Ordering Depth First Search to Improve AFD Mining

Jeremy T. Engle
Indiana University
Lindley Hall 215
Bloomington, Indiana, USA
jtengle@indiana.edu

Edward L. Robertson
Indiana University
Lindley Hall 215
Bloomington, Indiana, USA
edrbtsn@indiana.edu

Dimitar G. Nikolov
Indiana University
Lindley Hall 215
Bloomington, Indiana, USA
dnikolov@indiana.edu

ABSTRACT

This paper describes a new search algorithm, bottom-up attribute keyness depth-first search (BU-AKD), for mining powerset lattices with the use of a monotonic approximation measure; characteristics present in many problem domains. The research reported here focuses on one of these problem domains, the discovery of Approximate Functional Dependencies (AFDs). AFDs are measured versions of functional dependencies, which have received attention from the relational database and machine learning communities. Bottom-up depth-first search, BU-DFS, algorithms in general can improve efficiency over the traditional bottom-up breadth first search algorithm by doing a better job of avoiding the calculation of the approximation measure based on information learned as the search space is explored. The goal of BU-AKD is to resolve one important drawback of BU-DFS algorithms which makes their use in practice problematic - their inconsistent runtime performance as search parameters vary. The approach that BU-AKD takes is to use a heuristic to guide the exploration of a lattice which adapts to the search parameters, thus, providing consistent performance comparable to best-performing BU-DFS algorithms. This paper reports a variety of experiments which evaluate BU-AKD and other algorithms using an algorithmic, machine-independent cost measure, as well as traditional runtime tests. Experimental results show that BU-AKD performs consistently well and validates a number of insights used in its design.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; H.2.8 [**Database Applications**]: Data mining

General Terms

Algorithms, Design, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Keywords

AFDs, data mining, powerset lattices, MoLS, BU-AKD

1. INTRODUCTION

This paper focuses on the development of a heuristic variation of depth first search (DFS) algorithms, attribute keyness depth first search (BU-AKD), in the context of Approximate Functional Dependency (AFD) mining. AFD mining is among those data mining problems where the search space is composed of powerset lattices and a parameterized threshold is used to determine results. The motivation for developing BU-AKD is to improve algorithm efficiency by reducing the number of AFDs for which the measure of "approximateness" is calculated. This is in contrast to focusing on optimizing the cost of calculating "approximateness". Existing research, which reduces the number of times the approximation measure is calculated, has shown that DFS algorithms can outperform the traditionally used bottom-up breadth first search, (BU-BFS) [10], but never in a consistent way as parameters are varied. This is a significant drawback that makes applying DFS algorithms problematic, because even with good best- and average- case performance, a DFS algorithm's inconsitency makes poor performance an eventuality instead of a possibility. This motivates the development of a heuristic DFS algorithm which can adjust to varying search parameters and avoid spikes of catastrophically poor performance, thus balancing performance and consistency. This paper describes the underlying insights, implementation, and evaluation of BU-AKD and the factors behind its ability to improve performance in a consistent manner.

An AFD, X $\rightarrow$ Y, is a rule which expresses with measured likelihood the ability of data values for attribute set X to determine data values for attribute set Y. AFDs provide knowledge about the attributes (columns) of a relation (table) rather than knowledge about particular values. A specific value-level rule, such as $O^- \rightarrow UniversalDonor$, is a single fact. The attribute-level analog, $BloodType \rightarrow Compatability$, is generalized knowledge that is more broadly applicable. While $BloodType$ is typically a concatenation of attributes ABO and Rh, there are actually many attributes (called "systems" in hematology) reflecting the presence of rare antigens. Discovered AFDs involving these attributes and disease susceptibility (or other phenotypes) could provide significant medical knowledge.

Disease susceptability is only one example of a domain where AFD mining can be applied. AFDs represent statistical relationships which potentially reveal certain data semantics. Though we focus on the AFD discovery process,

the task of mining AFDs is usually secondary to some other task. Because AFDs can be used to make inferences about data in a probablisitic way, they have been used in areas ranging from relational database management to predicting cancer to bootstrapping a Bayes classifier. What all of these applications share is that they use AFDs as a form of generalized knowledge about the data. Given such uses, AFD mining can potentially be applied in the information extraction context to generate data semantics to be used in engines relying on formal logic or probabilistic reasoning.

The measured likelihood of an AFD is referred to as a rule's approximateness, or approximation value, and is determined by an approximation measure, formally described in Section 2. The goal of AFD mining is to find *pass* rules; rules in the power set of rule combinations whose approximation value is less than a parameterized threshold.

The cost of mining AFDs is dominated by the product of two factors: the cost of calculating the approximateness of an AFD multiplied by the combinatorially large number of times such calculations are done. This binary cost structure allows us to cluster AFDs into considered nodes which have their approximation value calculated and pruned nodes which are removed from the search space. Here, the cost of mining is proportional to the number of rules considered. The other side of controlling costs focuses on optimizing the calculation of an approximation measure.

The Modular Lattice Search (MoLS) framework [10, 11] was used to develop and test the algorithms discussed in this paper. MoLS was developed to support generalized AFD mining, using customization to meet the needs of a specific problem. One of MoLS's advanatages is providing tools for *inferencing*. Inferencing determines the *pass*/*fail* status of a node at lower cost than calculating the approximation measure by using the *pass*/*fail* status of ancestors/descendants nodes. DFS algorithms are especially adept at using inferencing. BU-AKD focuses on balancing the inherent DFS tradeoff between lowering costs with inferencing and increasing costs when considering a larger search space. In addition, BU-AKD is able to do so even when different approximation measures are used.

The design of BU-AKD uses insights about the effects on the approximation value of distribution properties of data values to guide the exploration of the search lattice with the intent of increasing inferencing. These insights are captured in the Attribute Key (AK) heuristic which determines how BU-AKD explores the lattice. By using properties of the underlying data, AK provides an ability to adapt to changing search parameters. We will demonstrate in experimental results that BU-AKD provides both performance gains and the ability to do so consistently for a wide range of tests.

Section 2 has definitions related to AFD mining. Section 3 surveys other works in the problem domain. Section 4 describes the specifics of BU-AKD. Section 5 covers the conceptual factors which lead to the development of BU-AKD, and finally, Section 6 present experimental evaluation.

2. DEFINITIONS

Throughout the paper we use the following conventions.

- $X \rightarrow A$ is a rule, a pair of attribute sets; henceforth A has only one attribute.
- LHS abbreviates Left Hand Side; RHS, Right Hand Side; BU, Bottom UP; TD, Top Down; BFS, Breadth First Search; DFS, Depth First Search.

AFDs are an extension of Functional Dependencies (FDs), which are integrity constraints found in database theory. The definition of an FD is absolute, in that $X \rightarrow A$ fails to hold in an instance, $\boldsymbol{r}$, if there are two tuples that agree on values for X but disagree on values for A. The precise definition of an FD can be found in [19] or most introduction to database text books. An approximation measure is used to characterize how close an AFD is to being an FD and is required to have the following properties.

Definition 2.1 *An* approximation measure *is a function from two attribute sets, X and A, to $[0, 1]$, written $\varphi(X \rightarrow A)$, such that*

(a) $\varphi(X \rightarrow A) = 0$ iff the FD $X \rightarrow A$ holds,

(b) $A \subset B \Rightarrow \varphi(X \rightarrow A) \leq \varphi(X \rightarrow B)$, and

(c) $X \subset W \Rightarrow \varphi(X \rightarrow A) \geq \varphi(W \rightarrow A)$.

All approximation measures are required to be monotonic, which allows techniques based on it to be independent of a specific measure. Monotonicity allows one of two statements to be made for a node whose *pass*/*fail* status is known: the sublattice of all of a *pass* node's ancestors are also *pass* nodes, and the sublattice of all of a *fail* node's descendants are also *fail* nodes. Figure 1 provides examples of the inferable sublattices knowing only the *pass*/*fail* status of A and CD.

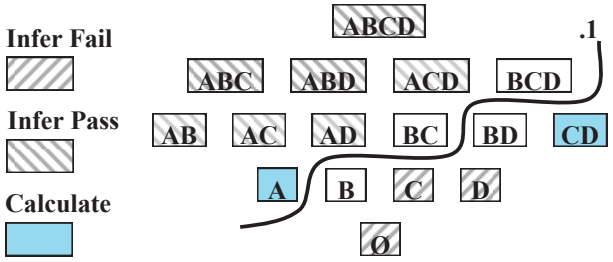

Figure 1: *pass* and *fail* inferable sublattices

We present results using the two most common approximation measures InD and g_3. The normalized version of the information dependency measure InD [6] is defined as $\varphi(X \rightarrow A) = \frac{\mathcal{H}_{XA} - \mathcal{H}_X}{\mathcal{H}_r}$, where $\mathcal{H}$ is Shannon's entropy [20], measured as manifested in a database relation. The g_3 measure [14] is the percentage of rows which would be removed for a rule to meet the strict definition of an FD.

Lattice theory terms such as lattice, sublattice, etc. are augmented from their standard meanings to deal with AFDs having both a LHS and RHS. A lattice of rules is defined as the power set of LHS attribute sets and each rule has the same RHS attribute.

Definition 2.2 *$X \rightarrow A$ is a* parent *of $W \rightarrow B$ when $W \subset X$ and $|X| = |W| + 1$ and $A = B$. Additionally* child*,* descendant*, and* ancestor *have their usual interpretation with respect to* parent*.*

AFD mining is the search for minimal, in terms of attributes, rules that are "close enough" according to a threshold, ϵ, and an instance $\boldsymbol{r}$ which is an implicit parameter.

Definition 2.3 *$X \rightarrow C$ is a* pass rule *provided that $\varphi(X \rightarrow C) \leq \epsilon$. When, in addition, for all Y, $Y \subset X$, $\varphi(Y \rightarrow C) \leq \epsilon$ does not hold, it is a* minimal pass rule *(written minP).*

Definition 2.4 *$X \rightarrow C$ is a* fail rule *provided that $\varphi(X \rightarrow C) > \epsilon$. When, in addition, for no Y, $X \subset Y \subseteq Z$, $\varphi(Y \rightarrow C) > \epsilon$, it is a* maximal fail rule *with respect to Z (written maxF(Z)). The context Z is omitted if it is in fact all attributes or the current set of active attributes.*

Correctness when using MoLS means finding the same set of *minP* rules as an exhaustive search.

A boundary determined by ϵ lies between *pass* and *fail* rules. More precisely, we can give the following definition:

Definition 2.5 *A boundary rule is a pass rule with a fail child or a boundary rule child.*

This boundary can be characterized by the sets of *minP* and *maxF* rules.

The *minP* and *maxF* sets and a threshold boundary restate the idea from FIM of positive and negative borders [22] as they apply to AFD mining.

3. RELATED WORK

AFD mining is in a broad group of data mining problems where the search space is composed of powerset lattices. This group includes but is not limited to: FDs, conditional functional dependencies [18], association rules [1], and Frequent Itemsets (FIs) [12]. While these problems have structurally similar search spaces, their syntax and semantics are less similar. The commonalities in search spaces allow speculation that techniques for AFD mining could be adapted to these other domains. However, it is unclear how effective they would be relative to existing work in those fields. Section 7 presents a high level discussion of impact this work could have on fields outside of AFD mining.

There are two significant approximation measures in the AFD mining field. The g_3 approximation measure is discussed below. The other significant approximation measure is Information Dependency (InD) [6], developed by Dalkilic and Robertson as an extension of Shannon's entropy [20].

There are a number of interesting works which have focused on the algorithms used to explore power set lattices. The first is in the FIM field by Dexters *et al.* referred to as the peak-jumping algorithm [7] whose goal was an algorithm which skipped levels of the lattice to avoid evaluating some itemsets. TANE, the most widely used AFD mining algorithm, was developed by a group at University of Helsinki. Their work focused on the development of the g_3 measure [14] and a partition-based data structure which facilitates efficient evaluation of g_3 [26]. At its core, TANE uses a traditional bottom-up BFS algorithm for navigating lattices. Other than specifying the parameter ϵ and, of course, the data set to be mined, TANE allows no customizations.

Other algorithms include B&B, reported in [4], which originally mined for FDs. It was extended in [15] to mine for AFDs by using the same algorithm with g_3. Similar to TANE, B&B is based on bottom-up BFS but differs in some details. Finally, Lopes *et al.* [21] used a formal concept analysis approach along with g_3 in a framework, though it is unclear how different plug-ins could be used in that framework. A final work focusing on algorithms is the Data Mining Template Library (DMTL) [23] which is a set of generic programming libraries. The DMTL project was motivated by the recognition that there are universal aspects of search so a set of high performance libraries makes developing new algorithms easier while also improving performance.

Beyond the mining process, the use of AFDs has spread to a variety of domains. The original motivation for AFD mining was to extend FD-based query optimization [5, 13]. More recently, researchers have used AFDs for machine learning problems [17, 3, 24, 25, 16]. In all of these uses, the developers have some unique needs so they build *ad hoc* systems that add customized pre/post mining modules to one of the existing algorithms. An AFD mining algorithm which can be customized was the motivation for developing a flexible algorithm framework, but the focus of this paper is taking MoLS and customizing it for performance, as well as describing the general principles behind BU-AKD.

4. DFS AND PATH SELECTION

A DFS algorithm is characterized by a path-wise exploration of a lattice. The path is determined by selecting which of a rule's parents are explored next. Choosing a parent equates to selecting an attribute and adding it to the LHS. The path selection plays an important role in determining performance given that a DFS algorithm's path determines what in the search space is considered and what is inferred.

Previous research in AFD mining algorithms [10], has prototyped two path selection functions, bottom-up left-most and right-most DFS, BU-LD and BU-RD, which correspond to textbook versions of DFS. Left-most and right-most DFS are based on labeling attributes with letters and then arranging nodes using lexicographic order, which selects variables according to the spelling of their names. Say the variables are named "A", "B", $\cdots$. Then the nodes are visited in order A, AB, ABC, $\cdots$ until the boundary is crossed, say at ABCDEF, then backtracking occurs. Upon backtracking to ABCDE, the next parent is ABCDEG, and so on. While lexicographic order is essentially arbitrary, it helps to imagine any path selection as merely A, B, $\cdot$.

DFS algorithms are able to use *fail* inferencing because as the algorithm explores a path through the lattice some portion of the inferable descendant sublattice has yet to be considered, and these rules can be inferred. At the same time, DFS's exploration of a lattice allows no guarantees about where it crosses the boundary which means that DFS algorithms will likely consider non-minimal *pass* rules. This makes DFS algorithms non-optimal on the *pass* side of the threshold boundary given that all *fail* rules get considered, like BU-BFS. Thus, it is guaranteed that in terms of rules considered $DFS\ search\ space \supseteq BFS\ search\ space$. It is only with *fail* inferencing that DFS algorithms can offset the additional search space considered and outperform BU-BFS.

We illustrate important details of the workings of search algorithms in MoLS in the pseudo source code in Figure 2. In this code segment, the italicized text refers to functionality provided by MoLS. Appendix A or [10] can be consulted for detail about the workings of these components. We have emphasized in bold text the call to the `generateCandidates` procedure (line 12) where a rule R is expanded and its children added to the queue. It is at this point of the search process that the path selection function comes into play and determines the progression of the search in breadth-first or depth-first fashion.

```
1  PROCEDURE DFSSearch
2     WHILE unexplored rules remain DO
3       rule R =  top priority rule
4       IF R should be explored THEN
5         IF MoLS.isInferred(R) THEN
6           status = MoLS.getPassFail(R)
7           approxVal = null
8         ELSE
9           approxVal = ApproximationMeasure(R)
10          status = approxVal <= Thresh
11        IF status == FALSE THEN
12          MoLS.           generateCandidates(R,
MoLS.queue)
```

Figure 2: The DFS search algorithm.

5. AK PATH SELECTION

The goals for BU-AKD are to achieve good performance and to do so consistently. BU-AKD uses the attribute key (AK) path selection function to heuristically decide which parent rule DFS considers next. The AK heuristic is based on insights about the effects distribution properties of data have on where *minP* rules occur in the lattice. We start by describing the desired behavior for a DFS algorithm which improves performance by balancing the inherent DFS tradeoff of considering a larger search space versus using inferencing to avoid calculating the approximation measure. We then describe the AK heuristic and how it creates the desired behaviors in a DFS algorithm. In Section 6 we evaluate each of these insights, how they correlate with performance, and how consistently they hold.

5.1 Insights on DFS Behavior

Insight 5.1 *fail inferencing is constant for DFS algorithms.*

As a DFS algorithm explores along a path, the inferable descendant sublattice of each *fail* rule subsumes that of any of its children. Therefore, changing the path selection function primarily changes which nodes get inferred and not how many nodes get inferred. In contrast, in the case of pass inferencing it is less likely that descendants of the current node have already been explored, and the decision which child to explore may have a more dramatic effect on the number of rules that are inferred.

Conclusion: Focusing on *pass* inferencing will have a greater impact on performance than *fail* inferencing.

Insight 5.2 *The smaller the number of attributes on the LHS, the more pass rules are inferred.*

Lattice theory states that smaller attribute sets have larger sublattices of ancestors, so to increase *pass* inferencing an algorithm should find the smallest *pass* rules which will have the largest inferable sublattice of ancestors. This insight is based on the fact that BU-BFS has optimal *pass* pruning in part because it guarantees that the first *pass* rule it finds is the smallest in the lattice. Though this insight is straightforward, the question of how an algorithm can consistently take advantage of it is non-trivial.

Conclusion: Crossing the threshold boundary lower will increase an algorithm's *pass* inferencing.

Insight 5.3 *Attributes which have all unique values (keys) always create a FD as a LHS.*

To cross the threshold boundary as low as possible is to find a key for the RHS in as few attributes on the LHS as possible. Since keys will not exist for most RHSs, we need to relax the above statement by allowing the LHS to be "close" to being a key for the RHS and be able to measure closeness. The AK heuristic precisely defines this notion, creating a relative ranking of attributes and thus being able to determine which ones will do the most to bring the LHS closer to a key for the RHS.

The basis for determining closeness, and thus for allowing an algorithm to cross the boundary low, must be an underlying property of the search space, so that as its main components - the data set, threshold, and approximation measure - change, the notion is still applicable. In order to find such a property, we look back to the formal definition of an FD which states that if the values for two tuples on the LHS never agree, the FD definition can never be violated. Because of this, when an attribute set has unique values for every tuple (a key) any AFD with that attribute set as the LHS will, in fact, be a FD regardless of the RHS. Emerging from this observation, the AK heuristic states that the closer an attribute is to having all unique values, the more it will do to form an AFD regardless of the RHS. The extent to which all values of an attribute are unique is referred to as attribute keyness, and is straightforward to calculate using the attribute's values' enthropy.

Conclusion: Selecting the next parent rule according to how close it is to being a key will result in crossing the threshold boundary lower.

5.2 AK Implementation

In Figure 3 we present pseudocode to illustrate how the AK path selection function is used. Parent and children rules will differ in exactly one attribute, so given a rule `R`, generating its parents is done by finding the attributes in the search space not on the LHS or RHS of `R` (line 2). It is then the `parentAttrs` that AK orders by how close they are to having all unique values. To do this, we use the entropy (H) value for each attribute. As seen in line 5, `parentAttrs` are ordered by decreasing entropy. Though not reflected in the pseudocode, the H of every attribute in the search space is cached at the beginning of the search so the only additional time cost for using it is a hash lookup.

```
1 PROCEDURE generateCandidates(R, global frontier)
2    parentAttrs = All Attrs - R.LHS ∪ R.RHS
3    decEntropySet = ∅
4    FOR EACH A IN parentAttrs DO
5      decEntropySet.add(H(A), A)
6    FOR EACH A IN decEntropySet DO
7      frontier.enqueue(R.LHS ∪ A → R.RHS)
```

Figure 3: The AK path selection function.

Given the above description of the AK heuristic, we provide a simple example of how BU-AKD would explore the lattice consisting of the folowing attributes with the given H values: (A .5), (B .8), (C .7), (D .2). Here, a value of 1 means an attribute is a key and 0 means it is a constant. Note that this example only illustrates the path that will be followed and ignores the *pass*/*fail* status of rules. BU-AKD

explores the nodes in the order: B, BC, ABC, ABCD, BCD, AB, ABD, BD, C, AC, ACD, CD, D.

AK was the path selection function which provided the best combination of performance and consistency among several path selection functions we prototyped using MoLS. We note that what makes the above pseudo code straightforward is the interplay of several functions and ideas underlying our system: MoLS's management of what has been explored in the search space, the one-time cost of calculating H values, and, of course, the insight that keyness is a good heuristic to guide the search process.

6. EXPERIMENTAL RESULTS

Our experiments focus on showing support for our overall goals, and also on finding evidence that accomplishing those goals is a result of the insights discussed in Section 5. In all experiments, we evaluate 4 DFS path selection functions. BU-AKD, BU-LD and BU-RD have already been described. BU-ACD is the exact opposite of BU-AKD ordering nodes by increasing entropy.

6.1 Measuring Performance

In the evaluation of BU-AKD, we take two different approaches for measuring BU-AKD's performance. First, we use a traditional measure of an algorithm's performance, clock runtime. Using runtime for evaluation, however, has its disadvantages. There are various factors that can affect the integrity of timing experiments, such as the hardware platform, the load balancing on the machine where tests are being run, the choice of programming language, particular implementation details, and so on. While some of these factors can be accounted for by running multiple tests, it can be difficult to conduct experiments which completely isolate specific variables that are different between experiments. This is especially important because a significant part of designing BU-AKD is looking at the impact on performance of different behaviors. For this reason, in addition to runtimes we present results according to several new measures we developed, which help directly connect the performance of an algorithm to how it explores lattices.

The new measures are based on whether rules in the lattice are inferred or have their approximation value calculated (`NumInferred` and `NumCalculated`, respectively). Both measures provide an evaluation of performance independent of the hardware on which the algorithm is run, and the method for calculating the approximation measure. In order to have a meaningful comparison across experiments, it is necessary to normalize on a per-data set basis expressed as a fraction of the cost of BU-BFS:

$$\texttt{NormNumCalc} = \frac{\texttt{NumCalculated}\ of Algorithm}{\texttt{NumCalculated}\ of BUBFS}$$

BU-BFS is used for normalization because of its traditional use in AFD mining and because it is independent of *PickAttr*. `NormNumCalc` values less than one represent a savings compared to BU-BFS, while values greater than one represent worse performance. In subsequent figures, `NormNumCalc` values are presented broken down into three categories of rules: *pass*, *fail*, and all. This distinction helps to understand how overall performance correlates with costs on the *pass* and *fail* portions of the search space. The second statistic is `NormNumInfer`, which is the percentage of rules which have been inferred instead of having their approximation value calculated. `NormNumInfer` can also be broken down by occurring on the *pass*/*fail* sides of the lattice.

The final statistic we use is `avgSizeBoundaryCross`. It is calculated by averaging for each lattice the number of attributes on the LHS of the first *pass* rule found. The `avgSizeBoundaryCross` is used in examining how the path of a DFS algorithm correlates with its ability to infer instead of calculate *pass*/*fail* status.

Unless indicated, results are averages across all of the *PickAttr* plug-ins.

6.2 Data Sets and Test Variations

We performed experiments with a number of data sets from different repositories [8, 2] and present results for a sample of the data sets which are representative of the trends we see across all data sets. We also performed experiments with multiple threshold values. Again, for brevity we only include experiments with a .1 threshold value. In general, as the threshold moves closer to 0, the performance gains that we demonstrate are increased, and as the threshold moves towards 1, performance becomes homogeneous. These trends are caused by how the threshold impacts where *minP* rules are found in lattices. As a threshold moves towards 0, the number of attributes in *minP* rules on the LHS generally increases and as the threshold moves towards 1, the number of attributes in *minP* rules decreases. When all AFDs with a single attribute on the LHS are *minP*, all bottom-up algorithms will perform the same.

6.3 Overall Performance

We ran runtime experiments on a Quad Quad-Core (16 total cores) 2.93GHz Intel Xeon system running 64-bit Red Hat Enterprise Linux with 32GB of RAM. We note that this machine is shared and we could not control its workload while our experiments ran. To account for this, we ran runtime experiments several times to ensure that the patterns that emerged were consistent.

Runtime experiments were ran for all *PickAttr*s and ordering algorithms mentioned so far. Figure 4 shows the results on three of our data sets averaged over the *PickAttr* values. As seen from the figure, BU-AKD consistently performs superior to BU-BFS and all other DFS algorithms.

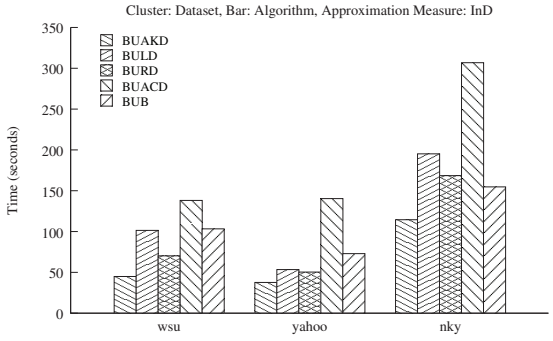

Figure 4: Runtime averaged over all *PickAttr*'s.

Figure 5 shows runtime results clustered by *PickAttr*s for the Windermere data set. We observed with all data sets the same pattern of BU-AKD's runtime performance always being better than BU-BFS's, and except for the small difference with the Key *PickAttr*, BU-AKD also performs better

than other DFS algorithms. Windermere is the most interesting data set to examine, because it is biggest in terms of the number of *minP* rules it has. Because of this, the search space that is considered is larger, so differences in performance are heightened. Even though Windermere's number of attributes is similar to that of the Yahoo and NKY data sets, the boundary crossing height is a more significant characteristic when it comes to the search process itself, and in this case shows the significant gain in efficiency that BU-DFS in general and BU-AKD in particular can bring.

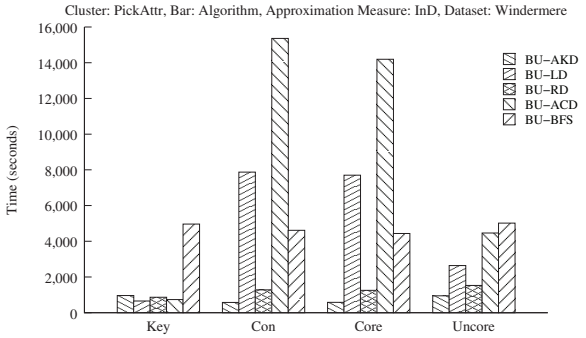

Figure 5: Runtime results on the Windermere data set for each PickAttr.

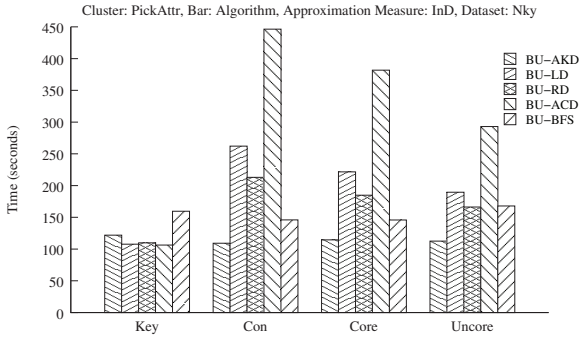

Figure 6: Runtime results on the Nky data set for each PickAttr.

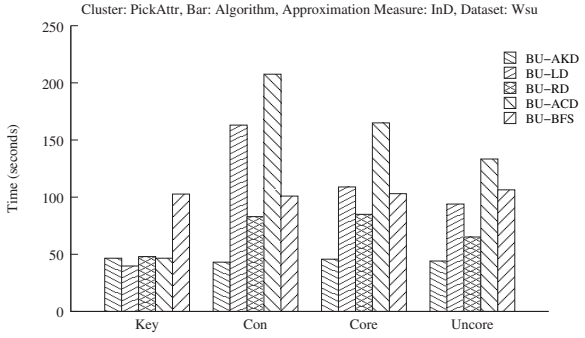

Figure 7: Runtime results on the WSU data set for each PickAttr.

The two conclusions that can be drawn from runtime experiments are that BU-AKD clearly outperforms BU-BFS

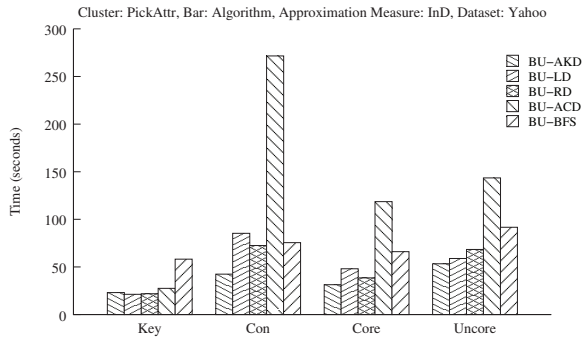

Figure 8: Runtime results on the Yahoo data set for each PickAttr.

and other DFS algorithms, and the performance of BU-AKD in absolute terms is good.

6.3.1 *Machine-independent Evaluation*

We also investigated the causes of BU-AKD's performance by looking at the performance measures we designed for this very purpose. Figure 9 presents the `NormNumCalc` broken down by *pass* and *fail* costs for each data set and DFS algorithm. The first and most notable point is that BU-AKD is the best performing algorithm for every data set. In addition, for most of the data sets BU-AKD is also significantly better than the second best performing algorithm. Another important point is that when Figure 9 is compared to Figure 10, a reverse correlation is observed in that as `NormNumInfer` increases, `NormNumCalc` decreases, representing a savings which is clearly attributable to the path which the AK heuristic produces.

We note an unexpected trend when comparing *fail* `NormNumCalc`s in Figure 9. Though values are relatively close to each other, BU-AKD is consistently observed to have larger *fail* `NormNumCalc` values than other algorithms. It was expected that similar to *fail* inferencing, *fail* calculations would be relatively consistent between algorithms. However, it seems clear that BU-AKD trades additional calculations on the *fail* side for savings on the *pass* side. Though this obviously relates to crossing the boundary lower, we did not investigate it further due to its small effect on the overall results.

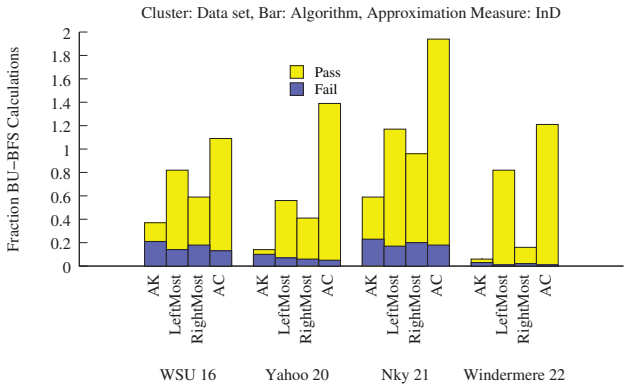

Figure 9: *pass*/*fail* `NormNumCalc` per algorithm, per dataset, averaged over *PickAttr*. Datasets ordered by number attributes.

6.4 Supporting Insights behind AK

We have seen that BU-AKD does well in performance tests, so we look at the evaluation measures we developed to shed light on the inner working of BU-AKD. In this section we provide evidence for each insight described in section 5 and how BU-AKD's performance can be attributed to it.

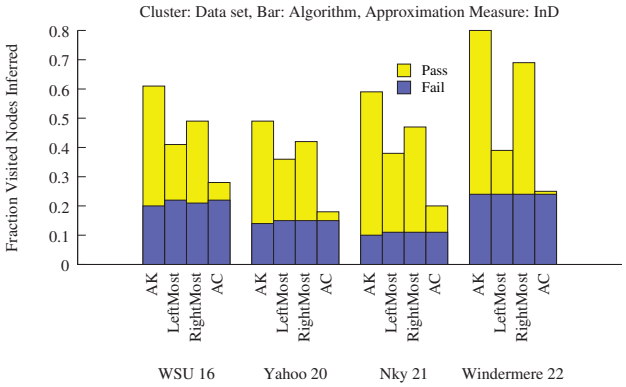

Figure 10: *pass*/*fail* `NormNumInferper` algorithm, per dataset, averaged over *PickAttr*. Data sets ordered by number attributes.

- *fail* inferencing varies little between DFS algorithms.

Figure 10 presents how well the DFS algorithms make use of inferencing broken down by whether inferred rules were *pass* or *fail* rules. The results support the idea that path selection functions have little effect on *fail* inferencing. Though some minor variations do occur, best seen in the WSU data set, the difference never reaches significant levels or alters which algorithm provides the most inferencing. Figure 10 also demonstrates the importance of *pass* inferencing because that is where significant differences in savings occur.

- Crossing the boundary low increases *pass* inferencing.

Figure 11 presents evidence of the correlation between using AK and crossing the threshold boundary lower in the search space. The results point to a clear correlation that the lower the threshold boundary is crossed, the better an algorithm does at *pass* inferencing.

These results create a clear chain of support showing that AK crosses the threshold boundary lower than other DFS algorithms thus increasing *pass* inferencing, and that an algorithm should focus on *pass* inferencing because it is where the differences in performance are much more significant than any small variations which can occur in *fail* inferencing.

- Selecting the next parent rule according to how close it is to being a key will result in crossing the threshold boundary lower.

Figure 12 presents the average number of LHS attributes for the rule at which each algorithm first crosses the threshold boundary. The graph clearly demonstrates that for every data set BU-AKD successfully crosses the boundary lower than other algorithms. This supports the basic premise motivating AK: by exploring attributes which are keys, a path is taken which lowers where the boundary is crossed. The

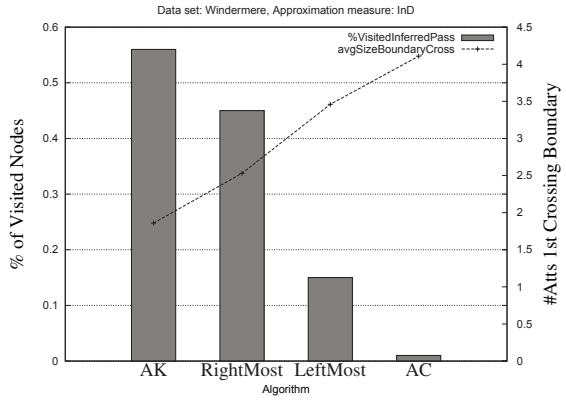

Figure 11: *pass* Inferencing vs. `avgSizeBoundaryCross`.

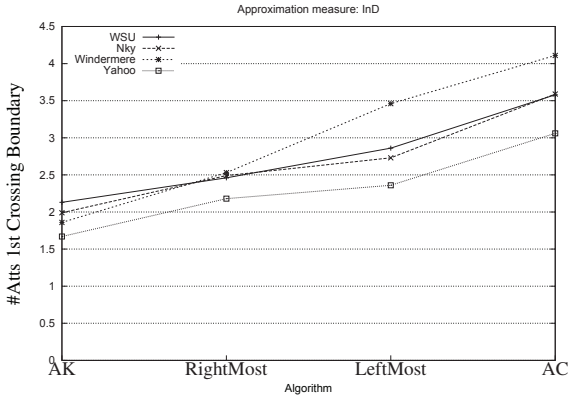

Figure 12: `avgSizeBoundaryCross` by DFS variant

validity of this hypothesis is further supported by the fact that AC, which is the exact opposite of AK, crosses the boundary higher than any other algorithm.

Finally, we note an unexpected result in Figure 12. The performance of BU-LD and BU-RD, both arbitrary orderings, seems to be following a trend. It is possible that this is due to conventions database designers follow when creating schemas, such as placing key columns together. Since this does not apply to BU-AKD and the other topics central to this paper, we do not analyze it further.

6.5 AK Consistency

Having demonstrated that BU-AKD exceeds expectations for improving performance we now explore its ability to do so in a consistent fashion. There are two factors for which we consider the impact on BU-AKD's performance; the approximation measure and the *PickAttr*. The *PickAttr* alters performance because it shapes the search space explored in each iteration. Though *PickAttr* is unique to MoLS, it is conceivable that in the vast array of data sets ones exist which mirror the effects of any *PickAttr*.

6.5.1 *Independence from PickAttr*

The *PickAttr* module determines the order in which attributes are iteratively added to the search process which also determines the order in which the search space is explored and to some extent what portions of the search space

get considered. Therefore, changing *PickAttr* modules can alter performance. The goal for BU-AKD is to provide consistent performance as the *PickAttr* module changes. We test five *PickAttr* plug-ins. The Key, Con, Core, and Uncore plug-ins have been previously discussed in [10]. The Random *PickAttr* was developed to test consistency and is the aggregated average of 10 randomly generated orders.

Figure 13 presents Windermere's NormNumCalc values for each combination of algorithm and *PickAttr*. The figure demonstrates that for every DFS algorithm besides BU-AKD, changing the *PickAttr* can radically alter performance. Some combinations are worse than BU-BFS by a factor of two and worse than BU-AKD by a factor of 20. The BU-AKD cluster, on the other hand, not only shows that BU-AKD performs significantly better, but, just as importantly, that it does not have the fluctuations in performance seen for other algorithms.

Figure 13 and Figure 5 both show an interesting trend when using the Key *PickAttr* plug-in. Both figures demonstrate that combined with the Key *PickAttr* plug-in, different BU-DFS algorithms result in little impact on performance. To explain this we first point out that the Key *PickAttr* does for adding attributes to the search space the same that the AK heuristic does for searching a lattice with BU-AKD. This causes arbitrary algorithms like BU-LD and BU-RD to behave similar to BU-AKD and therefore have similar performance. The Key *PickAttr* was developed specifically to improve performance, but because problem domain interests could make another *PickAttr* desirable, BU-AKD is needed to provide similar performance.

6.5.2 Independence from Approximation Measure

Comparing performance using different approximation measures goes directly to judge whether the hypotheses used to design BU-AKD actually represent underlying principles of AFD mining. Though there is not a large number of approximation measures for AFD mining, looking at results for both InD and g_3 strengthens the case that the benefits of using BU-AKD are broadly applicable. Practically, this is important because g_3 is the most frequently used approximation measure. Additionally, BU-AKD, which is focused on finding the threshold boundary, uses InD, so testing BU-AKD with g_3 as the approximation measure eliminates the case that insights for BU-AKD are only effective when the approximation measure is also InD.

Figures 14 and 15 present the min/max performance for each algorithm as the *PickAttr* varies. These graphs exclude tests using the Key *PickAttr* because, as previously seen, its impact on the search space results in little variation. Each of the 10 random orders, which were aggregated into Random results, are treated as separate tests, so the min/max are from among the 14 different orderings. The results for g_3 in Figure 15 show the same trends which have been discussed in this section. BU-AKD provides not only the best performance, but also consistent performance as demonstrated by the small differences seen between min/max performance. The similar trends in performance whether using InD or g_3 make the case that the benefits of using BU-AKD are broadly applicable for AFD mining.

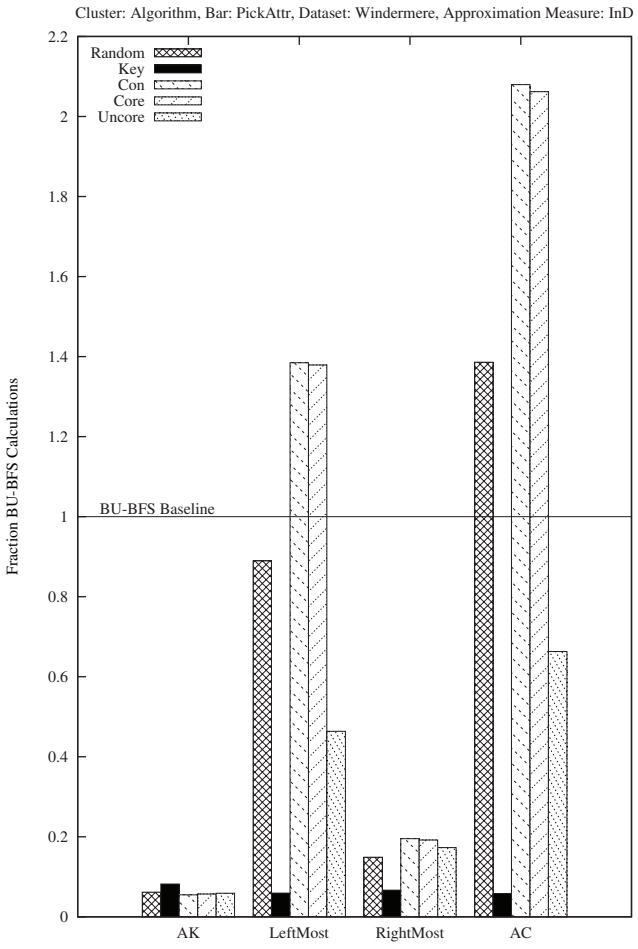

Figure 13: `NormNumCalc` demonstrating AK's independence from *PickAttr*

7. CONCLUSION

The goals in developing BU-AKD were for an algorithm which has similar performance to other DFS algorithms but is consistent as the search parameters are varied. These goals were necessary because the performance of arbitrary DFS algorithms such as BU-LD and BU-RD, while very good at times, could fluctuate by factors of ten or more in some experiments. With some parameter configurations, BU-LD and BU-RD could be twice as bad as BU-BFS. It was only if these wide fluctuations could be avoided that a DFS algorithm could be considered viable in practice.

In describing the AK heuristic, we outlined a set of insights which motivated its design. These insights formed a logical chain that lead to the ability of BU-AKD to perform consistently well. The cornerstone of our design was to use attribute keyness, the quality of how close an attribute is to being a key as measured by the enthropy of its values, to prioritize the order in which attributes are considered during the search process. Because enthropies could be precomputed, AK did not introduce additional computational costs compared to other algorithms.

In order to evaluate the insights that led to BU-AKD we performed tests while varying a number of parameters. One of the contributions of this paper is that we not only provide evidence of BU-AKD's overall performance, but also develop new measures of performance which provide more information than traditional runtime tests, and which quan-

titatively support the conclusion that it was the insights used to create BU-AKD that are responsible for its improved and consistent performance.

Beyond the savings that BU-AKD provides, we demonstrated that BU-AKD was consistently the best performing algorithm as data set and approximation measure were varied. Additionally, we provided evidence that unlike other DFS algorithms BU-AKD's performance is independent of *PickAttr* plug-ins. This makes a strong case for BU-AKD as the default algorithm choice in AFD mining.

7.1 Broader Impact

The methodology for developing and evaluating BU-AKD provides a contribution which could be applied in many domains. Designing BU-AKD represents an experimentally focused design process. The use of MoLS made BU-AKD possible by providing a testbed for the easy prototyping of algorithms. Because algorithms were quick to create, we were able to take a design approach of prototyping every idea for a path selection function and then looking at the outcomes. This process was only possible with performance measures which extend past runtime and allow an algorithm's behavior to be matched with conceptual goals. Performance of algorithms has traditionally been judged either theoretically through time complexity or practically through runtime. The performance measures we discuss provide insight not just on how well an algorithm performs, but how performance is being achieved. This leads to a better understanding of how algorithms behave in a broad range of scenarios and the tradeoffs they provide.

7.2 Future Work

The two tracks for future work on MoLS are optimizing the existing framework and adapting MoLS for other problem domains. The focus for optimizing MoLS is improving its management of stored information. The closest problem domains to AFD mining are frequent itemset, association rule, and CFD mining. All of these share similar search spaces with AFD mining so are natural transitions for the application of MoLS and Lozenge Search.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB Proceedings*, pages 487–499, 1994.

[2] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[3] A. Aussem, S. R. Morais, and M. Corbex. Nasopharyngeal carcinoma data analysis with a novel bayesian network skeleton learning algorithm. In *AIME '07: Proceedings of the 11th conference on Artificial Intelligence in Medicine*, pages 326–330, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] S. Bell and P. Brockhausen. Discovery of constraints and data dependencies in databases (extended abstract). In *European Conference on Machine Learning*, pages 267–270, 1995.

[5] D. G. Chris Giannella, Memhet Dalkilic and E. Robertson. Improving query evaluation with approximate functional dependency based decompositions. In *BNCOD*, pages 26–41, 2002.

[6] M. M. Dalkilic and E. L. Robertson. Information dependencies. In *PODS*, pages 245–253, 2000.

[7] N. Dexters, P. Purdom, and D. Van Gucht. Peak-jumping frequent itemset mining algorithms. In *PKDD*, 2006.

[8] A. Doan. Illinois semantic integration archive. http://pages.cs.wisc.edu/ anhai/wisc-si-archive/.

[9] J. T. Engle and E. L. Robertson. HLS: Tunable mining of approximate functional dependencies. In *BNCOD*, pages 28–39, 2008.

[10] J. T. Engle and E. L. Robertson. Depth first algorithms and inferencing for afd mining. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 54–65, 2009.

[11] J. T. Engle and E. L. Robertson. Modularizing data mining: A case study framework. Technical Report 678, Indiana University, 2009.

[12] B. Goethals. Survey on frequent pattern mining. Technical report, University of Helsinki, 2002.

[13] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD Proceedings*, pages 647–658, 2004.

[14] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. In *International Conference on Database Theory (ICDT)*, pages 129–149, 1995.

[15] V. Matos and B. Grasser. Sql-based discovery of exact and approximate functional dependencies. In *ITiCSE Working Group Reports*, pages 58–63, New York, NY, USA, 2004. Association for Computing Machinery.

[16] U. Nambiar and S. Kambhampati. Mining approximate functional dependencies and concept similarities to answer imprecise queries. In *WebDB Proceedings*, pages 73–78, New York, NY, USA, 2004. Association for Computing Machinery.

[17] R. J. M. Periklis Andritsos and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *SIGMOD Proceedings*, pages 731–742, 2004.

[18] F. G. X. J. Philip Bohannon, Wenfei Fan and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, pages 746–755, 2007.

[19] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2002.

[20] C. E. Shannon. A mathematical theory of communication. *Bell System Tech. J.*, 27:379–423, 623–656, 1948.

[21] J.-M. P. Stï£¡hane Lopes and L. Lakhal. Functional and approximate dependency mining: database and fca points of view. *Journal of Experimental and Theoretical Artificial Intelligence*, 14(2-3):93–114, 2002.

[22] H. Toivonen. Sampling large databases for association rules. pages 134–145. Morgan Kaufmann, 1996.

[23] S. S. M. J. Z. Vineet Chaoji, Mohammad Al Hasan. An integrated, generic approach to pattern mining: data mining template library. *Data Mining Knowledge Discovery*, pages 457–495, 2008.

[24] G. Wolf, H. Khatri, Y. Chen, and S. Kambhampati. Quic: A system for handling imprecision &

incompleteness in autonomous databases (demo). In *CIDR*, pages 263–268, 2007.

[25] G. Wolf, H. Khatri, B. Chokshi, J. Fan, Y. Chen, and S. Kambhampati. Query processing over incomplete autonomous databases. In *VLDB Proceedings*, pages 651–662. VLDB Endowment, 2007.

[26] P. P. Ykä Huhtala, Juha Kärkkäinen and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

APPENDIX

A. USING THE MoLS FRAMEWORK

The goal in developing MoLS[10, 11] was to create a framework whose skeleton handles the generic aspects of AFD mining, leaving many algorithmic details to customizable plug-ins. As part of developing MoLS, a number of default algorithm plug-ins were prototyped including BFS, left-most DFS, and right-most DFS, the latter two being based on the lexicographical order of the nodes. Included in the skeleton is functionality which manages and facilitates the use of inferred information. The framework uses Lozenge Search [9] as part of the skeleton to guide the search process at a high level. This appendix is an overview of MoLS to provide context for the development of BU-AKD.

The framework we used to develop BU-AKD is MoLS[10, 11], a customizable tool for mining, which provides a base solution that can be customized by users to the specific needs of a problem domain. There are three interesting features of MoLS: the use of Lozenge Search, an iterate-by-attribute template search algorithm to organize the search space, the ability to plug a lattice search algorithm into Lozenge Search, and MoLS's built in tools to manage inferencing. This appendix is an overview of MoLS to provide context for the development of BU-AKD.

A.1 Lozenge Search and *PickAttr*

Lozenge Search [9] is an iterate-by-attribute template search algorithm which organizes the search space into sets of sublattices. At each iteration a new attribute is added to the previously searched space. A lozenge is, then, the set of sublattices that result from adding the new attribute to the search space. There is a sublattice in a lozenge for each attribute with that attribute as the RHS. Thus, there will be as many sublattices in a lozenge, as there are possible RHSs, or alternatively, as there are attributes. A specific search algorithm is used to complete the Lozenge Search template and explore each sublattice. This conceptualization has the effect of creating an iterate-by-attribute algorithm.

The *PickAttr* plug-in determines the order in which attributes are selected to be added to the search space and, therefore, the order in which new rules are considered. Though this does not affect the overall results, it has the effect of altering what portions of the rules' approximation values are calculated, pruned or inferred. This feature of Lozenge Search is useful for enabling domain specific prioritizing because it allows the decision about which part of the search space to explore next to be delayed until runtime.

The impact of *PickAttr* plug-ins on performance was first explored in [9], where it was found that different *PickAttr* plug-ins led to dramatic changes in performance.

A.2 Inferencing Management

The mechanics of MoLS's management of inferencing is presented in [10, 11]. When MoLS learns the *pass*/*fail* status of a rule, either by calculating its approximateness or inferring it from stored information, that *pass*/*fail* status is propagated to one generation of ancestors/descendants, respectively. Propagating a generation at a time helps mitigate the combinatorial costs of storing information. The encapsulation of inferencing within MoLS separates the use of inferred knowledge from writing an algorithm. This allows algorithm development to benefit from the power of inferencing without understanding the mechanics of it.

A.3 Plug-in Algorithms

Algorithms for searching a lattice are implemented in MoLS using *Navigator* plug-ins. A part of the *Navigator* plug-in determines candidate rules to be generated and how these candidates are explored. Section 4 discusses examples of how the lattice exploration process works. The full requirements of a *Navigator* plug-in can be found in [11].

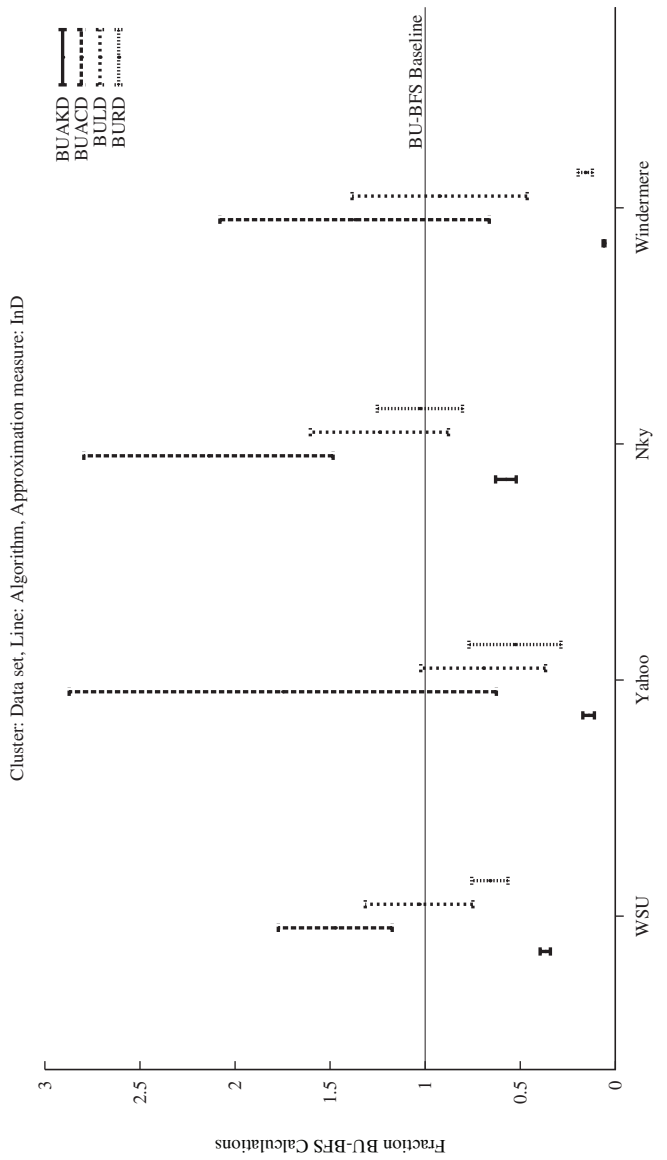

Figure 14: `NormNumCalc` range for an algorithm across *PickAttr*s (excluding Key)

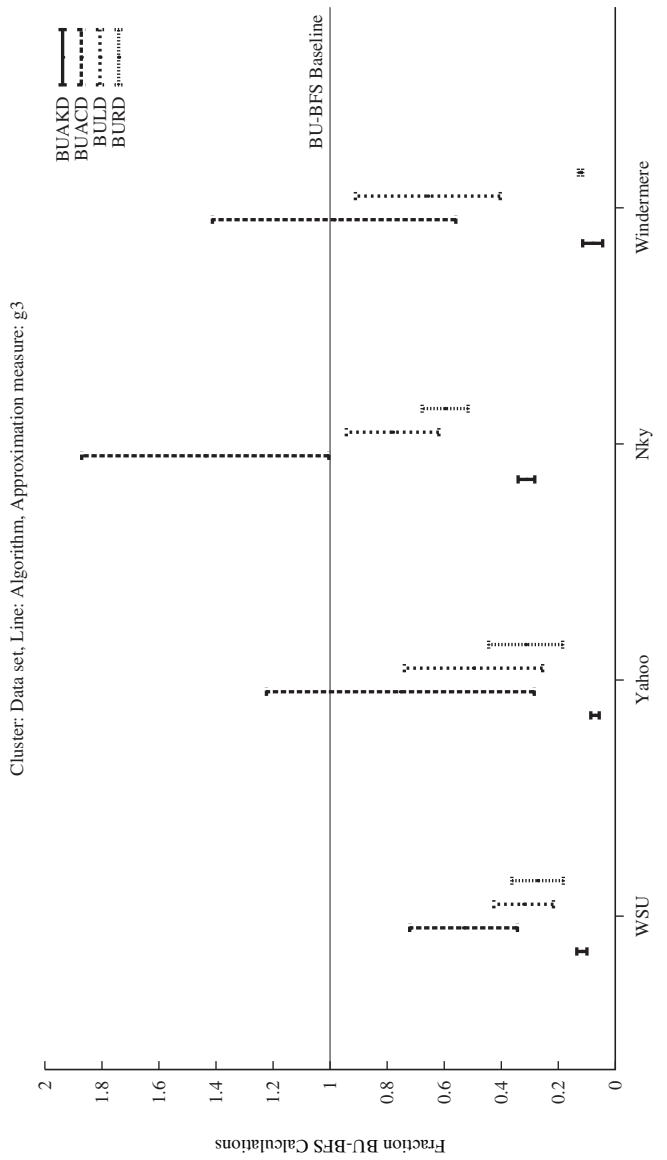

Figure 15: `NormNumCalc` range for an algorithm across *PickAttr*s (excluding Key)