Parallel and Distributed Search for Structure in Multivariate Time Series

Tim Oates, Matthew D. Schmill and Paul R. Cohen
Computer Science Department, LGRC
University of Massachusetts
Box 34610
Amherst, MA 01003-4610
Voice: 1-413-545-3616
Fax: 1-413-545-1249
{oates,schmill,cohen}@cs.umass.edu

Abstract

Efficient data mining algorithms are crucial for effective knowledge discovery. We present the Multi-Stream Dependency Detection (MSDD) data mining algorithm that performs a systematic search for structure in multivariate time series of categorical data. The systematicity of MSDD's search makes implementation of both parallel and distributed versions straightforward. Distributing the search for structure over multiple processors or networked machines makes mining of large numbers of databases or very large databases feasible. We present results showing that MSDD efficiently finds complex structure in multivariate time series, and that the distributed version finds the same structure in approximately $1/n$ of the time required by MSDD, where n is the number of machines across which the search is distributed. MSDD differs from other data mining algorithms in the complexity of the structure that it can find. MSDD also requires no domain knowledge to focus or limit its search, although such knowledge is easily incorporated when it is available.

Keywords: data mining, parallel and distributed algorithms, systematic search, multivariate time series

1 Introduction

Knowledge discovery in databases (KDD) is an iterative process in which data is repeatedly transformed and analyzed to reveal hidden structure.[1] The analysis portion of KDD, the actual search for structure in data, is called data mining. Efficient data mining algorithms are necessary when the number of databases to be mined is large, when the amount of data in a given database is large, or when many iterations of the transform/analyze cycle are required. The ease with which vast quantities of electronically available information can be generated and stored gives rise to the former two conditions. Parallel and distributed data mining algorithms can quickly mine large amounts of data by taking full advantage of existing hardware, both multiple processors on one machine and multiple machines on a network. Multi-Stream Dependency Detection (MSDD) is an easily parallelized data mining algorithm that performs an efficient systematic search for complex structure in multivariate time series of categorical data. Consider the streams of data flowing from the monitors in an intensive care unit, or periodic measurements of various indicators of the health of the economy, a computer network, or the earth's various ecosystems. There is clearly utility in determining how current and past values in those streams are related to future values. Empirical results demonstrate that MSDD's representation and core search algorithm are capable of expressing and finding interesting and complex structure, and that the distributed version of MSDD (D-MSDD) running on n networked machines is approximately n times faster than the centralized version.

MSDD finds *dependencies* between patterns of values that occur in multivariate time series of categorical data. We call each univariate time series a *stream* of data. Dependencies are unexpectedly frequent or infrequent co-occurrences of patterns in the streams, and can be expressed as *rules* of the following form: "If an instance of pattern x begins in the streams at time t, then an instance of pattern y will begin at time $t+\delta$ with probability p." A dependency is strong if the empirically determined value of p is very different from the probability of seeing a co-occurrence of x and y under the assumption that they are independent. Strong dependencies capture structure in the streams because they tell us that there is a relationship between their constituent patterns, that occurrences of those patterns are not independent.

Because many interesting and important databases contain a temporal component, such as those describing the ebb and flow of the stock market or the health of a patient in an intensive care unit, the constituent patterns of MSDD rules (x and y) can span multiple time steps, and they can be associated across arbitrary temporal lags (δ). In addition, right-hand-sides of rules can be as complex as left-hand-sides, specifying values for multiple streams over multiple time steps. Contrast the expressiveness of MSDD's rules with those found by other rule discovery algorithms, which often assume that databases contain temporally independent feature vectors and that only the values of a single pre-specified feature can serve as rule right-hand-sides. MSDD discovers structure that these algorithms cannot even represent.

MSDD finds the k strongest dependencies in a set of streams by performing a *systematic*

[1] KDD is a new term for an old activity, exploratory data analysis (EDA) [14]. Whereas EDA is typically associated with continuous-valued data and statistical procedures for transformation and analysis, KDD encompasses more complex forms of data (e.g. relational data) and transformation and analysis techniques (e.g. induction learning algorithms). However, KDD and EDA share the same underlying principles and goals.

search over the space of all possible dependencies. Systematic search expands the children of search nodes in a manner that ensures that no node can ever be generated more than once [6, 8–10, 15]. Because non-redundant expansion is achieved without access to large, rapidly changing data structures, such as lists of open and closed nodes, the search space can be divided between multiple processes on multiple machines. Only a small amount of inter-process communication is required to keep the list of the k strongest dependencies globally consistent.

Because MSDD returns a list of the k strongest dependencies, rather than all of the dependencies that it encounters during the search, it is possible to use upper bounds on the values of a node's descendants to prune. The expressiveness of MSDD's rule representation allows the algorithm to find complex structure in data, but also leads to an exponential search space, making effective pruning essential. We use the G statistic as a measure of dependency strength, and develop optimistic bounds on the value of G for the descendants of a node to prune.

The remainder of the paper is organized as follows: Section 2 discusses systematic search in detail. Section 3 presents the MSDD algorithm, defines the space of dependencies that the algorithm explores, and develops domain independent pruning techniques. Section 4 shows how the systematicity of MSDD's search can be exploited to develop parallel and distributed versions of the algorithm. Section 5 explores the ability of the core algorithm to find interesting and complex structure in multivariate time series, and compares the speed of the centralized (MSDD) and distributed (D-MSDD) versions of the algorithm. Section 6 reviews related work, and Section 7 concludes and explores future directions.

2 Systematic Search

MSDD's search for the k strongest dependencies in a set of streams is *systematic*, leading to search efficiency and parallelizability. Systematic search non-redundantly enumerates the elements of search spaces for which the value or semantics of any given node are independent of the path from the root to that node. Webb calls such search spaces *unordered* [15]. Consider the space of disjunctive concepts over the set of literals $\{A, B, C\}$. Given a root node containing the empty disjunct, $false$, and a set of search operators that add a single literal to a node's concept, a *non*-systematic elaboration of the search space is shown in Figure 1. Note that the concept $A \vee B \vee C$ appears six times, with each occurrence being semantically the same as the other five, yet syntactically distinct. In the space of disjunctive concepts, the semantics of any node's concept is unaffected by the path taken from the root to that node. For example, the two paths below yield semantically identical leaf nodes:

$$false \rightarrow A \rightarrow A \vee B \rightarrow A \vee B \vee C$$
$$false \rightarrow C \rightarrow C \vee B \rightarrow C \vee B \vee A$$

The search tree in Figure 1 contains six syntactic variants of the concept $A \vee B \vee C$, and two syntactic variants of the concepts $A \vee B$, $A \vee C$ and $B \vee C$. Clearly, naive expansion of nodes in unordered search spaces leads to redundant generation and wasted computation.

Systematic search of unordered spaces generates no more than one syntactic form of each semantically distinct concept. That is accomplished by imposing an order on the search

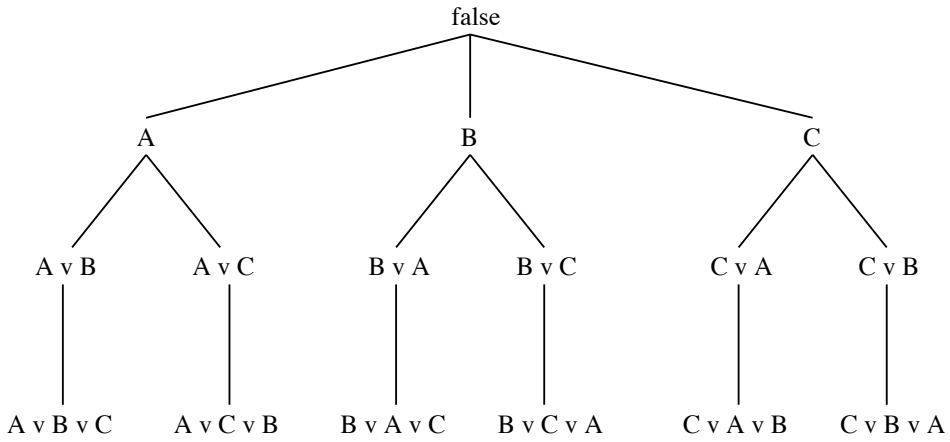

Figure 1: A naive elaboration of the space of disjunctive concepts over the set $\{A, B, C\}$ generated by applying all valid search operators at every node. Naive search generates multiple syntactic variants of individual concepts.

operators used to generate the children of a node, and applying only those operators at a node that are higher in the ordering than all other operators already applied along the path to the node. Let op_A, op_B and op_C be the operators that add the literals A, B and C respectively to a node's concept. If we order those operators so that $op_A < op_B < op_C$, then the corresponding space of disjunctive concepts can be enumerated systematically as shown in Figure 2. Note that each semantically distinct concept appears exactly once. The concept A is obtained by applying operator op_A to the root node. Because $op_B > op_A$ and $op_C > op_A$, both op_B and op_C can be applied to the concept A, generating the child concepts $A \vee B$ and $A \vee C$. In contrast, the concept C, which is obtained by applying op_C to the root node, has no children. Because all other operators (op_A and op_B) are lower in the ordering than op_C, none will be applied and no children will be generated.

Systematic search of unordered spaces is significantly more efficient than naive search because many fewer nodes are generated. Naive search may be made more efficient by maintaining a list of expanded nodes, and checking each newly expanded node against that list to determine if a semantically identical syntactic variant already exists somewhere in the tree. However, this approach adds both computational and storage overhead, and it does not completely eliminate redundant generation of syntactic variants. For example, if the concept $A \vee B$ exists in the tree, the subtree rooted at the concept $B \vee A$ can be pruned when that node is expanded. However, the concept $B \vee A$ is a redundant syntactic variant of the concept $A \vee B$, and pruning cannot take place until $B \vee A$ is expanded and the concept $A \vee B$ is found to already exist. In contrast, systematic search prunes the subtree rooted at $B \vee A$ without ever expanding the redundant root of that subtree.

The fact that unordered search spaces can be explored without redundant node generation through systematic search is the key to parallelizing MSDD. Given any search node in the tree, the only information required to simultaneously generate that node's children and avoid redundant node generation is the operator ordering (e.g. $op_A < op_B < op_C$). Consider the systematic search space shown in Figure 2. Each of the subtrees rooted at the three children of the root node, A, B and C, could be expanded by systematic search algorithms running on

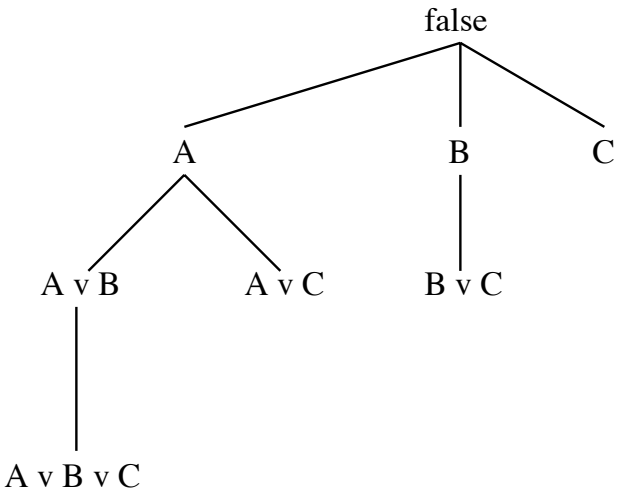

Figure 2: Systematic elaboration of the space of disjunctive concepts over the set $\{A, B, C\}$. Only one syntactic form of each semantically distinct concept is expanded.

different machines. The machine expanding node B would generate its children by applying all operators higher in the ordering than op_B, yielding the single child $B \vee C$ through the application of operator op_C. Because no operators are higher in the ordering than op_C, the node $B \vee C$ has no children, and the subtree rooted at B has been completely explored. Not only was no communication with the search algorithms running on the other machines required to expand that subtree, there was no need to know that they even existed or that other portions of the search space were being explored.

After presenting the core MSDD algorithm in the next section, we discuss the use of systematic search to create parallel and distributed versions in Section 4.

3 The MSDD Algorithm

MSDD accepts as input a set of streams that are used to define the space of dependencies the algorithm will search and to evaluate the strength of dependencies. The set of m input streams is denoted $\mathcal{S} = \{s_1, \ldots, s_m\}$, and the i^{th} stream is composed of categorical values, called *tokens*, taken from the set $\mathcal{V}_i$. All of the streams in $\mathcal{S}$ must have the same length, and we assume that all of the tokens occurring at a given position in the streams were recorded synchronously. Consider the following streams:

```
S1: D B B A D C D A B C
S2: X Y Y Z Y X Z Z X Y
S3: 2 1 3 2 2 1 2 3 2 1
```

All three streams contain ten tokens. Stream s_1 is composed of tokens drawn from the set $\mathcal{V}_1 = \{A, B, C, D\}$. Likewise, $\mathcal{V}_2 = \{X, Y, Z\}$ and $\mathcal{V}_3 = \{1, 2, 3\}$.

Recall that MSDD searches for dependencies expressed as rules of the following form: "If an instance of pattern x begins in the streams at time t, then an instance of pattern y will begin at time $t + \delta$ with probability p." Such rules are denoted $x \stackrel{\delta}{\Rightarrow} y$. We call x

the *precursor* and y the *successor*. p is computed by counting the number of time steps on which an occurrence of the precursor is followed δ time steps later by an occurrence of the successor, and dividing by the total number of occurrences of the precursor. To keep the space of patterns and the space of dependencies finite, we consider patterns that span no more than a constant number of adjacent time steps. Precursors can span at most w_p time steps, and successors can span at most w_s time steps. Both w_p and w_s are parameters of the MSDD algorithm.

Patterns of tokens (precursors and successors) are represented as sets of 3-tuples of the form $\tau = (v, s, d)$. Each 3-tuple specifies a stream, s, a token value for that stream, v, and a temporal offset, d, relative to an arbitrary time t. Because such patterns can specify token values for multiple streams over multiple time steps, they are called *multitokens*.[2] Tuples that appear in precursors are drawn from the set $T_p = \{(v, s, d) | 1 \leq s \leq m, v \in \mathcal{V}_s, 0 \leq d < w_p\}$. Likewise, tuples that appear in successors are drawn from the set $T_s = \{(v, s, d) | 1 \leq s \leq m, v \in \mathcal{V}_s, 0 \leq d < w_s\}$. For example, the multitoken $x = \{(B, 1, 0), (Y, 2, 1)\}$ specifies a pattern that occurs twice in the streams above. For $t = 2$ and $t = 9$, we see token B in stream one at time $t + 0$ and token Y in stream 2 at time $t + 1$. Likewise, the multitoken $y = \{(X, 2, 0), (2, 3, 0), (Y, 2, 1), (1, 3, 1)\}$ occurs twice in the streams above, once at time $t = 1$ and again at time $t = 9$. The following streams are a copy of the streams above, except we have removed all tokens not involved in occurrences of x or y.

```
S1: B . . . . . . . B .
S2: X Y . . . . . . X Y
S3: 2 1 . . . . . . 2 1
```

Although the input parameters w_p, w_s and δ together with the streams in $\mathcal{S}$ define the space of precursors, successors and dependencies that MSDD will explore, any given dependency can be expressed for a large number of different settings of those parameters. This makes MSDD's ability to find structure in streams robust with respect to variations in the settings of w_p, w_s and δ. Figure 3 shows how one particular relationship between two multitokens can be captured by dependency rules for three different settings of w_p, w_s and δ. Unfortunately, it is also the case that any given dependency can often be expressed in different ways within the confines of a single specification of values for w_p, w_s and δ. For example, the patterns in the left-most two blocks in Figure 3 could be shifted right by one time step, resulting in a syntactically distinct yet semantically identical dependency. Therefore, during the search we prune dependencies that can be shifted in the manner just described, knowing that an unshiftable syntactic variant exists elsewhere in the search space.

MSDD performs a general-to-specific search over the space of possible dependencies, starting with a root node that specifies no token values for either the precursor or the successor ($\{\} \Rightarrow \{\}$). Search operators either add a term from T_p to a node's precursor or add a term from T_s to a node's successor. To perform a systematic search over the space of possible dependencies between multitokens, we impose the following order on the terms in T_p and T_s: All of the terms in T_p are lower than all of the terms in T_s. For any $\tau_i, \tau_j \in T_p$, τ_i is lower than τ_j if $d_i < d_j$ or if $d_i = d_j$ and $s_i < s_j$. That is, terms in T_p are ordered first by their

[2]The definition of a multitoken given here is an extension of the one given in previous descriptions of the algorithm [6].

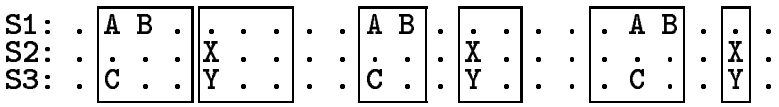

Figure 3: Any given relationship between two multitokens can be captured by dependencies for a wide variety of settings of w_p, w_s and δ. Therefore, MSDD's ability to find structure in streams is robust with respect to variations in those parameters. The figure shows one such relationship as captured by dependencies with the following values: $w_p = w_s = 3$, $\delta = 3$ (the left-most two blocks); $w_p = w_s = 2$, $\delta = 3$ (the middle two blocks); and $w_p = 3$, $w_s = 1$, $\delta = 4$ (the right-most two blocks).

temporal offset, and then by their stream index. Likewise for terms in T_s. By ordering all of the terms in T_p below all of the terms in T_s we force precursors to be constructed before successors. As long as no terms have been added to a node's successor, terms may be added to its precursor. However, as soon as a single term is added to the successor, the precursor must thereafter remain unchanged because all of the search operators that would add new terms to the precursor are lower than the operator that added the term to the successor.

Because MSDD returns a list of the k strongest dependencies, if we can derive an upper bound on the value of the evaluation function f for all of the descendants of a given node, then we can use that bound to prune the search. Suppose the function $fmax(N)$ returns a value such that no descendant of N can have an f value greater than $fmax(N)$. If at some point during the search we remove a node N from the open list for expansion, and $fmax(N)$ is less than the f value of all k nodes in the current list of best nodes, then we can prune N. There is no need to generate N's children because none of the descendants of N can have an f value higher than any of the current best nodes; none of N's descendants can be one of the k best nodes that will be returned by the search. The use of an optimistic bounding function is similar to the idea behind the h function in A* search. That is, if a goal node is found whose cost is less than underestimates of the cost-to-goal of all other nodes currently under consideration, then that goal node must be optimal. Pruning based on optimistic estimates of the value of the descendants of a node has been used infrequently in rule induction algorithms, with ITRULE [12] and OPUS [15] being notable exceptions.

The G statistic computed for 2x2 contingency tables is a statistical measure of non-independence, and we have derived bounds on the value of G for the descendants of a node, making it an ideal candidate for f. Given a 2x2 contingency table whose four cells (n_1, n_2, n_3, n_4) indicate the frequency with which occurrences and non-occurrences of the precursor are followed δ time steps later by occurrences and non-occurrences of the successor, G is computed as follows:

$$G = 2\sum_{i=1}^{4} n_i log(n_i/\hat{n}_i)$$

$\hat{n}_i$ is the expected value of n_i under the assumption of independence, and is computed from the margins and the table total [16]. Because the ordering imposed on MSDD's search operators causes precursors to be elaborated before successors, it is possible to reason about how the mass of a contingency table may move as one descends along a path in the search tree. As a consequence, given any node's dependency $(x \stackrel{\delta}{\Rightarrow} y)$ and contingency table (n_1, n_2, n_3, n_4),

we can establish the following upper bound on the value of G for the descendants of that node:

$$Gmax(n_1, n_2, n_3, n_4) =$$

$$max \left(\begin{array}{l} (1) \\ \quad \text{if } n_1 \leq n_2 + n_3 + n_4 \\ \quad\quad G(n_1, 0, 0, n_2 + n_3 + n_4) \\ \quad \text{else} \\ \quad\quad G(\frac{n_1+n_2+n_3+n_4}{2}, 0, 0, \frac{n_1+n_2+n_3+n_4}{2}) \\ \\ (2) \\ \quad \text{if } n_1 \geq abs(n_2 - n_3) \\ \quad\quad G(0, \frac{n_1+n_2+n_3}{2}, \frac{n_1+n_2+n_3}{2}, n_4) \\ \quad \text{else if } n_2 > n_3 \\ \quad\quad G(0, n_2, n_1 + n_3, n_4) \\ \quad \text{else} \\ \quad\quad G(0, n_1 + n_2, n_3, n_4) \end{array} \right)$$

We omit the proof due to lack of space.

A pseudo-code specification of MSDD is given below in Algorithm 3.1. MSDD requires seven input parameters: a set of streams, $\mathcal{S}$, that is to be searched for structure; the maximum number of time steps that precursor and successor multitokens can span, w_p and w_s respectively; the lag to be used when counting co-occurrences of precursors and successors, δ; an evaluation function that determines the strength of a given dependency, f; a function that returns an upper bound on the value of f for the descendant of a node, $fmax$; and the number of dependency rules to return, k. In all of our experiments, we used G as the node evaluation function and $Gmax$ as defined above to prune.

Algorithm 3.1 MSDD

MSDD($\mathcal{S}, w_p, w_s, \delta, f$,fmax, k)

1. *best* = ()
2. *nodes* = ($\{\} \Rightarrow \{\}$)
3. while not EMPTY(*nodes*) do
   a. *node* = NEXT-NODE(*open*)
   b. *children* = SYSTEMATICALLY-EXPAND(*node*, w_p, w_s)
   c. *children* = REMOVE-PRUNABLE(*children*, $fmax$)
   d. add *children* to *open*
   e. for *child* in *children* do
      i. if LENGTH(*best*) $< k$ or $\exists n \in$ *best* s.t. $f(child, \mathcal{S}, \delta) > f(n, \mathcal{S}, \delta)$ then add *child* to *best*
      i. if LENGTH(*best*) $> k$ then remove from *best* the node with the lowest f value
4. return *best*

4 Parallel and Distributed MSDD

In the same way MSDD guarantees non-redundant generation of search nodes, MSDD guarantees that distinct nodes at the same depth in the search tree are parent to completely disjoint sets of children. The result is a search space that can be trivially partitioned into computationally independent subsets, and consequently MSDD is an algorithm well suited for parallel and distributed implemetation. We begin by discussing a parallel implementation of MSDD

The easy partitioning of MSDD's search space allows us to treat any intermediate search node as a root of a new search tree. Consider the goal of "basic" MSDD; search for elaborations on the completely general rule $\{\} \rightarrow \{\}$ that maximize the evaluation function f. A more general, parallelized approach is to search for elaborations on *an arbitrary rule* that maximize f. In this way, we treat each node as an "island", independent of anything else MSDD has learned, and perform the search is if it were the root. The code for a parallel MSDD is given in algorithm 4.1.

Algorithm 4.1 PARALLEL MSDD

P-MSDD*(node)*
SPAWN {

1. *children* ← SYSTEMATICALLY-EXPAND*(node,w_p,w_s)*
2. for *ch* in *children* do
   a. if LENGTH(*best*) $< k$ or $\exists n \in best$ *s.t.* $f(child, \mathcal{S}, \delta) > f(n, \mathcal{S}, \delta)$ then
      GET-SEMAPHORE*(best)*
      add *ch* to *best*
      RELEASE-SEMAPHORE*(best)*
   b. if (not PRUNE*(ch)*) then
      P-MSDD*(ch)* }

An efficient parallel implementation of MSDD is possible because the search at any given node does not require access to previously elaborated search tree. The threads of P-MSDD need only non-exclusive read access to the time series and exclusive write access for insertions into the queue of k best nodes. Using a semaphore to provide exclusive writes to the k best list, the computation of MSDD can be effectively balanced over many processing elements.

4.1 Distributed MSDD

The implementation of parallel MSDD can be translated easily to an efficient distributed algorithm. This algorithm, D-MSDD, makes use of a client-server TCP tools to perform D-MSDD's search over a network of cooperating systems.

The D-MSDD algorithm begins with the *server*. The server is responsible for initiating the search, mediating communication, and declaring the search finished. Any number of *client* machines may contact the server to declare themselves as eligible for aiding in the

search. This declaration process is called *registration*, where the server takes note of each client machine, issuing it a unique identifier for future communication. Once a desirable number of clients have registered, the server is ready to initiate the search process. This process, executed on both client and server machines, is given in algorithm 4.2.

Algorithm 4.2 DISTRIBUTED MSDD

D-MSDD()
1. loop until SEARCH-FINISHED
   if EMPTY*(Agenda)* then
   REQUEST-MORE-NODES
   else
   a. *CurrentNode* $\leftarrow$ NEXT-NODE*(Agenda)*
   b. *Children* $\leftarrow$ GENERATE-CHILDREN*(CurrentNode)*
   c. for *ch* in *Children* do
      a. if LENGTH(*best*) $< k$ or $\exists n \in best$ *s.t.* $f(child, \mathcal{S}, \delta) > f(n, \mathcal{S}, \delta)$ then
         GET-SEMAPHORE*(best)*
         add *ch* to *best*
         RELEASE-SEMAPHORE*(best)*
      b. if (not PRUNE*(ch)*) then
         add *ch* to *agenda*

The distributed search proceeds on each participant machine according to a local *agenda*. Each machine's agenda is an independent partition of the unexplored MSDD search space. As with P-MSDD, the only shared strucutres are the list of k best dependencies and the dataset itself. Each machine participating in a D-MSDD search maintains local copies of these structures, keeping them synchronized through network message passing. We simulate the accessing of shared data by sending *best* messages to describe a candidate node for the k best list. We emulate the load balancing that goes on on a parallel machine by passing *node* messages that transfer nodes from an overloaded machine's agenda to a machine with a lighter agenda.

5 Empirical Results

To demonstrate the power and flexibility of MSDD, we applied the algorithm to the real-world task of discovering structure in DNA sequences represented as strings of nucleotides (one of A, G, T or C). The dataset, which was taken from the UC Irvine machine learning repository, contains 106 sequences, each comprising 57 nucleotides. Half of the sequences correspond to *promoters*, genetic regions that initiate the expression of an adjacent gene. Most rule discovery algorithms learn *classification rules* from this dataset, rules that use combinations of nucleotides to predict whether a sequence is a promoter. In contrast, MSDD can search for more complex structure, finding rules that combine subsets of domain variables on *both* sides of rules. Running MSDD with $w_p = w_s = 1$ and $\delta = 0$ eliminates the temporal

component of dependencies, allowing the algorithm to find structure in a set of temporally independent vectors of categorical values. Note that the space of dependencies between patterns of nucleotides is enormous, containing in excess of 10^{80} elements.

The biological literature suggests that a small subset of the 57 positions in each sequence are important in distinguishing promoters from non-promoters [7]; valid positions range from -50 to $+6$, and are denoted `Px` where $-50 \leq x \leq +6$. Therefore, we ran an initial MSDD search with $k = 50$ to depth five in an attempt to determine those positions. Only the class label ($+$ or $-$) and values for 13 of the 57 possible positions appeared in one or more of the 50 strongest dependencies. We then ran a depth-unlimited search for the $k = 100$ strongest dependencies in that subset of relevant nucleotide positions. The top six rules of the 100 rules returned by that search are shown below in Table 1.

Rule	**Contingency Table**	**G**
1. `(P-35 T) (P-14 T) (P-9 T)` $\Rightarrow$ `(P-13 A) (P-10 A)`	(14 2 1 89)	63.39
2. `(CLASS +) (P-14 T) (P-9 T)` $\Rightarrow$ `(P-13 A) (P-10 A)`	(14 3 1 89)	59.90
3. `(P-35 T) (P-14 T) (P-9 T)` $\Rightarrow$ `(P-13 A) (P-10 A) (P-2 C)`	(12 4 0 90)	56.88
4. `(CLASS +) (P-14 T) (P-7 C)` $\Rightarrow$ `(P-13 A) (P-8 G)`	(12 2 1 91)	56.38
5. `(P-14 T) (P-10 A) (P-9 T)` $\Rightarrow$ `(P-13 A) (P-7 C) (P-2 C)`	(11 3 0 92)	56.11
6. `(P-35 T) (P-34 G)` $\Rightarrow$ `(CLASS +)`	(34 1 19 52)	55.38

Table 1: The top six rules of 100 found by MSDD in the promoter dataset.

Both the syntax and semantics of the rules in Table 1 are interesting. First, consider their syntax. Five of the six rules have conjunctive right-hand-sides, only half of them refer to the class label, and only rule number 6 is a classification rule. Most rule discovery algorithms are incapable of representing the five strongest dependencies in this dataset. Consider the most highly ranked rule, `(P-35 T) (P-14 T) (P-9 T)` $\Rightarrow$ `(P-13 A) (P-10 A)`. Rule discovery algorithms with less expressive representations may be able to find the two "constituents" of that rule: `(P-35 T) (P-14 T) (P-9 T)` $\Rightarrow$ `(P-13 A)` and `(P-35 T) (P-14 T) (P-9 T)` $\Rightarrow$ `(P-10 A)`. However, the G value of the former rule is 33.91 and the value of the latter is 19.45; both are much lower than the value of the original rule with a complex right-hand-side, 63.39.

The rules in Table 1 capture semantically interesting structure in the data, as demonstrated by comparing them to a previously published domain theory derived from the biological literature [13]. The domain theory states that promoters contain patterns drawn from each of three sets, two different sets of *contact* patterns (ct_1 and ct_2) and one set of *conformation* patterns. All of the patterns in ct_1 contain a `T` at position `P-35` and a `G` at position `P-34`, and we see `(P-35 T)` in several of the rules found by MSDD. One of the patterns in ct_2 specifies the sequence `TATAAT` starting at position `P-14`. The first rule in Table 1 captures the fact that promoters have patterns from both ct_1 and ct_2 by strongly associating an occurrence of `(P-35 T)` and part of the pattern from ct_2, `(P-14 T) (P-9 T)`, with other parts of that pattern, `(P-13 A) (P-10 A)`. Portions of the conformation pattern `GCGCC*CC` occurring at position `P-8` are captured in rules 3, 4 and 5. Also, because all of the patterns in ct_1 have `T` at position `P-35` and `G` at position `P-34`, co-occurrences of those nucleotides

are a strong indicator that the sequence is a promoter. That structure is captured in the one classification rule in Table 1.

In another set of experiments, we compared the performance of MSDD and D-MSDD. For each of three datasets, the two algorithms found the $k = 20$ strongest dependencies. We ran D-MSDD on both two and three machines connected via a local area network. The datasets, which were all taken from the UC Irvine repository, included chess end-games, solar flares, and congressional voting records. The results are summarized below in Tables 2, 3 and 4. Each table shows the number of nodes expanded, CPU cycles consumed, and the number of messages sent to keep the list of the k best dependencies consistent. When D-MSDD ran on two and three machines (the D-MSDD – 2 and D-MSDD – 3 cases respectively), table entries contain the sum of the value over all machines participating in the search. Note that relatively few search nodes were required to find the 20 strongest dependencies in exponential spaces; pruning based on optimistic estimates of G is effective. Because the distributed search may be at different depths on different machines, the total number of nodes expanded may vary depending on when strong dependencies are found and used for subsequent pruning. However, in each case the total number of CPU cycles required to complete the search remains fairly constant, independent of the number of participating machines. Because load-balancing between the machines is fine grained, n machines can complete the search for structure roughly n times faster than one machine.

Algorithm	**Search Nodes**	**CPU Cycles**	**Messages**
MSDD	107,858	6,911,826	0
D-MSDD – 2	124,234	7,915,858	7024
D-MSDD – 3	115,375	7,963,435	6697

Table 2: Comparison of MSDD and D-MSDD on the congressional voting records dataset.

Algorithm	**Search Nodes**	**CPU Cycles**	**Messages**
MSDD	22,346	1,507,160	0
D-MSDD – 2	27,073	1,573,309	1321
D-MSDD – 3	31,955	1,793,743	2964

Table 3: Comparison of MSDD and D-MSDD on the chess dataset.

Algorithm	**Search Nodes**	**CPU Cycles**	**Messages**
MSDD	12,199	805,920	0
D-MSDD – 2	13,544	906,188	457
D-MSDD – 3	17,941	1,095,695	1,706

Table 4: Comparison of MSDD and D-MSDD on the solar flares dataset.

6 Related Work

Several systematic search algorithms have appeared in the literature [6,8–10,15], all of them variations on the basic idea of imposing an order on search operators, and applying only those operators at a node that are higher in the order than all other operators that have been applied on the path from the root to the node. Some of these cut off the search at an arbitrary depth to limit the size of the search space (e.g. [8,10]). In contrast, MSDD returns a list of the k strongest dependencies, regardless of the depth at which they exists in the search tree. Our use of optimistic bounds on the value of the node evaluation function for pruning systematic search spaces is similar to the OPUS algorithm [15], which in turn is a generalization of the same idea as applied to non-systematic search in the ITRULE induction algorithm [12]. MSDD and ITRULE return the k best rules, whereas OPUS returns a single goal node or the single node with the highest value.

Both parallel algorithms and consideration of data with a temporal component are rare in the KDD and data mining literature. Holsheimer and Kersten describe a system for inducing rules from large relational databases that performs a parallelized beam search over the space of possible rules and accesses the data through a parallel DBMS [3]. However, their system is limited to classification rules (a conjunct of literals predicting a single literal), and it can miss high quality rules due to the use of beam search. Aronis and Provost developed a parallel algorithm that builds new features from existing features in relational databases [1]. The newly constructed features are then passed to a standard (serial) inductive learning algorithm. While parallelism speeds the search for new features, it does not affect the speed with which rules using those features can be learned. Finally, Berndt and Clifford describe a dynamic programming algorithm for finding recurring patterns in univariate time series [2].

Our approach to rule discovery in databases differs from others, including all of those cited above that perform systematic search, in that it does not require the user to specify a set of target concepts to serve as rule right-hand-sides. Most existing rule discovery methods return rules that use the values of one or more domain variables (e.g. attribute values), appropriately combined, to characterize one of a small number of pre-specified target concepts (e.g. class labels). Often, such algorithms must be run multiple times to learn rules for multiple concepts, once for each concept [10,15], losing the benefit of pruning information generated during previous runs. The ITRULE algorithm is somewhat more general in that it simultaneously searches for rules whose right-hand-sides can specify the value of any single domain variable, not just the one containing the class label. In contrast, MSDD explores the space of dependencies between pairs of arbitrary patterns of values, looking for structure in the data regardless of where it exists. That is advantageous when there is no *a priori* knowledge concerning probable relationships that exist in the streams, or when the "target concepts" themselves involve complex combinations of domain variables that may be difficult for a human to accurately express.

7 Conclusion

In this paper we presented the MSDD data mining algorithm which performs a systematic search for structure in multivariate time series. MSDD discovers the k strongest dependencies

between pairs of multitokens, arbitrary patterns of values that can span multiple streams and multiple time steps. MSDD prunes the search space with an upper bound on the value of the descendant of a given node, and we derived such a bound on the value of G. We recognized that systematic search over unordered spaces is easily parallelized, and developed D-MSDD, a distributed version of MSDD. MSDD is a powerful tool for discovering complex structure in very large databases due to the efficiency and expressiveness of the core algorithm and the ease with which the search for structure can be distributed over multiple machines on a network via D-MSDD.

Current work includes an incremental version (I-MSDD) of the basic algorithm [11], and an investigation of techniques for dynamically adjusting the search operator ordering to maximize the effects of pruning [15]. In terms of applications, work is proceeding in two areas. First, we are using MSDD to find structure in the interactions of an artificial agent with its environment for the purpose of learning planning operators [5]. Precursor multitokens encode state/action pairs, and successor multitokens encode state changes. Strong dependencies capture state changes that the agent can reliably bring about. Second, we are using MSDD to learn how current and past states of computer networks are related to future states for the purpose of acquiring rules that will allow network managers to predict and avoid problems in their networks before they arise [4].

Acknowledgements

This research was supported by ARPA/Rome Laboratory under contract numbers F30602-91-C-0076 and F30602-93-0100, and by a National Defense Science and Engineering Graduate Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References

[1] John M. Aronis and Foster J. Provost. Efficiently constructing relational features from background knowledge for inductive machine learning. In *Working Notes of the Knowledge Discovery in Databases Workshop*, pages 347–358, 1994.

[2] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Working Notes of the Knowledge Discovery in Databases Workshop*, pages 359–370, 1994.

[3] Marcel Holsheimer and Martin L. Kersten. Architectural support for data mining. In *Working Notes of the Knowledge Discovery in Databases Workshop*, pages 217–228, 1994.

[4] Tim Oates. Fault identification in computer networks: A review and a new approach. Technical Report 95-113, University of Massachusetts at Amherst, Computer Science Department, 1995.

[5] Tim Oates and Paul R. Cohen. Searching for structure in multiple streams of data. Submitted to the Thirteenth International Conference on Machine Learning, 1996.

[6] Tim Oates, Dawn E. Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In *Preliminary Papers of the Fifth International Workshop on Artificial Intelligence and Statistics*, pages 417–423, 1994. Does not contain work on incremental algorithm reported in book version.

[7] M. O'neill. Escherichia coli promoters: I - consensus as it relates to spacing class, specificity, retreat substructure, and three dimensional organization. *Journal of Biological Chemistry*, 264:5522–5530, 1989.

[8] Patricia Riddle, Richard Segal, and Oren Etzioni. Representation design and brute-force induction in a boeing manufacturing domain. *Applied Artificial Intelligence*, 8:125–147, 1994.

[9] Ron Rymon. Search through systematic set enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[10] Jeffrey C. Schlimmer. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 284–290, 1993.

[11] Matthew D. Schmill, Tim Oates, and Paul R. Cohen. Tools for detecting dependencies in AI systems. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*, pages 148–155, February 1995.

[12] Padhraic Smyth and Rodney M. Goodman. An information theoretic approach to rule induction from databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):301–316, 1992.

[13] Geoffrey G. Towell, Jude W. Shavlik, and Michiel O. Noordeweir. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861–866, 1990.

[14] John W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, 1977.

[15] Geoffrey I. Webb. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:45–83, 1996.

[16] Thomas D. Wickens. *Multiway Contingency Tables Analysis for the Social Sciences.* Lawrence Erlbaum Associates, 1989.