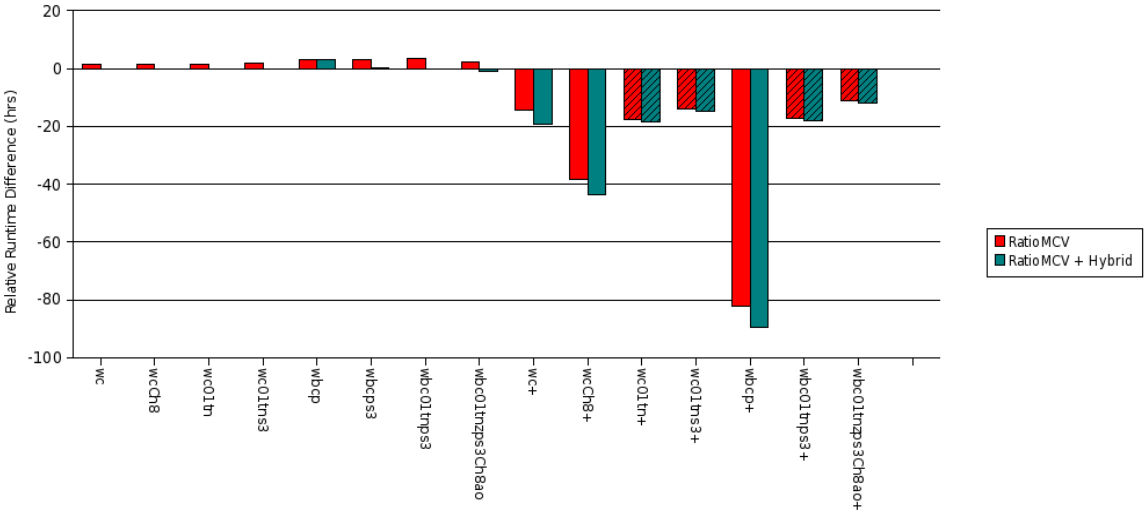

Figure 6.9: Relative difference in runtime (in hours) from the baseline (lower is better). The improvement in runtime is not always large but it is consistent across (almost) all feature sets. It also consistently larger for larger feature sets since the partial static amortizes a larger percentage of the work done constructing the tree. Bars with hash marks did not finish all of the queries in the respective query logs due to memory complications.

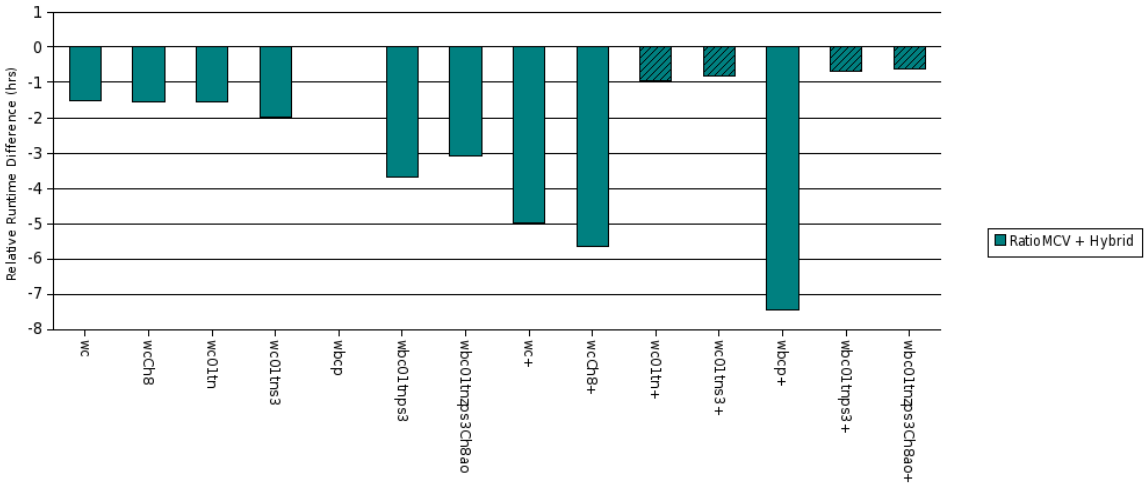

Figure 6.10: Relative difference in runtime (in hours) relative to the RatioMCV results in Figure 6.9 (lower is better). The apparently poorer performance of those bars marked with hash marks is due to early termination caused by memory management bugs. It is reasonable to infer that the trend of increased improvement for larger feature sets would have continued without these complications.

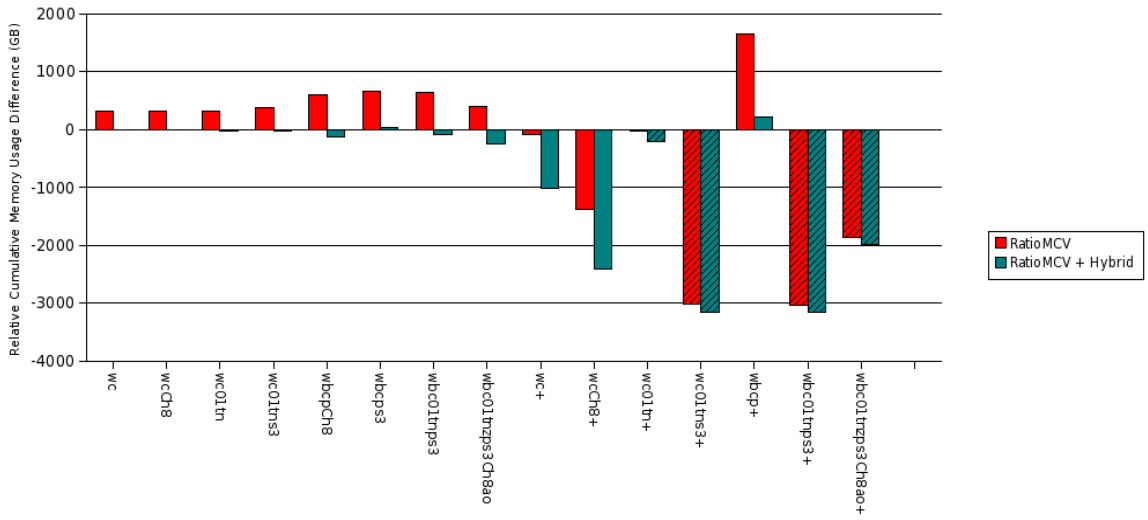

Figure 6.11: Difference in cumulative memory usage (in GB) from the baseline (lower is better). Bars with hash marks did not finish all of the queries in the respective query logs due to memory complications.

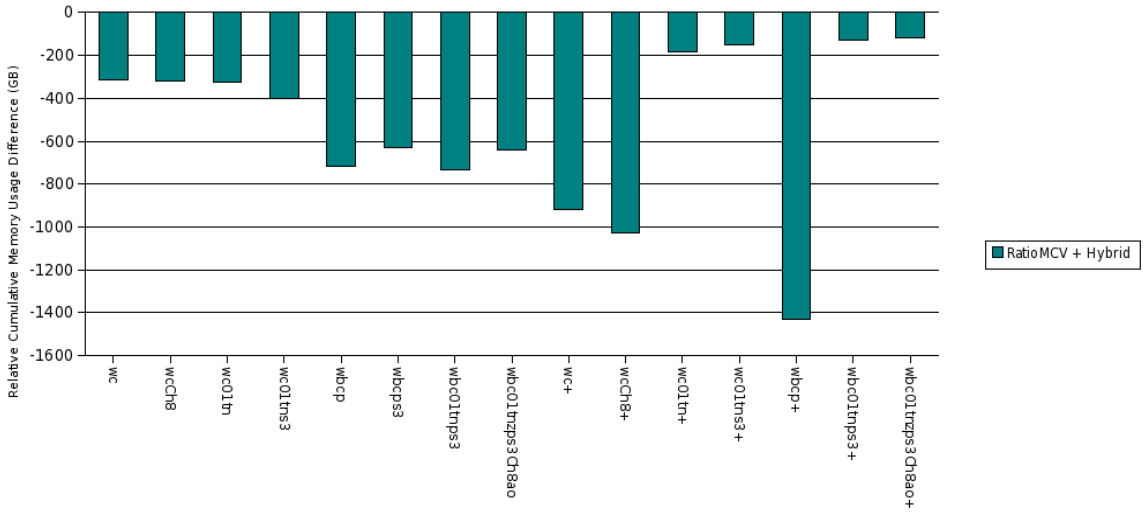

Figure 6.12: Difference in cumulative memory usage (in GB) relative to the RatioMCV results in Figure 6.11 (lower is better). Again, the bars with hash marks presumably would have continued the trends if all queries had been completed.

dynamically building the same set of nodes. However, the cost of building the tree will still generally be greater than the improvements seen in some of the least improved feature sets (particularly those farthest right in the graph that use neighboring words). Therefore, if the tree is to be built only once, building a hybrid tree is not useful for all datasets. However, if that one-time cost can be amortized over many builds of the same or similar trees, both space and time savings will generally be achievable in most situations and for most problems/datasets.

Regarding the issue of using one only static tree with all features, obviously it would also be possible to generate a static two-level tree for use in a hybrid using exactly the features with which the tree would be built. For instance, one such static tree could have been pre-built for all of the feature sets experimented with here. However, this is not representative of many real-world scenarios. It is often the case that at the outset of solving a new research problem, the exact set of features that will be used is still unknown. If it is known that a dynamic ADtree will be needed (due to expected memory constraints), a hybrid tree will almost always be a reasonable compromise. However, the question of which features should be included in the static portion of the tree would still be an open question. Rather than generate one static tree for each possible subset of features, a single static tree with all possible features could be generated and used throughout the solution development stages. Then, once the set of features has been fixed, a new static tree could be built to exactly those specifications. The results obtained here show that using a hybrid with unnecessary features, though clearly not optimal, will still provide a significant improvement in runtime over a simple dynamic approach without exhausting available memory resources. Then, once the feature set has been determined, the excess memory use can be eliminated by simply generating a single, new two-level tree.

6.3 Ordering

Throughout the experiments presented here, it has been assumed that the variety of subsets of features of the enhanced WSJ corpus could serve as a reasonable substitute for comparing results across a wider range of datasets. The results for Complete Vary Nodes, RatioMCVs, Clump nodes, and hybrid trees indicated that this was not an unreasonable assumption to make, with interesting and illustrative differences among the various feature sets. Unfortunately, the results of trying different ordering strategies bring to the forefront a particular weakness in this assumption. As can be seen in Figures 6.13 and 6.14, the tree sizes tend to cluster into relatively clean groups. Those two groups are almost entirely defined by where in the ordering each strategy places the features 'word' (w) and 'base' (b). More specifically, the two main clusters are determined by the placement of feature 'word' (w) and instances where alphabetical (dictionary) and its reverse break from the clusters is determined by placement of the feature 'base' (b).

There are likely two main reasons why 'word' (and to a lesser extent 'base') so clearly dominate these results. The first is that all the features for the central word (that is, excluding the neighboring words) are very strongly correlated to word. Therefore, if 'word' occurs high in the tree, there should be significantly less branching in those paths following 'word'. Similarly, since 'base' is the morphological stem of 'word', in many ways its behavior in the tree mimics that of 'word' (although the relative ordering of 'word' and 'base' appears to be the primary cause of the distinct paths of the dictionary and reverse dictionary orderings). The second reason is simply that 'word' and 'base' are the two highest arity features and so they quickly subdivide the dataset such that, as was anticipated, very little further branching occurs before the leaf-list threshold is reached.

In general, the four orderings which appear to perform the best (both in terms of tree size and runtime) are high entropy, high arity, canonical, and reverse dic-

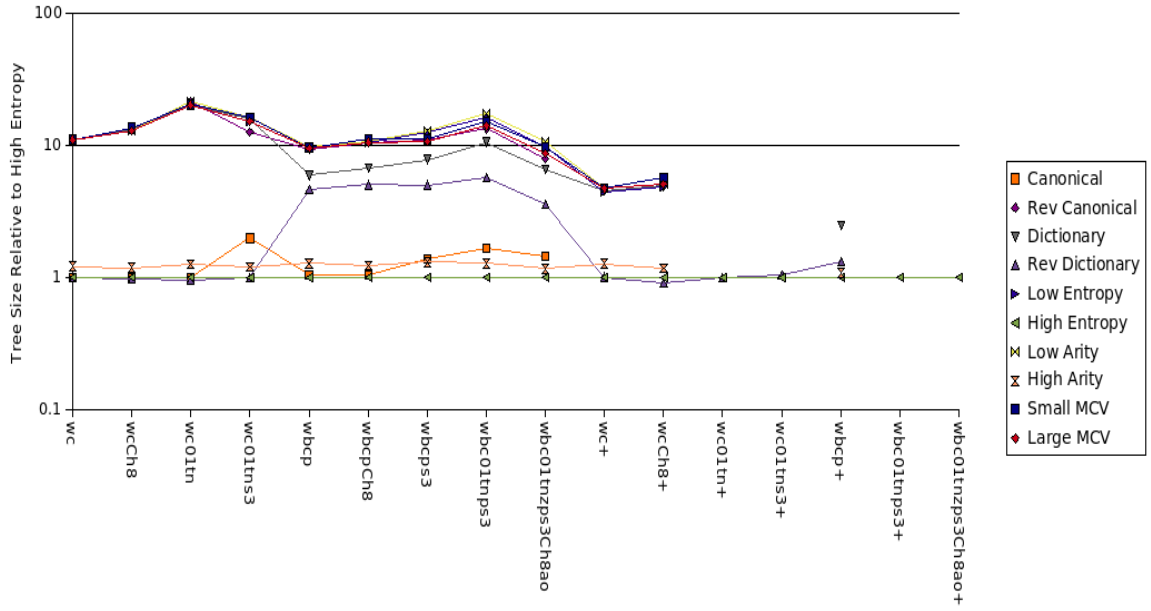

Figure 6.13: Tree size of each ordering strategy relative to High Entropy feature ordering (lower is better). A value of 10 is 10 times worse than High Entropy. The ordering strategies tend to cluster into two groups, with the notable exception of Reverse Dictionary and to a lesser extent Dictionary (over only a portion of the feature sets).

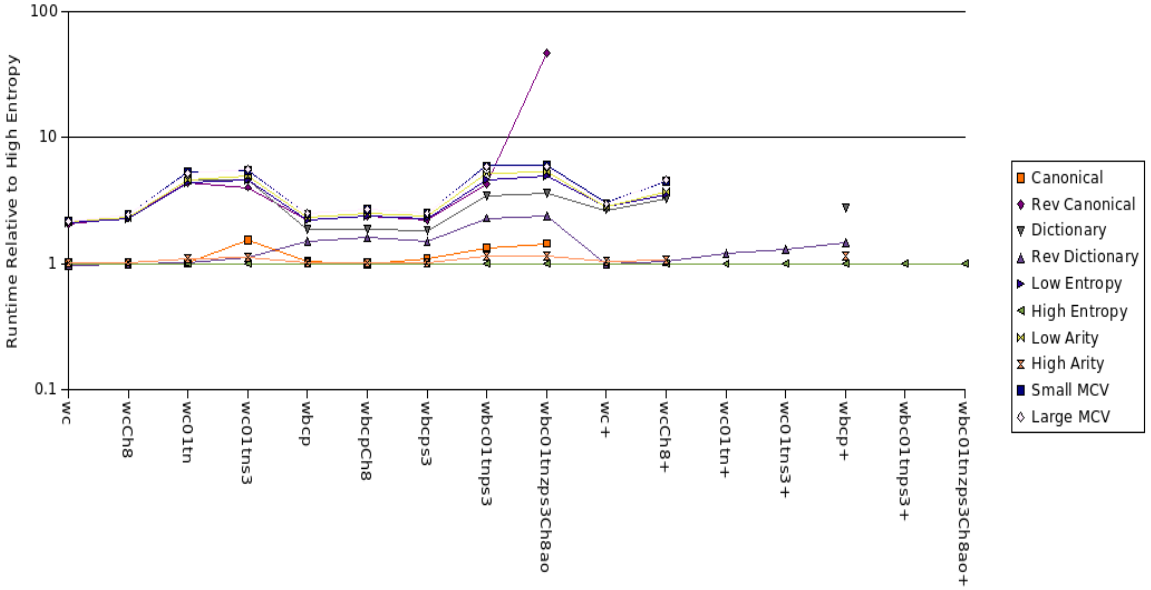

Figure 6.14: Runtime in minutes of each ordering strategy relative to High Entropy feature ordering (lower is better). A value of 10 is 10 times worse than High Entropy. Again, the orderings tend towards two main groups (although Dictionary and Reverse Dictionary again show some deviation from this.)

tionary (alphabetical) in roughly that order. Figures 6.15-6.18 show the data from Figures 6.13-6.14 transposed so that trends with respect to each feature set can be better observed. Figures 6.15 and 6.17 present the ordering strategies sorted such that pairs of forward and reverse strategies are adjacent to one another. Figures 6.16 and 6.18 sort the ordering strategies according to the tree size of the feature set 'wbc01tnzp3ch8ao'[2] (even though the runtimes are then not exactly in order) making the difference between the two clusters of ordering schemes very explicit.

Interestingly, both strategies based on the count of MCVs faired poorly, whereas for all other pairs, one of each pair performed well. Although this is certainly due in part to relative placement of 'word' and 'base' in the orderings, it may also be likely that the absolute size of the MCV is not a useful indicator. In particular, it is likely that relative MCV size (meaning, what percentage is the MCV among its siblings) could perform somewhat better that an absolute comparison. On the other hand, the absolute and relative MCV size metrics would be indentical at the first level of the tree (at which point all rows in the dataset are still relevant) and at least in cases observed here, the first level ordering appears to generally dominate the overall tree size.

Another interesting trend which can't as easily be discerned from the figures occurs among those datapoints which have been left out. As was explained previously, the trees for each set of features are built using a log of queries as generated by the original tagger, rather than re-running the tagger in each instance. This establishes an upper limit for each feature set in terms of number of sentences, which no amount of modifying the tree can surpass. The cases where the combination of feature set and ordering strategy did not finish all of the available sentence taggings have been removed from the graphs, resulting in some gaps in the lines. Although these cases are incomplete and therefore less reliable, there are some trends that indicate that for

[2]'wbc01tnzp3ch8ao' was chosen because it is the largest set for which all the orderings finished.

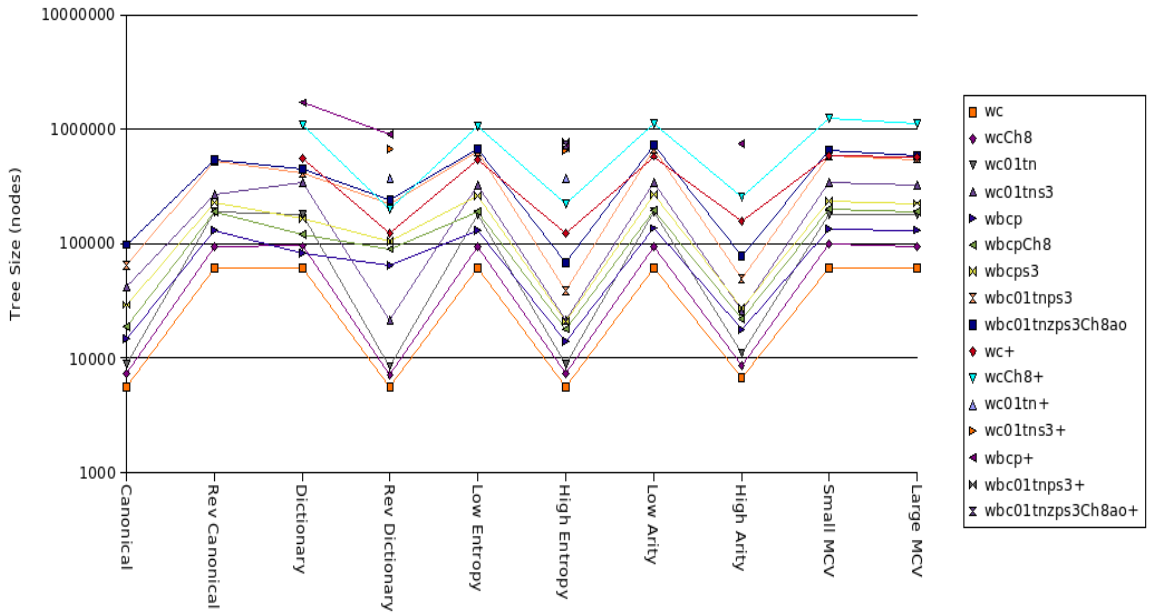

Figure 6.15: Actual tree size of each ordering strategy (lower is better). The x-axis is sorted such that pairs of forward and reverse ordering strategies are next to each other. Note the log scale for the y-axis. Most of the pairs has one ordering which clearly outperforms the other. Only the orderings related to MCV both perform poorly.

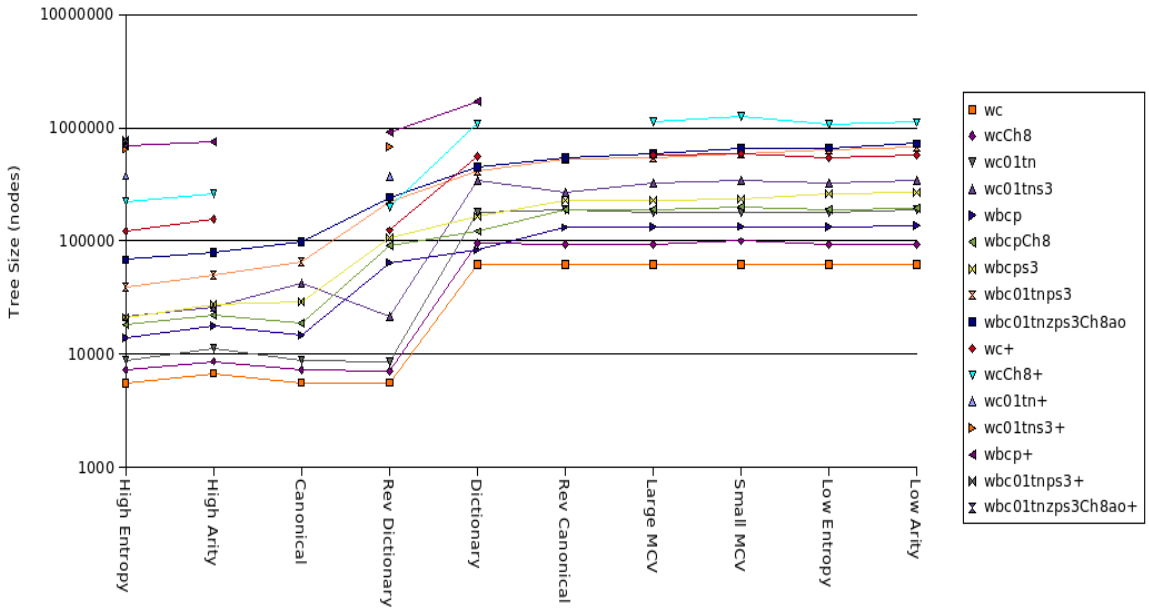

Figure 6.16: Actual tree size of each ordering strategy (lower is better). The x-axis is sorted according to the tree size obtained by wbc01tnzp3ch8ao. This sorting more clearly illustrates the two basic groups of orderings. Note the log scale for the y-axis.

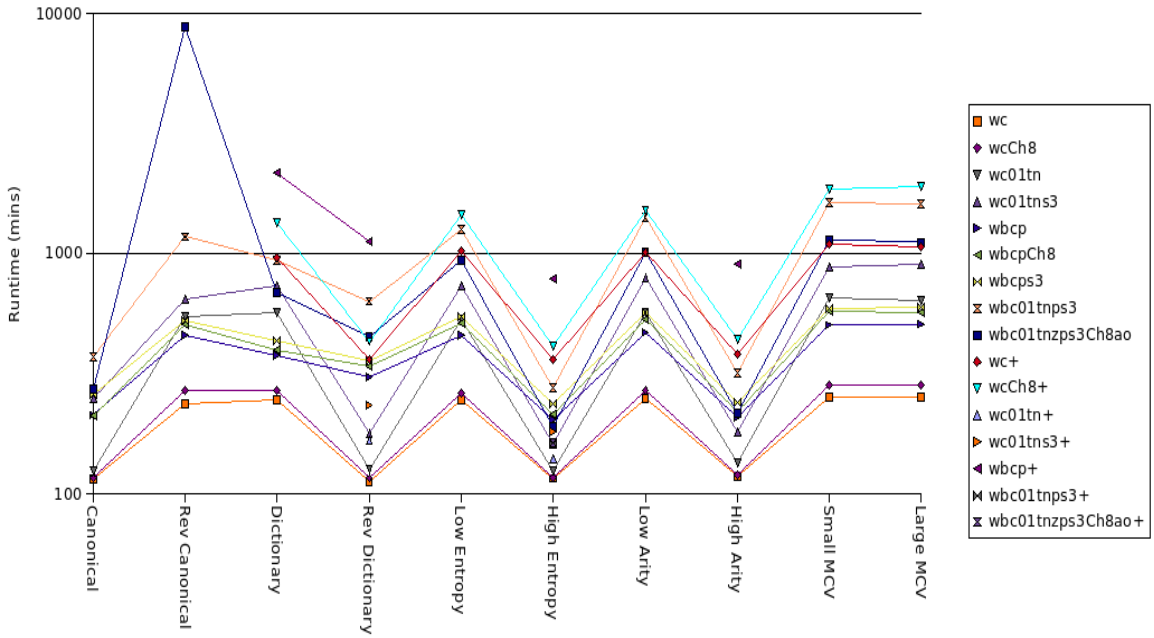

Figure 6.17: Actual runtime of each ordering strategy (lower is better). The x-axis is sorted such that pairs of forward and reverse ordering strategies are next to each other. The alternating pattern of ordering pairs repeats as with tree size. Note the log scale for the y-axis.

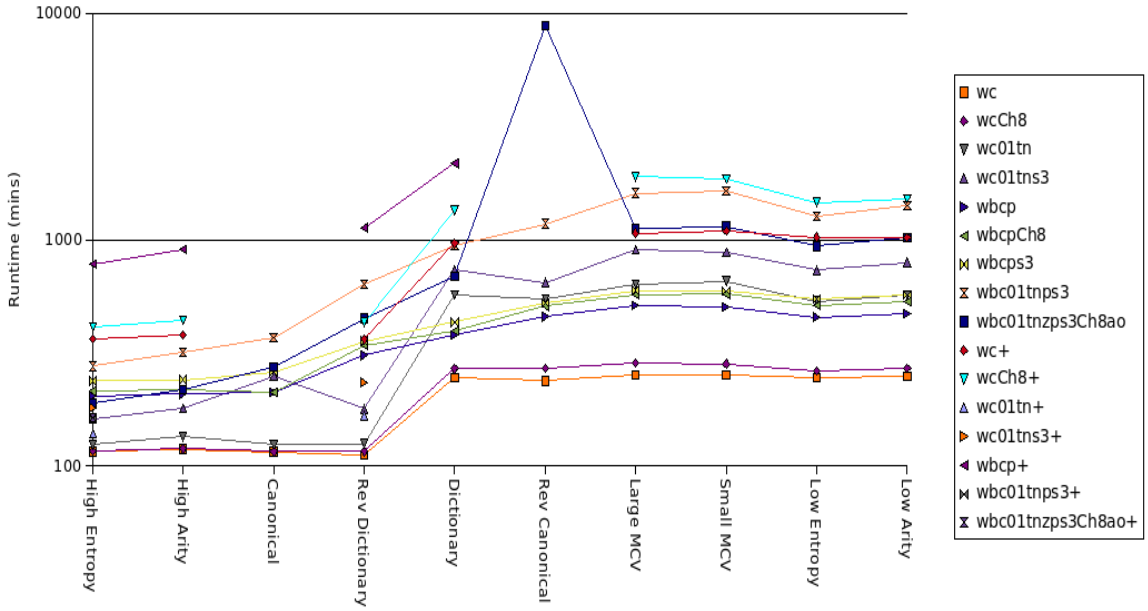

Figure 6.18: Actual runtime of each ordering strategy (lower is better). The groupings remain approximately the same as with the tree size. The x-axis is sorted the same as in Figure 6.16. Note the log scale for the y-axis.

some of the largest feature sets, the ordering strategies that performed well actually swap with their reversed pair. Although further research would have to be performed to determine whether these trends are meaningful, the possibility still exists that highest entropy is not always the best ordering as it appears to be from the current results.

6.4 Pruning

The purpose of performing pruning within a dynamic ADtree is to try to reduce the overall tree size without significantly increasing build time. The particular pruning strategy, therefore, has to balance the reward of pruning with the risk of wasting time re-generating parts of the tree. The pruning strategy chosen in this case removes the low count ADnodes of high arity features after every 100 sentences are tagged. Figures 6.20 - 6.25 summarize the results in a format similar to that used for the hybrid tree. There are fewer feature sets represented here because several of the larger feature sets had query logs that covered less than 100 sentences and therefore no pruning occurred.

As was expected, pruning contributed significant tree size reductions to all feature sets, ranging from 5-25% on top of the already reduced RatioMCV trees. Interestingly, pruning appears to have contributed little to no additional runtime or cumulative memory usage. The largest runtime increase ('wbcp+') was 1.65 hours or approximately 2%, and the largest memory usage increase ('wcCh8+') was 8.7 GB or about 0.6%. Since pruning requires no additional data structures, no significant increase in either memory usage or runtime is expected. However, it is not unreasonable to consider the possibility that extra garbage collection could cause a slower overall runtime for some implementations. Unfortuantely, for the purposes of measuring memory usage, garbage collection was artificially triggered after each sentence was tagged and this likely masked any extra garbage collection triggered by prun-

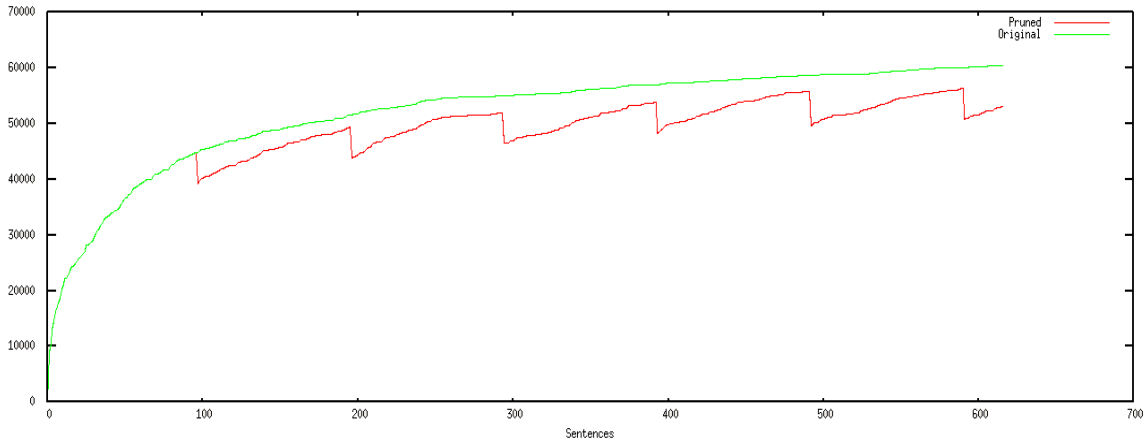

Figure 6.19: Example of how tree size varies as sentences are tagged using the feature set wbc01tnzps3Ch8ao with and without pruning. The pruning strategy only removes a small percentage of the tree, some of which is re-built over the course of subsequent queries.

ing. Furthermore, since this particular pruning strategy prunes the tree at most 13 times (there are slightly more than 1300 sentences), it is not anticipated that garbage collection could cause a significant increase in runtime..

Additionally, the values shown in Figures 6.20 - 6.25 represent the values obtained after the last sentence was tagged. However, since this is not necessarily immediately after the occurrence of a pruning, these numbers represent approximately "average" performance over the RatioMCV baseline. As can be seen in Figure 6.19, after each pruning the tree size continues to grow. This continued growth occurs due to a combination of regrowing some pruned nodes and adding previously unqueried nodes. The higher the rate of increase of the regrowth (relative to the unpruned rate of increase), the more nodes are being regenerated. Some sections appear to have more replacement growth than others but overall it appears that despite this pruning strategy's overall risk averse design, nodes are being removed that are later needed again. However, there are sufficient nodes that are never used again that the pruned tree always remains below the unpruned tree.

On the other hand, in some situations in which memory is severely limited,

this pruning strategy might be considered insufficient. In that case, since the pruning here seems to perform without using significantly more time (given the caveat about garbage collection above), a very aggressive pruning approach (such as pruning the tree to a predetermined size) could still be usable despite the expected increase in the space/time tradeoff.

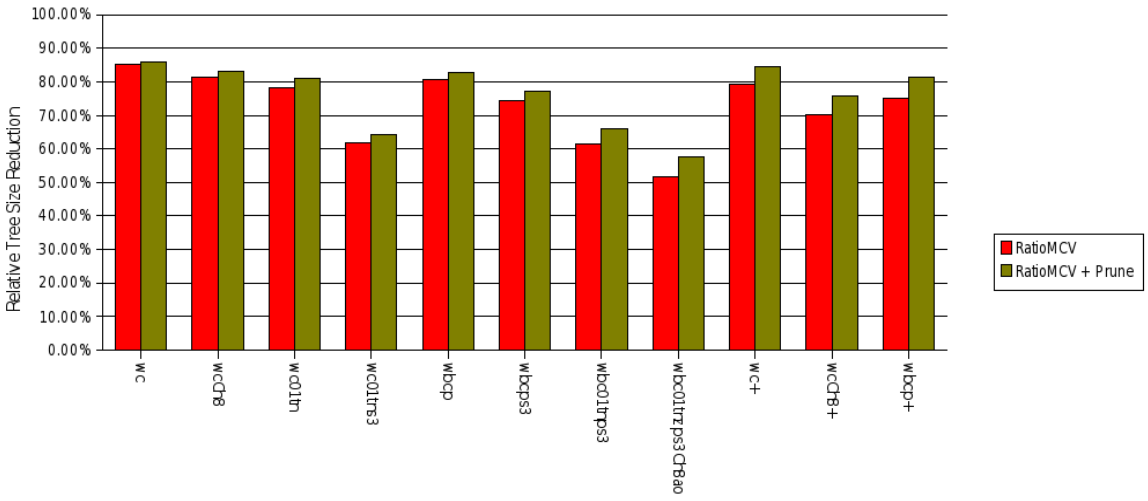

Figure 6.20: Percent tree size reduction relative to the baseline when using pruning compared the results for Complete Vary Nodes and RatioMCVs (higher is better). Tree size is slightly better for all feature sets and performs the best for larger feature sets.

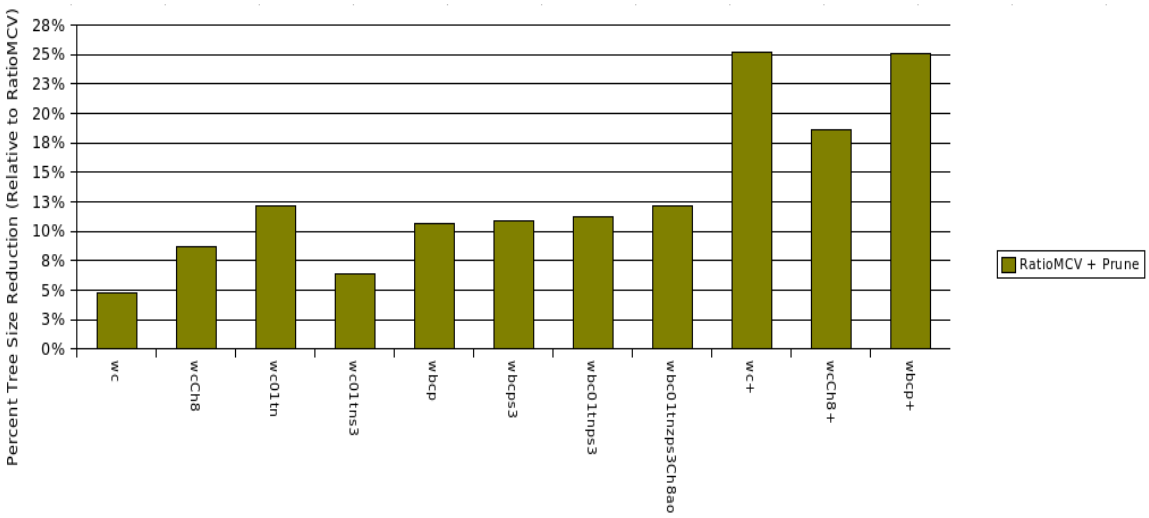

Figure 6.21: Percent tree size reduction relative to the RatioMCV results in Figure 6.20 (higher is better). Here the trend of improved performance as feature set size increases is fairly clear.

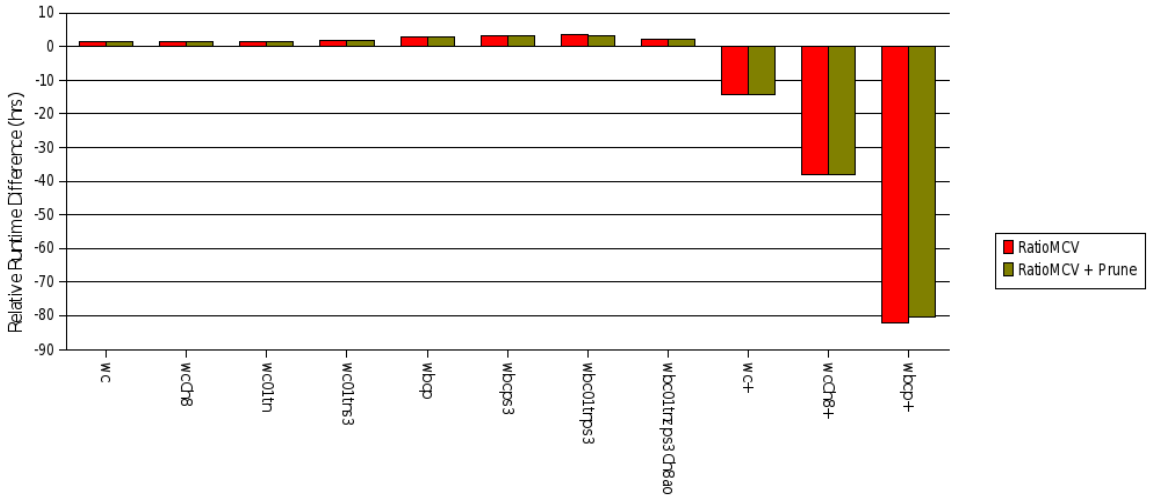

Figure 6.22: Relative difference in runtime (in hours) from the baseline (lower is better). Pruning is a fairly inexpensive operation timewise and so the results are not significantly different.

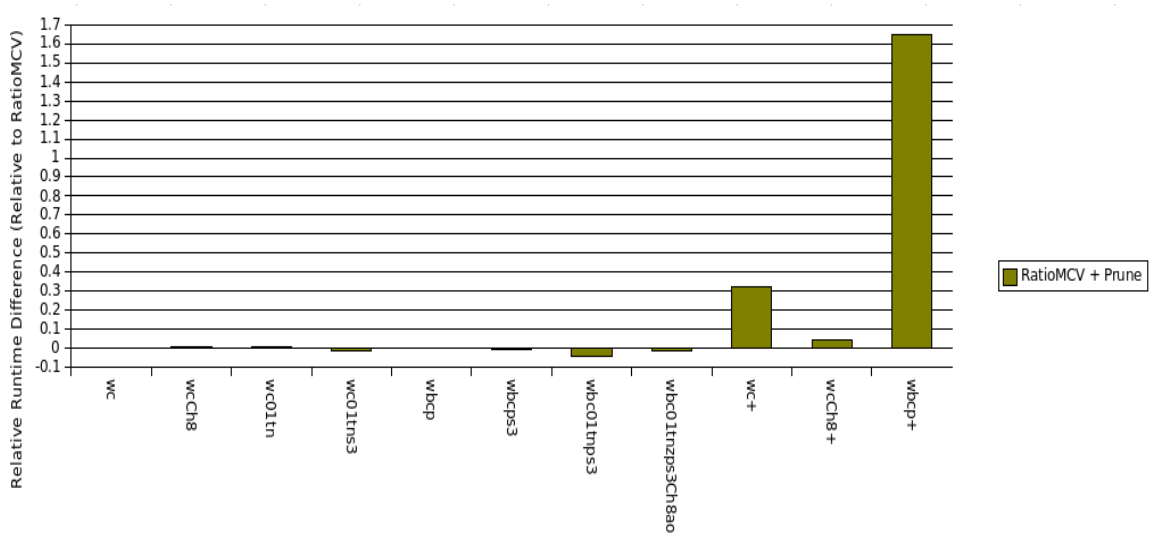

Figure 6.23: Relative difference in runtime (in hours) relative to the RatioMCV results in Figure 6.22 (lower is better).

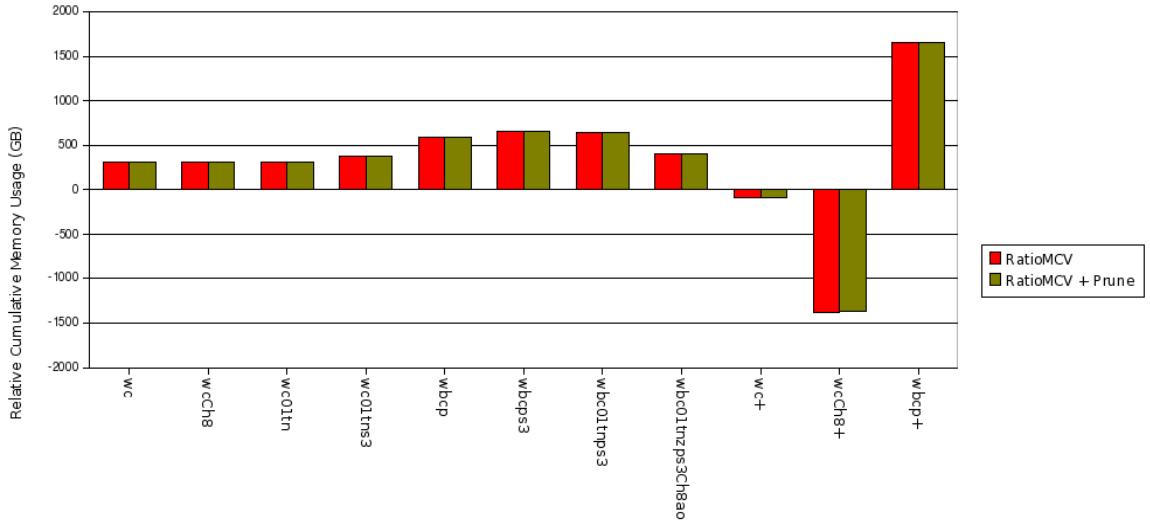

Figure 6.24: Difference in cumulative memory usage (in GB) from the baseline (lower is better).

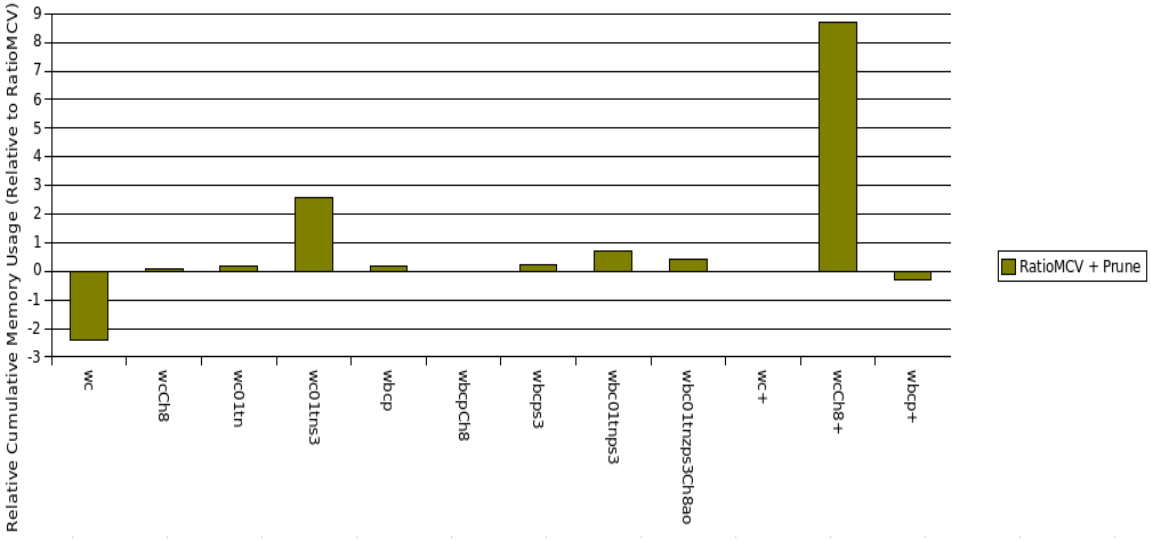

Figure 6.25: Difference in cumulative memory usage (in GB) relative to the RatioMCV results in Figure 6.24 (lower is better).

Chapter 7

Conclusion

Despite the generally efficient design of the ADtree, the preceding results clearly illustrate that the modifications presented provide both generally applicable improvements as well as improvements that can be targeted at the specific characteristics of a given dataset. In particular, the addition of Complete Vary Nodes, RatioMCVs, and Clump nodes provide substantial improvements for datasets containing high arity attributes. The hybridization technique and pruning strategy presented here can be readily applied to any application of ADtrees in which using a dynamic tree is expected to be necessary. The ordering strategy based on highest entropy appears in these initial results to be generally applicable. Although this might not turn out to be true for all datasets, it certainly provides a beginning from which more advanced and/or more data specific ordering strategies can be developed.

Although there are several ways in which each of these modifications might individually be explored and developed further, in general, the results presented here would be complemented by further study involving a wider range of datasets. Then, need for more variety in the datasets is most pronounced in the results obtained from the tested ordering strategies, but the ability to duplicate the positive results on additional datasets would help to firmly establish the efficacy of each modification. Additionally, by testing on more, distinct datasets, it would be possible to develop a better sense of what data characteristics influence each technique. For instance, it could be very informative to determine if the improvements for high arity attributes

are maintained outside of the domain of natural language. It would also be interesting to explore a wider variety of pruning strategies in order to help determine how often and how much of the tree is reasonable to prune, given different memory and time requirements and/or restrictions.

Bibliography

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

[2] J. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the Association for Computing Machinery*, pages 509–517, 1975. 5

[3] C. Blake and C. Mertz. *UCI Repository of Machine Learning Databases*. 1998.

[4] R. Caruana. Multitask learning. *Machine Learning*, pages 42–75, 1997. 2

[5] R. Van Dam, I. Langkilde-Geary, and D. Ventura. Adapting adtrees for high arity features. In *Proceedings of the Association for the Advancement of Artificial Intelligence*, To appear 2008. 4

[6] R. Van Dam and D. Ventura. Adtrees for sequential data and n-gram counting. In *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics*, 2007.

[7] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *International Conference on Computer Graphics and Interactive Techniques*, pages 124–133, 1980. 5

[8] V. Gaede and O. Gunther. Multidimensional access methods. *Association for Computing Machinery Computing Survey*, pages 170–231, 1998. 5

[9] P. Komarek and A. Moore. A dynamic adaptation of ad-trees for efficient machine learning on large data sets. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 495–502, 2000. 6

[10] I. Langkilde-Geary and J. Betteridge. A factored functional dependency transformation of the english penn treebank for probabilistic surface generation. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, 2006.

[11] M. Marcus, B. Santorini, and M. Marcinkiewicz. Building a large annotated corpus of English: the Penn treebank. *Computational Linguistics*, 19(2), 1993.

[12] A. Moore and M. S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998. 5, 9

[13] J. Moraleda and T. Miller. 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*, pages 313–320, 2002. 13

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

[15] R. Rymon. An se-tree based characterization of the induction problem. In *International Conference on Machine Learning*, pages 268–275, 1993.

[16] K. Toutanova and C. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 63–70, 2000. 16