Scalable Data Mining with Model Constraints

Minos Garofalakis
Bell Labs, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
minos@research.bell-labs.com

Rajeev Rastogi
Bell Labs, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
rastogi@research.bell-labs.com

ABSTRACT

Data mining can be abstractly defined as the process of extracting concise and interesting *models* (or, patterns) from large amounts of data. Unfortunately, conventional mining systems provide users with only very restricted mechanisms for specifying models of interest. As a consequence, the mining process is typically characterized by lack of focus and users often end up paying computational costs that are inordinately high compared to the specific models/patterns of interest. Exploiting user-defined *model constraints* during the mining process can help alleviate this problem and ensure system performance that is commensurate with the level of user focus. Attaining such performance goals, however, is not straightforward and, typically, requires the design of novel data mining algorithms that make effective use of the model constraints. In this paper, we provide an overview of our recent work on scalable, constraint-based algorithms for (1) *decision tree* construction with size and accuracy constraints for the desired decision tree model, and (2) *sequential pattern* extraction in the presence of structural, regular expression constraints for the target patterns. By "pushing" the model constraints inside the mining process, our algorithms give mining users exactly the models that they are looking for, while achieving performance speedups that often exceed one order of magnitude. Further, our work on sequential pattern mining has uncovered some valuable insights into the tradeoffs that arise when complex constraints that do not subscribe to "nice" properties (like anti-monotonicity) are integrated into the mining process. We argue that, in general, a *cost-based approach* (similar to that employed in conventional query optimizers) is needed to explore these tradeoffs in a principled manner and produce effective execution plans for ad-hoc mining queries.

Keywords

Data Mining, Constraints, Decision Trees, Sequential Patterns

1. INTRODUCTION

At a high level, all data mining techniques have the same goal, namely, that of efficiently extracting concise and interesting *models* from large amounts of data. These models can range from Bayesian networks [7; 8] and decision or regression trees [11; 17], to dense clusters [12] and frequent patterns [1; 20] in the data. This model-mining process is computation-intensive, typically requiring multiple passes over the input data. As a consequence, the design of effective mining algorithms has been the subject of intense research efforts in recent years.

A major common thread that runs across the vast majority of proposed mining algorithms is the lack of "knobs" that allow users to specify *constraints* on the models being mined; this, in turn, means that the mining process is typically characterized by *lack of user-controlled focus.* For example, the interaction of the user with a conventional pattern mining system is limited to specifying a lower bound on the desired support for the extracted patterns. The system then executes an appropriate mining algorithm and returns a very large number of sequential patterns, only some of which may be of actual interest to the user. As another example, decision-tree induction algorithms typically return a tree model for a specified class attribute that is "optimal" in an information-theoretic sense (e.g., based on Rissanen's Minimum Description Length (MDL) principle [14; 16]). Unfortunately, such decision tree structures can be extremely complex, comprising hundreds or thousands of nodes and, consequently, very difficult to comprehend and interpret. This is a serious problem and calls into question an often-cited benefit of decision trees, namely that they are easy to assimilate by humans.

Conventional, "unfocused" approaches to model mining suffer from two major drawbacks. First, they imply *disproportionate computational cost for "selective" users.* Despite the development of efficient algorithms, extracting models from a large database remains a computation-intensive task that can often take many hours to complete. Often, however, users are highly "selective" in the sense that they are only interested in models of a very specific form. For example, a user may only want to discover sequential patterns that conform to a very specific structure or very concise decision tree models that only provide a "rough picture" of the class attribute. Ignoring user focus can be extremely unfair to such selective users. Ideally, the computational cost of the mining process should be *commensurate* with the level of user focus (i.e., selective users should not be penalized for something that they did not ask for). Second, they result in an *overwhelming volume of potentially useless model rules or pat-*

terns. The lack of user-level focus during the mining process means that selective users will typically be swamped with a huge number of model rules or patterns, most of which are useless for their purposes. Sorting through a morass of patterns to find specific forms or trying to identify "strong" rules in a thousand-node decision tree can be a daunting task, even for the most experienced user.

In this paper, we provide an overview of our recent work on scalable, constraint-based algorithms for (1) *decision tree* construction with size and accuracy constraints for the desired decision tree model [9], and (2) *sequential pattern* extraction in the presence of structural, regular expression constraints for the target patterns [10]. Our contributions can be summarized as follows.

• **Constrained Decision Tree Induction [9].** We have developed novel algorithms that allow users to effectively trade accuracy for simplicity during the decision tree induction process. Our framework gives users the ability to specify constraints on either (1) the *size* (i.e., number of nodes); or, (2) the *inaccuracy* (i.e., MDL cost [14; 16; 17] or number of misclassified records) of the target classifier, and then exploits these constraints to *efficiently* construct the "best possible" decision tree.

More specifically, our main contribution lies in the introduction of novel decision tree induction algorithms that *integrate size and accuracy constraints into the tree-building phase*. Our algorithms employ *branch-and-bound* techniques to identify, during the growing of the decision tree, nodes that cannot possibly be part of the final constrained subtree. Since such nodes are guaranteed to be pruned when the user-specified size/accuracy constraints are enforced, our algorithms stop expanding such nodes early on. Furthermore, by only pruning nodes that are guaranteed not to belong to the optimal constrained subtree, we are assured that the final (sub)tree generated by our integrated approach is *exactly the same* as the subtree that would be generated by a naive approach that enforces the constraints only after the full tree is built. Determining, during the building phase, whether a node will be pruned by size or accuracy constraints is problematic, since the decision tree is only partially generated. To guarantee that only suboptimal parts of the tree are pruned, our branch-and-bound induction algorithms compute *lower bounds* on the inaccuracy (MDL cost or number of misclassifications) of the subtree rooted at a node (based on the corresponding set of training records).

• **Constrained Sequential Pattern Discovery [10].** We have formulated the problem of mining sequential patterns with *Regular Expression (RE) constraints* on the structure of the discovered patterns, and we have developed a novel family of algorithms (termed SPIRIT – Sequential Pattern mIning with Regular expressIon consTraints) for mining frequent sequential patterns that also belong to the language defined by the user-specified RE. Our algorithms exploit the equivalence of REs to deterministic finite automata [13] to push RE constraints deep inside the pattern mining computation. The main distinguishing factor among the proposed schemes is the *degree* to which the RE constraint is enforced within the generation and pruning of candidate patterns during the mining process. We observe that, varying the level of user focus (i.e., RE enforcement) during pattern mining gives rise to certain interesting tradeoffs with respect to computational effectiveness. Enforcing the RE constraint at each phase of the mining process certainly minimizes the amount of "state" maintained after each phase, focusing only on patterns that could potentially be in the final answer set. On the other hand, minimizing this maintained state may not always be the best solution since it can, for example, limit our ability to do effective support-based pruning in later phases. Such tradeoffs are obviously related to the fact that, in general, RE constraints are *not* anti-monotone [15]; thus, effectively integrating REs into Apriori-style mining is not straightforward. We believe that our results provide useful insights into the more general problem of constraint-driven, ad-hoc data mining, showing that there can be a whole spectrum of choices for dealing with constraints, even when they do not subscribe to nice properties like anti-monotonicity or succinctness [15].

For both constraint-based mining scenarios, our experimental results with both synthetic and real-life data sets have clearly validated the effectiveness of exploiting model constraints during the mining process, demonstrating that more than an order of magnitude improvement in performance is often possible. Further, our work on sequential pattern mining has uncovered some valuable insights into the tradeoffs that arise when complex model constraints are integrated into the mining process. We argue that, in general, a *cost-based approach* (similar to that employed in conventional query optimizers) is needed to explore these tradeoffs in a principled manner and produce effective execution plans for ad-hoc mining queries.

2. DECISION TREE INDUCTION WITH SIZE AND ACCURACY CONSTRAINTS

2.1 Problem Formulation

Preliminaries. We begin by presenting a brief overview of the building and pruning phases of a traditional decision tree classifier. More detailed descriptions of existing decision tree induction algorithms can be found in [4; 17; 19].

The overall algorithm for growing a decision tree classifier is depicted in Figure 1(a). Basically, the tree is built breadth-first by recursively partitioning the data until each partition is *pure* (i.e., it only contains records belonging to the same class). The splitting condition for each internal node of the tree is selected so that it minimizes an *impurity function*, such as the *entropy*, of the induced data partitioning [4].

To prevent overfitting of the training data, the MDL principle [14; 16] is applied to prune the tree built in the growing phase and make it more general. Briefly, the MDL principle states that the "best" tree is the one that can be encoded using the smallest number of bits. The cost of encoding the tree comprises three distinct components: (1) the cost of encoding the structure of the tree (single bit); (2) the cost of encoding for each split, the attribute and the value for the split ($C_{split}(N)$ is used to denote the cost of encoding the split at node N); and, (3) the cost of encoding the classes

```
Procedure BuildTree(S)
begin
1.  Initialize root node using data set S
2.  Initialize queue Q to contain root node
3.  while Q is not empty do {
4.     dequeue the first node N in Q
5.     if node N is not pure {
6.        for each attribute A
7.           Evaluate splits on attribute A
8.        Use best split to split N into N_1 and N_2
9.        Append N_1 and N_2 to Q
10.    }
11. }
end
```

(a)

```
Procedure PruneTree(Node N)
begin
1.  if N is a leaf return $(C(S) + 1)$
2.  minCost_1 := PruneTree(N_1);
3.  minCost_2 := PruneTree(N_2);
4.  minCost_N := min{$C(S) + 1$ ,
           $C_{split}(N) + 1 +$ minCost_1 + minCost_2};
5.  if minCost_N = $C(S) + 1$
6.     prune child nodes N_1 and N_2 from tree
7.  return minCost_N
end
```

(b)

Figure 1: (a) Tree-building algorithm. (b) Tree-pruning algorithm.

of data records in each leaf of the tree ($C(S)$ is the cost of encoding the classes for records in set S). In the rest of this section, we refer to the cost of encoding a tree computed above as the *MDL cost* of the tree. Also, for a node N of the tree, we use S to denote the set of records in N, and N_1 and N_2 to refer to the children of N.

The goal of the pruning phase is to compute the minimum MDL-cost subtree of the tree T constructed in the building phase. Briefly, this is achieved by traversing T in a bottom-up fashion, pruning all descendents of a node N if the cost of the minimum-cost subtree rooted at N is greater than or equal to $C(S) + 1$ (i.e., the cost of directly encoding the records corresponding to N). The cost of the minimum-cost subtree rooted at N is computed recursively as the sum of the cost of encoding the split and structure information at N ($C_{split}(N) + 1$) and the costs of the cheapest subtrees rooted at its two children. Figure 1(b) gives the pseudocode for the pruning procedure that computes the subtree of T with minimum MDL cost; more details can be found in [17].

Problem Statement. Let T be the *complete* tree constructed during the conventional tree-building phase. Our goal is to develop efficient algorithms for computing the minimum MDL-cost subtree of T in the presence of size constraints. More specifically, our problem can be stated as follows: "For a given k, compute the subtree T_f of T comprising *at most* k nodes and has the minimum possible MDL cost." (The treatment of accuracy constraints can be found in the full version of [9].)

2.2 Pushing Constraints into Tree-Building

Bohanec and Bratko [3] and Almuallim [2] present dynamic programming algorithms for computing the minimum cost subtree that satisfies the size constraint. However, the proposed dynamic programming algorithms enforce the user-specified size/accuracy constraints *only after a full decision tree has been grown by the building algorithm.* As a consequence, substantial effort (both I/O and CPU computation) may be wasted on growing portions of the tree that are subsequently pruned when constraints are enforced. Clearly, by "pushing" size and accuracy constraints into the tree-building phase, significant gains in performance can be attained. In this section, we present such *integrated* decision tree induction algorithms that integrate the constraint-enforcement phase into the tree-building phase instead of performing them one after the other.

Our integrated algorithms are similar to the BuildTree procedure depicted in Figure 1(a). The only difference is that periodically or after a certain number of nodes are split (this is a user-defined parameter), the partially built tree T_p is pruned using the user-specified size/accuracy constraints. Note, however, the pruning algorithm in Figure 1(b) cannot be used to prune the partial tree.

The problem with applying constraint-based pruning before the full tree has been built is that, in procedure PruneTree (Figure 1(b)), the MDL cost of the cheapest subtree rooted at a leaf N is assumed to be $C(S) + 1$ (Step 1). While this is true for the fully-grown tree, it is not true for a partially-built tree, since a leaf in a partial tree may be split later thus becoming an internal node. Obviously, splitting node N could result in a subtree rooted at N with cost much less than $C(S) + 1$. Thus, $C(S) + 1$ may overestimate the MDL cost of the cheapest subtree rooted at N and this could resulting in over-pruning; that is, nodes may be pruned during the building phase that are actually part of the optimal size- or accuracy-constrained subtree. This is undesirable since the final tree may no longer be the optimal subtree that satisfies the user-specified constraints.

In order to perform constraint-based pruning on a partial tree T_p, and still ensure that only suboptimal nodes are pruned, we adopt an approach that is based on the following observation. (For concreteness, our discussion is based on the case of size constraints.) Suppose U is the cost of the cheapest subtree of size at most k of the partial tree T_p. Note that this subtree may not be the final optimal subtree, since expanding a node in T_p could cause its cost to reduce by a substantial amount, in which case, the node along with its children may be included in the final subtree. U does, however, represent an upper bound on the cost of the final optimal subtree T_f. Now, if we could also compute *lower bounds* on the cost of subtrees of various sizes rooted at nodes of T_p, then we could use these lower bounds to determine the nodes N in T_p such that every potential subtree of size at most k (of the full tree T) containing N is guar-

```
Procedure ComputeCostUsingConst(Node $N$, integer $l$)
begin
1.  if Tree[$N$, $l$].computed = true
2.      return [Tree[$N$, $l$].realCost, Tree[$N$, $l$].lowCost]
3.  else if $l < 3$ or $N$ is a "pruned" or "not expandable" leaf
4.      Tree[$N$, $l$].realCost := Tree[$N$, $l$].lowCost := $C(S) + 1$
5.  else if $N$ is a "yet to be expanded" leaf {
6.      Tree[$N$, $l$].realCost := $C(S) + 1$
7.      Tree[$N$, $l$].lowCost := lower bound on cost of subtree
            cost rooted at $N$ with at most $l$ nodes
8.  } else {
9.      Tree[$N$, $l$].lowCost := Tree[$N$, $l$].realCost := $C(S) + 1$
10.     for $k_1$ := 1 to $l - 2$ do {
11.         $k_2 := l - k_1 - 1$
12.         [realCost$_1$, lowCost$_1$] :=
                    ComputeCostUsingConst($N_1$, $k_1$)
13.         [realCost$_2$, lowCost$_2$] :=
                    ComputeCostUsingConst($N_2$, $k_2$)
14.         if realCost$_1$ + $C_{split}(N) + 1$ + realCost$_2$ <
                        Tree[$N$, $l$].realCost
15.             Tree[$N$, $l$].realCost := realCost$_1$ + $C_{split}(N)$+
                        1 + realCost$_2$
16.         if lowCost$_1$ + $C_{split}(N) + 1$ + lowCost$_2$ <
                        Tree[$N$, $l$].lowCost
17.             Tree[$N$, $l$].lowCost := lowCost$_1$ + $C_{split}(N)$+
                        1 + lowCost$_2$
18.     }
19. }
20. Tree[$N$, $l$].computed := true
21. return [Tree[$N$, $l$].realCost, Tree[$N$, $l$].lowCost]
end
```

Figure 2: Computing minimum MDL-cost subtrees using lower bounds.

```
Procedure PruneUsingConst(Node $N$, integer $l$, real $B$)
begin
1.  Mark node $N$
2.  if $B \leq$ Bound[$N$, $l$] return
3.  for $i$ := 1 to $l$ do
4.      if $B >$ Bound[$N$, $i$]
5.          Bound[$N$, $i$] := $B$
6.  if Tree[$N$, $l$].lowCost $> B$ or
                    Tree[$N$, $l$].lowCost = $C(S) + 1$
7.      return
8.  else if $N$ is not a leaf node and $l \geq 3$ {
9.      for $k_1$ := 1 to $l - 2$ do {
10.         $k_2 := l - k_1 - 1$
11.         if $C_{split}(N) + 1+$ Tree[$N_1$, $k_1$].lowCost +
                    Tree[$N_2$, $k_2$].lowCost $\leq B$ {
12.             $B_1 := B - (C_{split}(N) + 1) -$ Tree[$N_2$, $k_2$].lowCost
13.             $B_2 := B - (C_{split}(N) + 1) -$ Tree[$N_1$, $k_1$].lowCost
14.             PruneUsingConst($N_1$, $k_1$, $B_1$);
15.             PruneUsingConst($N_2$, $k_2$, $B_2$);
16.         }
17.     }
18. }
end
```

Figure 3: Branch-and-bound pruning algorithm.

anteed to have a cost greater than U. Clearly, such nodes can be safely pruned from T_p, since they cannot possibly be part of the optimal subtree whose cost is definitely less than or equal to U.

While it is relatively straightforward to compute U, we still need to (1) estimate the lower bounds on cost at each node of the partial tree T_p, and (2) show how these lower bounds can be combined with the upper bound U (in a "branch-and-bound" fashion) to identify prunable nodes of T_p.

Computing Lower Bounds on Subtree Costs. To obtain lower bounds on the MDL cost of a subtree at arbitrary nodes of T_p, we first need to be able to compute lower bounds for subtree costs at leaf nodes that are "yet to be expanded". These bounds can then be propagated "upwards" to obtain lower bounds for other nodes of T_p. Obviously, any subtree rooted at node N must have an MDL cost of at least 1, and thus 1 is a simple, but conservative estimate for the MDL cost of the cheapest subtree at leaf nodes that are "yet to be expanded". In our earlier work [17], we have derived more accurate lower bounds on the MDL cost of subtrees by also considering split costs.

Computing an Optimal Size-Constrained Subtree. As described earlier, our integrated constraint-pushing strategy involves the following three steps, which we now describe in more detail: (1) compute the cost of the cheapest subtree of size (at most) k of the partial tree T_p (this is an upper bound U on the cost of the final optimal tree T_f); (2) compute lower bounds on the cost of subtrees of varying sizes that are rooted at nodes of the partial tree T_p; and, (3) use the bounds computed in steps (1) and (2) to identify and prune nodes that cannot possibly belong to the optimal constrained subtree T_f. Procedure ComputeCostUsingConst (depicted in Figure 2) accomplishes the first two steps, while procedure PruneUsingConst (depicted in Figure 3) achieves step (3).

Procedure ComputeCostUsingConst distinguishes among three classes of leaf nodes in the partial tree. The first class includes leaf nodes that still need to be expanded ("yet to be expanded"). The two other classes consist of leaf nodes that are either the result of a pruning operation ("pruned") or cannot be expanded any further because they are pure ("not expandable"). ComputeCostUsingConst uses dynamic programming to compute in Tree[N, l].realCost the MDL cost of the cheapest subtree of size at most l that is rooted at N in the partially-built tree. In addition, ComputeCostUsingConst also computes in Tree[N, l].lowCost a lower bound on the MDL cost of the cheapest subtree with size at most l that is rooted at N (if the partial tree were expanded fully) – the lower bounds on the MDL cost of subtrees rooted at "yet to be expanded" leaf nodes are used for this purpose. The only difference between the computation of the real costs and the lower bounds is that, for a "yet to be expanded" leaf node N, the former uses $C(S) + 1$ while the latter uses the lower bound for the minimum MDL-cost subtree rooted at N. Procedure ComputeCostUsingConst is invoked with input parameters R and k, where R is the root of T_p and k is the constraint on the number of nodes. Again, note that U = Tree[R, k].realCost represents an upper bound on the cost of the final optimal subtree satisfying the user-specified constraints.

Once the real costs and lower bounds are computed, the next step is to identify *prunable* nodes N in T_p and prune them. A node N in T_p is prunable if every potential subtree of size at most k (after "yet to be expanded leaves" in T_p are expanded) that contains node N is guaranteed to have an MDL cost greater than Tree[R, k].realCost. Invoking procedure PruneUsingConst(illustrated in Figure 3) with input parameters R (root node of T_p), k, and Tree[R, k].realCost (upper bound on the cost of T_f) ensures that *every non-prunable* node in T_p is *marked*. Thus, after PruneUsingConst completes execution, it is safe to prune all unmarked nodes from T_p, since these cannot possibly be in the MDL-optimal subtree T_f with size at most k.

Intuitively, procedure PruneUsingConst works by using the computed lower bounds at nodes of T_p in order to "propagate" the upper bound (Tree[R, k].realCost) on the cost of T_f down the partial tree T_p (Steps 12–15). Assume that some node N (with children N_1 and N_2) is reached with a "size budget" of l and a cost bound of B. If there exists some distribution of l among N_1 and N_2 such that the sum of the corresponding lower bounds does not exceed B (Steps 9–11), then N_1 and N_2 may belong the optimal subtree and PruneUsingConst is invoked recursively (Steps 12–15) to (a) mark N_1 and N_2 (Step 1), and (b) search for nodes that need to be marked in the corresponding subtrees. Thus, nodes N_1 and N_2 will be left unmarked if and only if, for every possible size budget that reached N, no combination was ever found that could beat the corresponding upper bound B.

More formally, consider a node N' in the subtree of T_p rooted at N_1 and let l and B denote the size budget and cost upper bound propagated down to N (parent of N_1 and N_2). We say that N' is *prunable with respect to* (N, l, B) if every potential subtree of size at most l (after T_p is fully expanded) that is rooted at N and contains N', has an MDL cost greater than B. PruneUsingConst is based on the following key observation: If N' is *not prunable* with respect to (N, l, B), then, for some $1 \leq k_1 \leq l-2$,

1. $C_{split}(N) + 1+$ Tree[N_1, k_1].lowCost $+$ Tree[N_2, $l - k_1 - 1$].lowCost $\leq B$, and
2. N' is not prunable with respect to (N_1, k_1, $B - (C_{split}(N) + 1) -$ Tree[N_2, $l - k_1 - 1$].lowCost).

That is, if N' is not prunable with respect to (N, l, B) then there exists a way to distribute the size budget l along the path from N down to N' such that the lower bounds on the MDL cost never exceed the corresponding upper bounds, on all the nodes in the path. Obviously, N' is not prunable (i.e., should be marked) if it is not prunable with respect to *some* triple (N, l, B). Based on these observations, we can formally prove that if a node in T_p is not prunable, then it is marked by procedure PruneUsingConst. (The proof can be found in the full version of [9].)

As an optimization, procedure PruneUsingConst maintains the array Bound[] in order to reduce computational overheads. Each entry Bound[N, l] is initialized to 0 and is used to keep track of the maximum value of B with which PruneUsingConst has been invoked on node N with size budget $l' \geq l$. The key observation here is that if a node N' in the subtree rooted at N is not prunable with respect to (N, l, B), then it is also not prunable with respect to (N, l', B'), for all $B' \geq B$, $l' \geq l$. Intuitively, this says that if we have already reached node N with a cost bound B' and size budget l', then invoking PruneUsingConst on N with a smaller bound $B \leq B'$ and smaller size budget $l \leq l'$ cannot cause any more nodes under N to be marked. Thus, when such a situation is detected, our marking procedure can simply return (Step 2).

Overview of Experimental Results. To investigate the performance gains that can be realized as a result of exploiting size and accuracy constraints, we have conducted an extensive experimental study on real-life as well as synthetic data sets. Our experimental results with both types of data sets clearly demonstrate the effectiveness of integrating the user-specified constraints into the tree-building phase. Our constraint-pushing algorithms always result in significant reductions in execution times that are sometimes as high as two or three orders of magnitude. For the complete details, the interested reader is referred to the full version of [9].

3. SEQUENTIAL PATTERN DISCOVERY WITH RE CONSTRAINTS

3.1 Problem Formulation

Preliminaries. The main input to our pattern-mining problem is a database of sequences, where each sequence is an ordered list of *elements*. These elements can be either (a) *simple items* from a fixed set of literals (e.g., the identifiers of WWW documents available at a server [6], the amino acid symbols used in protein analysis [21]), or (b) *itemsets*, that is, non-empty sets of items (e.g., books bought by a customer in the same transaction [20]). The list of elements of a data sequence s is denoted by $< s_1\ s_2 \cdots s_n >$, where s_i is the i^{th} element of s. We use $|s|$ to denote the *length* (i.e., number of elements) of sequence s. A sequence of length k is referred to as a *k-sequence.*

Consider two data sequences $s = < s_1\ s_2\ \cdots\ s_n >$ and $t = < t_1\ t_2\ \cdots\ t_m >$. We say that s is a *subsequence* of t if s is a "projection" of t, derived by deleting elements and/or items from t. More formally, s is a subsequence of t if there exist integers $j_1 < j_2 < \ldots < j_n$ such that $s_1 \subseteq t_{j_1}$, $s_2 \subseteq t_{j_2}$, ..., $s_n \subseteq t_{j_n}$. Note that for sequences of simple items the above condition translates to $s_1 = t_{j_1}$, $s_2 = t_{j_2}$, ..., $s_n = t_{j_n}$. For example, sequences $< 1\ 3 >$ and $< 1\ 2\ 4 >$ are subsequences of $< 1\ 2\ 3\ 4 >$, while $< 3\ 1 >$ is not. A sequence s is said to *contain* a sequence p if p is a subsequence of s. We define the *support* of a pattern p as the fraction of sequences in the input database that contain p. Given a set of sequences $\mathcal{S}$, we say that $s \in \mathcal{S}$ is *maximal* if there are no sequences in $\mathcal{S} - \{s\}$ that contain it.

A RE constraint $\mathcal{R}$ is specified as a RE over the alphabet of sequence elements using the established set of RE operators, such as disjunction ($|$) and Kleene closure (*) [13]. Thus, a RE constraint $\mathcal{R}$ specifies a language of strings over the element alphabet or, equivalently, a regular family of sequential patterns that is of interest to the user. A well-known result from complexity theory states that REs have exactly the same expressive power as *deterministic finite automata* [13]. Thus, given any RE $\mathcal{R}$, we can always build a deterministic finite automaton $\mathcal{A}_{\mathcal{R}}$ such that $\mathcal{A}_{\mathcal{R}}$ accepts exactly the language generated by $\mathcal{R}$. Informally, a deterministic finite automaton is a finite state machine with (a) a well-defined *start* state (denoted by a) and one or more *accept* states, and (b) deterministic transitions across states on symbols of the input alphabet (in our case, sequence elements). A transition from state b to state c on element s_i is denoted by $b \xrightarrow{s_i} c$. We also use the shorthand $b \overset{s}{\Rightarrow} c$ to denote the sequence of transitions on the elements of sequence s starting at state b and ending in state c. A sequence s is *accepted by* $\mathcal{A}_{\mathcal{R}}$ if following the sequence of transitions for the elements of s from the start state results in an accept state. Figure 4 depicts the state diagram of a deterministic finite automaton for the RE 1^* (2 2 | 2 3 4 | 4 4) (i.e., all sequences of zero or more 1's followed by 2 2, 2 3 4, or 4 4). Following [13], we use double circles to indicate an accept state and $>$ to emphasize the start state (a) of the automaton.

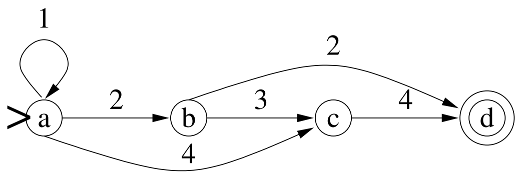

Figure 4: Automaton for the RE 1^* (2 2 | 2 3 4 | 4 4).

Problem Statement. Given an input database of sequences, we define a sequential pattern to be *frequent* if its support in the database exceeds a user-specified minimum support threshold. In our work [10], we have proposed novel, efficient algorithms for mining frequent sequential patterns in the presence of user-specified RE constraints. (We focus on sequences of simple items with no max-distance bounds for pattern occurrences; the modifications necessary to handle itemset sequences and distance bounds are described in the full version of [10].) The following definitions establish some useful terminology for our discussion.

Definition 3.1. A sequence s is said to be *legal with respect to state* b of automaton $\mathcal{A}_{\mathcal{R}}$ if every state transition in $\mathcal{A}_{\mathcal{R}}$ is defined when following the sequence of transitions for the elements of s from b.

Definition 3.2. A sequence s is said to be *valid with respect to state* b of automaton $\mathcal{A}_{\mathcal{R}}$ if s is legal with respect to b *and* the final state of the transition path from b on input s is an *accept* state of $\mathcal{A}_{\mathcal{R}}$. We say that s is *valid* if s is valid with respect to the start state a of $\mathcal{A}_{\mathcal{R}}$ (or, equivalently, if s is accepted by $\mathcal{A}_{\mathcal{R}}$).

Example 3.1. Consider the RE $\mathcal{R} = 1^*$ (2 2 | 2 3 4 | 4 4) and the automaton $\mathcal{A}_{\mathcal{R}}$, shown in Figure 4. Sequence $< 1\ 2\ 3 >$ is legal with respect to state a and sequence $< 3\ 4 >$ is legal with respect to state b, while sequences $< 1\ 3\ 4 >$ and $< 2\ 4 >$ are not legal with respect to any state of $\mathcal{A}_{\mathcal{R}}$. Similarly, sequence $< 3\ 4 >$ is valid with respect to state b (since $b \overset{<3\ 4>}{\Longrightarrow} d$ and d is an accept state), however it is not valid, since it is not valid with respect to the start state a of $\mathcal{A}_{\mathcal{R}}$. Examples of valid sequences include $< 1\ 1\ 2\ 2 >$ and $< 2\ 3\ 4 >$.

Having established the necessary notions and terminology, we can now provide an abstract definition of our constrained pattern mining problem as follows.

- **Given:** A database of sequences $\mathcal{D}$, a user-specified minimum support threshold, and a user-specified RE constraint $\mathcal{R}$ (or, equivalently, an automaton $\mathcal{A}_{\mathcal{R}}$).
- **Find:** All *frequent and valid* sequential patterns in $\mathcal{D}$.

Thus, our objective is to efficiently mine patterns that are not only frequent but also belong to the language of sequences generated by the RE $\mathcal{R}$. To this end, we have proposed the SPIRIT family of mining algorithms for pushing user-specified RE constraints to varying degrees inside the pattern mining process [10].

3.2 The SPIRIT Family of Algorithms

Overview. Figure 5 depicts the basic algorithmic skeleton of the SPIRIT family, using an input parameter $\mathcal{C}$ to denote a generic user-specified constraint on the mined patterns. The output of a SPIRIT algorithm is the set of frequent sequences in the database $\mathcal{D}$ that satisfy constraint $\mathcal{C}$. At a high level, our algorithmic framework is similar in structure to the general Apriori strategy of Agrawal and Srikant [1]. Basically, SPIRIT algorithms work in passes, with each pass resulting in the discovery of longer patterns. In the k^{th} pass, a set of candidate (i.e., potentially frequent and valid) k-sequences C_k is generated and pruned using information from earlier passes. A scan over the data is then made, during which the support for each candidate sequence in C_k is counted and F_k is populated with the frequent k-sequences in C_k. There are, however, two crucial differences between the SPIRIT framework and conventional Apriori-type schemes (like GSP [20]) or the Constrained APriori (CAP) algorithm [15] for mining associations with anti-monotone and/or succinct constraints.

1. *Relaxing $\mathcal{C}$ by inducing a weaker (i.e., less restrictive) constraint $\mathcal{C}'$ (Step 1).* Intuitively, constraint $\mathcal{C}'$ is *weaker* than $\mathcal{C}$ if every sequence that satisfies $\mathcal{C}$ also satisfies $\mathcal{C}'$. The "strength" of $\mathcal{C}'$ (i.e., how closely it emulates $\mathcal{C}$) essentially determines the degree to which the user-specified constraint $\mathcal{C}$ is pushed inside the pattern mining computation. The choice of $\mathcal{C}'$ differentiates among the members of the SPIRIT family and leads to interesting tradeoffs that are discussed in detail later in this section.

2. *Using the relaxed constraint $\mathcal{C}'$ in the candidate generation and candidate pruning phases of each pass.* SPIRIT algorithms maintain the set F of frequent sequences (up to a given length) that satisfy the relaxed constraint $\mathcal{C}'$. Both F and $\mathcal{C}'$ are used in:

(a) the candidate generation phase of pass k (Step 6), to produce an initial set of candidate k-sequences C_k that satisfy $\mathcal{C}'$ by appropriately extending or combining sequences in F; and,

(b) the candidate pruning phase of pass k (Steps 8-9), to delete from C_k all candidate k-sequences containing at least one subsequence that satisfies $\mathcal{C}'$ and does not appear in F.

Thus, a SPIRIT algorithm maintains the following *invariant*: at the end of pass k, F_k is exactly the set of all frequent k-sequences that satisfy the constraint $\mathcal{C}'$. Note that incorporating $\mathcal{C}'$ in candidate generation and pruning also impacts the terminating condition for the **repeat** loop in Step 15. Finally, since at the end of the loop, F contains frequent patterns satisfying the induced relaxed constraint $\mathcal{C}'$, an additional filtering step may be required (Step 17).

```
Procedure SPIRIT($\mathcal{D}$ , $\mathcal{C}$)
begin
1.  let $\mathcal{C}'$ := a constraint weaker (i.e., less restrictive) than $\mathcal{C}$
2.  $F := F_1 :=$ frequent items in $\mathcal{D}$ that satisfy $\mathcal{C}'$
3.  $k := 2$
4.  repeat {
5.     // candidate generation
6.     using $\mathcal{C}'$ and $F$ generate $C_k := \{$ potentially frequent
       $k$-sequences that satisfy $\mathcal{C}'$ $\}$
7.     // candidate pruning
8.     let $P := \{s \in C_k : s$ has a subsequence $t$ that satisfies
       $\mathcal{C}'$ and $t \notin F \}$
9.     $C_k := C_k - P$
10.    // candidate counting
11.    scan $\mathcal{D}$ counting support for candidate sequences in $C_k$
12.    $F_k :=$ frequent sequences in $C_k$
13.    $F := F \cup F_k$
14.    $k := k + 1$
15. } until TerminatingCondition($F$ , $\mathcal{C}'$) holds
16. // enforce the original (stronger) constraint $\mathcal{C}$
17. output sequences in $F$ that satisfy $\mathcal{C}$
end
```

Figure 5: SPIRIT constrained pattern mining framework.

Given a set of candidate k-sequences C_k, counting support for the members of C_k (Step 11) can be performed efficiently by employing specialized search structures, like the *hash tree* [20], for organizing the candidates. The implementation details can be found in [20]. The candidate counting step is typically the most expensive step of the pattern mining process and its overhead is directly proportional to the size of C_k [20]. Thus, at an abstract level, the goal of an efficient pattern mining strategy is to employ the minimum support requirement and any additional user-specified constraints to restrict as much as possible the set of candidate k-sequences counted during pass k. The SPIRIT framework strives to achieve this goal by using two different types of pruning within each pass k.

- *Constraint-based pruning* using a relaxation $\mathcal{C}'$ of the user-specified constraint $\mathcal{C}$; that is, ensuring that all candidate k-sequences in C_k satisfy $\mathcal{C}'$. This is accomplished by appropriately employing $\mathcal{C}'$ and F in the candidate generation phase (Step 6).
- *Support-based pruning*; that is, ensuring that all subsequences of a sequence s in C_k that satisfy $\mathcal{C}'$ are present in the current set of discovered frequent sequences F (Steps 8-9). Note that, even though *all* subsequences of s must in fact be frequent, we can only check the minimum support constraint for subsequences that satisfy $\mathcal{C}'$, since only these are retained in F.

Intuitively, constraint-based pruning tries to restrict C_k by (partially) enforcing the input constraint $\mathcal{C}$, whereas support-based pruning tries to restrict C_k by checking the minimum support constraint for qualifying subsequences. Note that, given a set of candidates C_k and a relaxation $\mathcal{C}'$ of $\mathcal{C}$, the amount of support-based pruning is maximized when $\mathcal{C}'$ is *anti-monotone* [15] (i.e., all subsequences of a sequence satisfying $\mathcal{C}'$ are guaranteed to also satisfy $\mathcal{C}'$). This is because support information for *all* of the subsequences of a candidate sequence s in C_k can be used to prune it. However, when $\mathcal{C}'$ is *not* anti-monotone, the amounts of constraint-based and support-based pruning achieved vary depending on the specific choice of $\mathcal{C}'$.

Pushing Non Anti-Monotone Constraints. Consider the general problem of mining all frequent sequences that satisfy a user-specified constraint $\mathcal{C}$. If $\mathcal{C}$ is anti-monotone, then the most effective way of using $\mathcal{C}$ to prune candidates is to push $\mathcal{C}$ "all the way" inside the mining computation. In the context of the SPIRIT framework, this means using $\mathcal{C}$ *as is* (rather than some relaxation of $\mathcal{C}$) in the pattern discovery loop. The optimality of this solution for anti-monotone $\mathcal{C}$ stems from two observations. First, using $\mathcal{C}$ clearly maximizes the amount of constraint-based pruning since the strongest possible constraint (i.e., $\mathcal{C}$ itself) is employed. Second, since $\mathcal{C}$ is anti-monotone, all subsequences of a frequent candidate k-sequence that survives constraint-based pruning are guaranteed to be in F (since they also satisfy $\mathcal{C}$). Thus, using the full strength of an anti-monotone constraint $\mathcal{C}$ maximizes the effectiveness of constraint-based pruning as well as support-based pruning. Note that this is exactly the methodology used in the CAP algorithm [15] for anti-monotone itemset constraints. An additional benefit of using anti-monotone constraints is that they significantly simplify the candidate generation and candidate pruning tasks. More specifically, generating C_k is nothing but an appropriate "self-join" operation over F_{k-1} and determining the pruned set P (Step 8) is simplified by the fact that all subsequences of candidates are guaranteed to satisfy the constraint.

When $\mathcal{C}$ is *not* anti-monotone, however, things are not that clear-cut. A simple solution, suggested by Ng et al. [15] for itemset constraints, is to take an anti-monotone relaxation of $\mathcal{C}$ and use that relaxation for candidate pruning. Nevertheless, this simple approach may not always be feasible. For example, our RE constraints for sequences do not admit any non-trivial anti-monotone relaxations. In such cases, the degree to which the constraint $\mathcal{C}$ is pushed inside the mining process (i.e., the strength of the (non anti-monotone) relaxation $\mathcal{C}'$ used for pruning) impacts the effectiveness of both constraint-based pruning and support-based pruning in different ways. More specifically, while increasing the strength of $\mathcal{C}'$ obviously increases the effectiveness of constraint-based pruning, it can also have a negative effect on support-based pruning. The reason is that, for any given sequence in C_k that survives constraint-based pruning, *the number of its subsequences that satisfy the stronger, non anti-monotone constraint* $\mathcal{C}'$ *may decrease.* Again, note that only subsequences that satisfy $\mathcal{C}'$ can be used for support-based pruning, since this is the only "state" maintained from previous passes (in F).

Pushing a non anti-monotone constraint $\mathcal{C}'$ in the pattern discovery loop can also increase the computational complexity of the candidate generation and pruning tasks. For candidate generation, the fact that $\mathcal{C}'$ is not anti-monotone means that some (or, all) of a candidate's subsequences may be absent from F. In some cases, a "brute-force" approach (based on just $\mathcal{C}'$) may be required to generate an initial set of candidates C_k. For candidate pruning, computing the subsequences of a candidate that satisfy $\mathcal{C}'$ may no longer be trivial, implying additional computational overhead. We should note, however, that candidate generation and pruning are inexpensive CPU-bound operations that typically constitute only a small fraction of the overall computational cost. This fact is also clearly demonstrated in our experimental results [10]. Thus, the major tradeoff that needs to be considered when choosing a specific $\mathcal{C}'$ from among the spectrum of possible relaxations of $\mathcal{C}$ is the extent to which that choice impacts the effectiveness of constraint-based and support-based pruning. The objective, of course, is to strike a reasonable balance between the two different types of pruning so as to minimize the number of candidates for which support is actually counted in each pass.

The SPIRIT Algorithms. The four SPIRIT algorithms for constrained pattern mining are points spanning the entire spectrum of relaxations for the user-specified RE constraint $\mathcal{C} \equiv \mathcal{R}$. Essentially, the four algorithms represent a natural progression, with each algorithm pushing a stronger relaxation of $\mathcal{R}$ than its predecessor in the pattern mining loop. The first SPIRIT algorithm, termed SPIRIT(N) ("N" for Naive), employs the weakest relaxation of $\mathcal{R}$ – it only prunes candidate sequences containing elements that do not appear in $\mathcal{R}$. The second algorithm, termed SPIRIT(L) ("L" for Legal), requires every candidate sequence to be *legal* with respect to some state of $\mathcal{A}_{\mathcal{R}}$. The third algorithm, termed SPIRIT(V) ("V" for Valid), goes one step further by filtering out candidate sequences that are not *valid with respect to any state of* $\mathcal{A}_{\mathcal{R}}$. Finally, the SPIRIT(R) algorithm ("R" for Regular) essentially pushes $\mathcal{R}$ "all the way" inside the mining process by counting support only for *valid* candidate sequences, i.e., sequences accepted by $\mathcal{A}_{\mathcal{R}}$. Table 1 summarizes the constraint choices for the four members of the SPIRIT family within the general framework depicted in Figure 5.

The SPIRIT algorithms employ novel techniques for candidate generation and pruning that, essentially, exploit the structure of the constraint automaton $\mathcal{A}_{\mathcal{R}}$ to implement these steps effectively. In what follows, we provide a brief overview of SPIRIT(L); the complete details for all SPIRIT algorithms can be found in [10].

Algorithm	**Relaxed Constraint $\mathcal{C}'$ ($\mathcal{C} \equiv \mathcal{R}$)**
SPIRIT(N)	all elements appear in $\mathcal{R}$
SPIRIT(L)	legal wrt some state of $\mathcal{A}_{\mathcal{R}}$
SPIRIT(V)	valid wrt some state of $\mathcal{A}_{\mathcal{R}}$
SPIRIT(R)	valid, i.e., $\mathcal{C}' \equiv \mathcal{C} \equiv \mathcal{R}$

Table 1: The four SPIRIT algorithms.

The SPIRIT(L) Algorithm. SPIRIT(L) uses the automaton $\mathcal{A}_{\mathcal{R}}$ to prune from C_k candidate k-sequences that are not *legal* with respect to any state of $\mathcal{A}_{\mathcal{R}}$. In our description, we use $F_k(b)$ to denote the set of frequent k-sequences that are legal with respect to state b of $\mathcal{A}_{\mathcal{R}}$.

In the *candidate generation* step of SPIRIT(L), given a state b in $\mathcal{A}_{\mathcal{R}}$, we add to C_k candidate k-sequences that are legal with respect to b and have the potential to be frequent.

LEMMA 3.1. Consider a k-sequence s that is legal with respect to state b in $\mathcal{A}_{\mathcal{R}}$, where $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$. For s to be frequent, $< s_1 \cdots s_{k-1} >$ must be in $F_{k-1}(b)$ and $< s_2 \cdots s_k >$ must be in $F_{k-1}(c)$.

Thus, the candidate sequences for state b can be computed as follows. For every sequence s in $F_{k-1}(b)$, if $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$, then for every sequence t in $F_{k-1}(c)$ such that $s_{j+1} = t_j$ for all $1 \leq j \leq k-2$, the candidate sequence $< s\ t_{k-1} >$ is added to C_k. This is basically a join of $F_{k-1}(b)$ and $F_{k-1}(c)$, on the condition that the $(k-2)$-length suffix of $s \in F_{k-1}(b)$ matches the $(k-2)$-length prefix of $t \in F_{k-1}(c)$ and $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$.

For the *candidate pruning* step of SPIRIT(L), note that, since we only count support for sequences that are legal with respect to some state of $\mathcal{A}_{\mathcal{R}}$, we can prune s from C_k only if we find a *legal* subsequence of s that is not frequent (i.e., not in F). The candidate pruning procedure computes the set of maximal subsequences of s with length less than k that are legal with respect to some state of automaton $\mathcal{A}_{\mathcal{R}}$. If any of these maximal subsequences is not contained in F, then s is deleted from C_k. We have proposed an novel *dynamic programming* algorithm that works off the structure of the constraint automaton $\mathcal{A}_{\mathcal{R}}$ to efficiently compute the maximal legal subsequences of a candidate sequence s; the details can be found in [10].

Finally, the *terminating condition* for SPIRIT(L) is that the set of frequent k-sequences that are legal with respect to the start state a of $\mathcal{A}_{\mathcal{R}}$ is empty; that is, $F_k(a)$ is empty.

Overview of Experimental Results. We have conducted an empirical study of the four SPIRIT algorithms with synthetic and real-life data sets. Note that, in general, RE constraints whose automata contain fewer transitions per state, fewer cycles, and longer paths tend to be more *selective*, since they impose more stringent restrictions on the ordering of items in the mined patterns. Our expectation is that for RE constraints that are more selective, constraint-based pruning will be very effective and the latter SPIRIT algorithms will perform better. On the other hand, less selective REs increase the importance of good support-based pruning, putting algorithms that use the RE constraint too aggressively (like SPIRIT(R)) at a disadvantage. Our experimental results corroborate our expectations. More specifically, our findings can be summarized as follows.

1. The SPIRIT(V) algorithm emerges as the overall winner, providing consistently good performance over the entire range of RE constraints. For certain REs, SPIRIT(V) is more than an order of magnitude faster than the "naive" SPIRIT(N) scheme.

2. For highly selective RE constraints, SPIRIT(R) outperforms the remaining algorithms. However, as the RE constraint becomes less selective, the number of candidates generated by SPIRIT(R) explodes and the algorithm fails to even complete execution for certain cases (it runs out of virtual memory).

3. The overheads of the candidate generation and pruning phases for the SPIRIT(L) and SPIRIT(V) algorithms are negligible. They typically constitute less than 1% of the total execution time, even for complex REs with automata containing large numbers of transitions, states, and cycles.

Thus, our experimental results have clearly validated our thesis that intelligent integration of RE constraints into the mining process can lead to significant performance benefits. For the complete details, the interested reader is referred to [10].

4. FUTURE WORK: COST-BASED CONSTRAINT PUSHING

As already discused in Section 3.2, complex model constraints (such as REs for frequent sequential patterns) that do not subscribe to nice properties, like anti-monotonicity and succinctness [15], raise a host of new issues and tradeoffs in the design of effective data mining strategies. Our experience with the SPIRIT algorithms has demonstrated two important facts. First, even in the case of such "difficult" user-specified model constraints, there can be a *whole spectrum of possible strategies* for exploiting the user's preferences; these strategies can differ, for example, on how aggressively the given constraints are pushed inside the mining loop. Second, within the space of all possible constraint-pushing strategies, there typically is no clear winner over all constraints and input data sets; instead, the winning strategy depends on a number of factors relating to the specific problem instance, such as the "selectivity" of the constraint and the relevant statistical characteristics of the input data. Furthermore, the best strategy may even vary across different stages of the model-extraction algorithm; for example, it is possible that more aggressive constraint-based pruning (e.g., SPIRIT(R)) can give better results if applied only in later iterations of the Apriori loop.

The above scenario is obviously reminiscent of traditional *query optimization* in relational database systems [5; 18], with the constraint-based mining strategies essentially corresponding to "query execution plans" for ad-hoc mining queries. We believe that a principled methodology for exploring the various performance issues and tradeoffs that arise in constraint-based, ad-hoc data mining should employ a *cost-based approach*, similar to that used in relational query optimizers. Of course, the increased complexity of model-mining algorithms and their corresponding constraints compared to simple relational algebra operations implies that cost-based optimization issues need to be fundamentally re-thought in the context of ad-hoc data mining. For example, the types of statistical synopses (e.g., histograms) of the underlying data that are needed to make effective execution-plan decisions have to be carefully designed based on the specific data-mining strategy and form of constraint at hand. We are currently investigating the design of cost-based approaches for integrating user constraints in various data-mining tasks.

5. CONCLUSIONS

Exploiting user-defined *model constraints* during the mining process can help users get exactly what they want from a data mining system, while ensuring system performance that is commensurate with the level of user focus. Attaining such performance, however, is not straightforward and, typically, requires the design of novel data mining algorithms that make effective use of the model constraints. This paper has provided an overview of our recent work on scalable, constraint-based algorithms for (1) decision tree construction with size and accuracy constraints for the desired model, and (2) sequential pattern extraction in the presence of RE constraints for the target patterns. By "pushing" the model constraints inside the mining process, our algorithms give mining users exactly the models that they are looking for, while achieving performance speedups that often exceed one order of magnitude. Further, our work on sequential pattern mining has uncovered some valuable insights into the tradeoffs that arise when complex constraints that do not subscribe to nice properties (like anti-monotonicity) are integrated into the mining process. Our thesis is that, in general, a *cost-based approach* (similar to that employed in conventional query optimizers) is needed to explore these tradeoffs in a principled manner and produce effective execution plans for ad-hoc mining queries. We believe that the design of such approaches can provide fertile ground for innovative data mining research in coming years.

ACKNOWLEDGEMENTS: We would like to acknowledge the valuable contributions of Kyuseok Shim and Dongjoon Hyun (our coauthors in [9; 10]) on most of the results surveyed in this short paper.

6. REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". In *Proc. of the 20th Intl. Conference on Very Large Data Bases*, pages 487–499, Santiago, Chile, August 1994.

[2] Hussein Almuallim. "An efficient algorithm for optimal pruning of decision trees". *Artificial Intelligence*, 83:346–362, 1996.

[3] Marko Bohanec and Ivan Bratko. "Trading Accuracy for Simplicity in Decision Trees". *Machine Learning*, 15:223–250, 1994.

[4] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *"Classification and Regression Trees"*. Chapman & Hall, 1984.

[5] Surajit Chaudhuri. "An Overview of Query Optimization in Relational Systems". In *Proc. of the 17th ACM Symposium on Principles of Database Systems*, Seattle, Washington, June 1998.

[6] Ming-Syan Chen, Jong Soo Park, and Philip S. Yu. "Efficient Data Mining for Path Traversal Patterns". *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, March 1998.

[7] Gregory F. Cooper and Edward Herskovits. "A Bayesian Method for Constructing Bayesian Belief Networks from Databases". In *Proc. of the 7th Annual Conference on Uncertainty in Artificial Intelligence*, pages 86–94, Los Angeles, California, 1991.

[8] Nir Friedman, Iftach Nachman, and Dana Peér. "Learning Bayesian Network Structure from Massive Datasets: The "Sparse Candidate" Algorithm". In *Proc. of the 15th Annual Conference on Uncertainty in Artificial Intelligence*, Stockholm, Sweden, August 1999.

[9] Minos Garofalakis, Dongjoon Hyun, Rajeev Rastogi, and Kyuseok Shim. "Efficient Algorithms for Constructing Decision Trees with Constraints". In *Proc. of the 6th ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining*, pages 335–339, Boston, Massachusetts, August 2000.

[10] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. "SPIRIT: Sequential Pattern Mining with Regular Expression Constraints". In *Proc. of the 25th Intl. Conference on Very Large Data Bases*, pages 223–234, Edinburgh, Scotland, September 1999. (Full version to appear in *IEEE Trans. on Knowledge and Data Engineering.*).

[11] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, and Wei-Yin Loh. "BOAT – Optimistic Decision Tree Construction". In *Proc. of the 1999 ACM SIGMOD Intl. Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999.

[12] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. "CURE: An Efficient Clustering Algorithm for Large Databases". In *Proc. of the 1998 ACM SIGMOD Intl. Conference on Management of Data*, pages 73–84, Seattle, Washington, June 1998.

[13] Harry R. Lewis and Christos Papadimitriou. *"Elements of the Theory of Computation"*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[14] Manish Mehta, Jorma Rissanen, and Rakesh Agrawal. "MDL-based Decsion Tree Pruning". In *Proc. of the 1st Intl. Conference on Knowledge Discovery and Data Mining*, Montreal, Quebec, August 1995.

[15] Raymond T. Ng, Laks V.S. Lakshmanan, Jiawei Han, and Alex Pang. "Exploratory Mining and Pruning Optimizations of Constrained Associations Rules". In *Proc. of the 1998 ACM SIGMOD Intl. Conference on Management of Data*, pages 13–24, Seattle, Washington, June 1998.

[16] J. Ross Quinlan and Ronald L. Rivest. "Inferring Decision Trees Using the Minimum Description Length Principle". *Information and Computation*, 80:227–248, 1989.

[17] Rajeev Rastogi and Kyuseok Shim. "PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning". In *Proc. of the 24th Intl. Conference on Very Large Data Bases*, pages 404–415, New York, USA, August 1998.

[18] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. "Access Path Selection in a Relational Database Management System". In *Proc. of the 1979 ACM SIGMOD Intl. Conference on Management of Data*, pages 23–34, Boston, Massachusetts, June 1979.

[19] John Shafer, Rakesh Agrawal, and Manish Mehta. "SPRINT: A Scalable Parallel Classifier for Data Mining". In *Proc. of the 22nd Intl. Conference on Very Large Data Bases*, Mumbai(Bombay), India, September 1996.

[20] Ramakrishnan Srikant and Rakesh Agrawal. "Mining Sequential Patterns: Generalizations and Performance Improvements". In *Proc. of the 5th Intl. Conference on Extending Database Technology (EDBT'96)*, Avignon, France, March 1996.

[21] Jason Tsong-Li Wang, Gung-Wei Chirn, Thomas G. Marr, Bruce Shapiro, Dennis Shasha, and Kaizhong Zhang. "Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results". In *Proc. of the 1994 ACM SIGMOD Intl. Conference on Management of Data*, pages 115–125, Minneapolis, Minnesota, May 1994.