A Distributed Approach to Finding Complex Dependencies in Data

Matthew D. Schmill

Computer Science Technical Report 98-13

Experimental Knowledge Systems Laboratory
Computer Science Department, Box 34610
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003-4610

Abstract

Learning complex dependencies from time series data is an important task; dependencies can be used to make predictions and characterize a source of data. We have developed Multi-Stream Dependency Detection (MSDD), a machine learning algorithm that detects complex dependencies in categorical time-series data. DMSDD attempts to balance the search for strong dependencies across a heterogeneous network of workstations. We develop a load balancing policy for DMSDD– first using only static techniques, and then adding in dynamic measures – on canonical machine learning datasets.

Contents

1 Introduction

Data mining is the process of revealing hidden structure in complex datasets by successive transformation and analysis. Data mining, a more modern term for what has historically been called exploratory data analysis [Tuk77], has come to comprise more than just mathematical, statistical, and visualization operations. More sophisticated automated techniques including pattern recognition, causal modeling, and decision tree induction have all been added into the mix to generate deeper, more informative analyses.

We are concerned with a particular data mining application called *time series analysis.* Time series are synchronized recordings of data sources as they change over time. Examples of this kind of data include economic indicators, distributed network status reports, and binned continuous streams such as flight recorder data. A successful data mining technique might elicit many useful details from such data. Perhaps there is a strong correlation in network logs between packet loss and a particular network route, or that your stock loses 2 points with unusual frequency following any type of press release from a competitor. These findings we might call *dependencies*, or rules that express an unexpectedly frequent occurrence of one pattern (called the *precursor*) by another (the *successor*) in the data. In general, dependencies take the form: "if pattern x is seen at time t, then at time $t + \delta$, pattern y will occur with probability p."

We have developed an algorithm called Multi-Stream Dependency Detection (MSDD) that searches for the strongest dependencies in multiple, synchronized streams of discrete time series data. Consider the following three streams:

S_1	**A**	C	F	B	L	**A**	A	B	B	B	B	C	**A**	F	F	L	B
S_2	V	V	W	**W**	V	X	X	X	**W**	X	Y	Y	W	V	X	**W**	X
S_3	**7**	3	1	**3**	1	**7**	6	6	**3**	5	3	5	**7**	5	5	**3**	1

The boldfaced tokens highlight a dependency between two patterns in these streams. The rule $(A\ *\ 7) \overset{3}{\Rightarrow} (*\ W\ 3)$ represents this dependency, which is observed three times in the data above. Specifically, this rule says, "When you see A in Stream 1 and 7 in Stream 3, on timestep t, then expect to see W in Stream 2 and 3 in Stream 3 on timestep $t + 3$." Note that some token values are irrelevant for predictive purposes; these are wildcarded in the rule. The number of timesteps between the onset of the precursor and successor patterns is called the *lag* of the rule.

MSDD finds the k strongest dependencies in a dataset by conducting a systematic search in the space of 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 [OGC95a, RSE94, Rym92, Sch93, Web96]. 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 into many computationally independent subsets, each of which may be processed in parallel. Distributed MSDD (DMSDD) is an implementation of MSDD designed to take advantage of this property by distributing the search among a network of computing resources.

Distributing MSDD's computational load across available computing resources could provide a clear saving in the time to completion by performing computations in parallel. MSDD makes use of an aggressive pruning heuristic, though, making the effective and efficient balancing of computations among processors challenging. The problem of *load balancing* is that of ensuring that the computational load is distributed such that all processing elements are maximally utilized for the duration of the search.

The remainder of the text is structured as follows: In section 2, we describe the MSDD algorithm in detail, along with some applications of the algorithm. Section 3 details the development of a comprehensive load balancing policy, and reports on their empirical performance. We finish with comments on the general applicability of MSDD and DMSDD , and recommendations on load balancing for dynamic search problems.

2 The MSDD Algorithm

MSDD finds the k strongest dependencies by executing a search through the space of dependencies defined by the training data. Training data is represented by m input streams, denoted $s_1, \ldots, s_m$, with each stream consisting of a series of categorical data points called *tokens*. We denote the set of all streams as $\mathcal{S}$, and the *alphabet* (set of possible tokens) for stream s_i as $\mathcal{V}_i$. For example, $\mathcal{V}_2 = \{V, W, X, Y\}$ in the data presented in section 1.

Recall the basic form of an MSDD dependency: "If an instance of pattern x occurs at time t, then an instance of pattern y will occur at time $t + \delta$." We shall denote these rules by $x \overset{\delta}{\Rightarrow} y$, where x is the *precursor*, and y is the *successor*. The patterns x and y are called *multitokens*, and are represented as parenthesized lists, where item i in the list representation of x represents the value of stream s_i for x. Alternatively, patterns may include *wildcard* tokens, denoted as $*$, to indicate that any stream value will match the wildcard. We denote the successor pattern for the sample rule in section 1 $(* \; W \; 3)$.

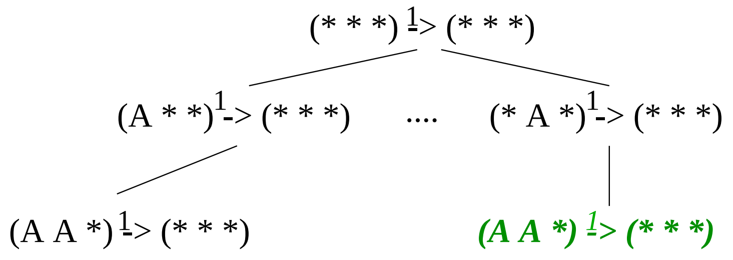

Figure 1: A sample MSDD search space. The highlighted node is redundant and will not be generated in the systematic expansion scheme.

The search space of MSDD is defined as the set of all dependencies possible given $\mathcal{S}$, $\mathcal{V}$, and δ. MSDD performs its search in a general-to-specific manner. That is, the search space is rooted at the rule where both precursor and successor consist completely of wildcards. The children of a rule r is the set of all dependencies generated by replacing a single wildcard in r with an actual token. The process of replacing a wildcard with a token value is called *token instantiation*, and is performed in a systematic fashion – a token may only be instantiated if there are only wildcards to the right of it in both the precursor and successor patterns. This simple rule of generation prevents rules from appearing redundantly in the search space. Consider the example space of figure 1. The rule $(A \; A \; *) \Rightarrow (* \; * \; *)$ highlighted at the lower-right portion of the tree is redundant with its sibling pictured to the left. The systematic expansion rule will not allow the highlighted child to be generated, though, because it would require instantiating to the left of the rightmost instantiated token in its parent.

The search proceeds, starting at the root, in breadth first fashion. The root node is systematically expanded, and each child rule is rated according to some evaluation function f. The current implementation of MSDD uses the G statistic, a measure of nonindependence, as f. The computation of G begins by counting negative and positive instances in the training data of the rule being rated. For a dependency $x \Rightarrow y$, MSDD generates the following 2×2 contingency table:

	X	$\overline{X}$
Y	n_1	n_2
$\overline{Y}$	n_3	n_4

The cells of this table 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. The G test is computed on this table 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 margin and table totals [Wic89]. Using the G test, MSDD can compare the apparent strength of dependencies found in the data, and consequently maintaining a heap of the k strongest dependencies is a simple task.

The G statistic has another property, though, that makes it a good choice for MSDD's evaluation function. Tim Oates has shown that an optimistic upper bound on G score for the descendents of a dependency can be established. The proof is omitted here for the sake of brevity, but can be found in [OC96b]. The upper bound on G is computed as follows:

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

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

This optimistic lower bound on G is the basis of a simple, but powerful pruning heuristic. The value $Gmax$ is computed for node under current consideration and compared against the last entry in the k best heap. If $Gmax$ is less than the G score of the last entry in the *k* best list, then there is no point in continuing with the current node. The search tree can then be pruned at that point. Experiments with real and artificial data show that the use of this *G*-based pruning heuristic results in very focused exploration, often exploring only tiny fractions of the complete space of dependencies [OSJC97]. The *k*-best MSDD algorithm is outlined in algorithm 2.1.

Algorithm 2.1 MSDD

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

2.1 Applications of MSDD

The MSDD algorithm has been applied to a number of domains, including pathology detection in a simulated shipping network [OGC95b] and learning the effects of planning operators for a simulated robot in a simple block painting task [OC96a]. These application serve as motivating examples of why dependency detection (and data mining in general) is a worthwhile process to engage in.

In the aforementioned applications, the goal is to find complex temporal dependencies in multiple streams of time series data. MSDD's approach to dependency detection is quite general, though, and is not limited to time series applications. MSDD learns dependencies among arbitrary pairs of multitokens; any problem that can be phrased in these terms can be tackled by MSDD. One class of problems that do not involve time series data, but can be represented as pairs of multitokens for the purpose of learning dependencies, is *machine classification.*

Machine classification is the process of assigning class labels to objects based their descriptive features. Examples of machine classification problems are predicting whether mushrooms are poisonous or edible based on their physical characteristics, predicting the party affiliation (democrat or republican) of a U.S. Representative based on their voting records, and predicting the class and number of solar flares given a window of previous activity. A machine classification algorithm is given a training set of tuples (A, c), where A is a vector of attribute values, and c is the class label of the tuple. Attributes in the mushroom dataset might include the diameter or color of the mushroom, and class label might be 0 if the object is edible and 1 if it is poisonous. Given the training set, the machine classification algorithm must build structures that will allow it to predict a class label c', given only a set of attribute values A'.

MSDD can be used as a tool for machine classification with good results [Oat94]. Each training instance is presented using A as the vector of precursor streams and c as the lone successor stream. MSDD then carries out its search using the training set to find the maximally predictive set of classification rules. MSDD might learn that the rule "if the mushroom is red and shiny, then it is poisonous" is accurate for 93% of the mushrooms it has seen. The generality of the MSDD approach allows machine classification to go one step further by being able to learn multiple class labels simultaneously. MSDD, for example, could learn that not only are red, shiny mushrooms poisonous, but that they cause nausea and hallucinations. A third extension of the machine classification task would be to learn attribute values that frequently occur together. If a mushroom is red, for example, it may almost always be shiny as well. In this task, the attribute vector A is presented as both the precursor and successor pattern for MSDD to learn on. We return to the task of detecting correlated attribute values as our primary means of evaluation in section 3.

3 Distributed Search

The systematic expansion of nodes performed by MSDD ensures that any two frontier nodes at a given time will be computationally independent. Further, the evaluation and expansion of a dependency only requires access to two data structures: the dataset, and the list of the k best dependencies. As a result, performing the search for dependencies in a distributed setting is a simple extension MSDD . Because computations are independent, no synchronization of calculations is necessary. Because access to the training data is read-only, it can simply be replicated at each of the hosts involved in the search. And finally, because an out-of-date k best list will only result in *underestimates* of the pruning threshold, the algorithm will not suffer a loss of admissibility if k-best lists are updated lazily.

The properties of the MSDD algorithm make a distributed implementation straightforward, and allows the more important problem of load balancing to be the focus of attention. After a short description of DMSDD's implementation details, we examine the problem of load balancing as it applies to DMSDD, and describe and evaluate some possible approaches.

3.1 DMSDD

DMSDD uses a centralized model of communication to coordinate its distributed search. The server acts as a hub for communication and user control, with one or more clients connecting via TCP/IP to offer their computational resources to the search. Simplified pseudocode of the DMSDD server is shown in algorithm 3.1.

n

The search begins and ends with the server, and all user controls over the search are mediated through the server. As shown in the pseudocode listing, the server is responsible for distributing the dataset to the client as well as performing the initial partitioning of the work to be distributed among the clients.

Algorithm 3.1 DMSDD Server

```
DMSDD-SERVER($\mathcal{S}$)
    1. SEND-DATA(clients)
    2. open = SYSTEMATICALLY-EXPAND(root − node)
    3. partitions = PARTITION(open, clients)
    4. SEND-PARTITIONS(partitions, clients)
    5. while open do
        a. HANDLE-MESSAGES()
        b. node = NEXT-NODE(open)
        c. children = SYSTEMATICALLY-EXPAND(node)
        d. children = REMOVE-PRUNABLE(children, best)
        e. add children to open
        f. for child in children do
            i. if LENGTH(best) $< k$ or $\exists n \in best$ s.t. $G(child, \mathcal{S}, \delta) > G(n, \mathcal{S}, \delta)$ then
                add child to best
                BROADCAST(best, clients)
            ii. if LENGTH(best) $> k$ then
                remove from best the node with the lowest $f$ value
    6. wait for clients to finish
    7. return best
```

A sketch of the client system is shown in algorithm 3.2. As with the server, the client algorithm is very similar to basic MSDD with some additional code for handling and sending messages. Indeed, both server and clients are essentially the basic MSDD algorithm operating on different partitions of the same search space. The messages passed between client and server come in four basic types:

- DATA messages contain portions of the dataset and are sent from the server to the client before the search begins.
- NODE messages are used to pass a node from one machine's agenda to another's. The function SEND-PARTITIONS in algorithm 3.1 generates messages of this type.
- BEST messages are used to indicate that a searcher has added a node to it's best list. These are sent to the server and broadcasted to all clients if the centralized k best list is updated.
- WAITING messages tell the server that a client's open list has become empty. If all clients have indicated that they are waiting, and the server's agenda is empty, the server declares that the search is finished.

3.2 Load Balancing

The major advantage to distributing the search for dependencies across multiple computing resources is obvious: ideally, a computation requiring γ milliseconds of computing time would take $\frac{\gamma}{n}$ milliseconds to complete on n machines. Due to message passing and other overhead,

Algorithm 3.2 DMSDD CLIENT

```
DMSDD-CLIENT()
    1. while *connected* do
    2. HANDLE-MESSAGES()
    3. if *open*
        a. *node* = NEXT-NODE(*open*)
        b. *children* = SYSTEMATICALLY-EXPAND(*node*)
        c. *children* = REMOVE-PRUNABLE(*children*, *best*)
        d. add *children* to *open*
        e. for *child* in *children* do
            i. if LENGTH(*best*) $< k$ or $\exists n \in best$ s.t. $G(child, \mathcal{S}, \delta) > G(n, \mathcal{S}, \delta)$ then
               add *child* to *best*
               SEND-TO-SERVER("*best*", *child*)
            ii. if LENGTH(*best*) $> k$ then
               remove from *best* the node with the lowest $f$ value
        f. if EMPTY(*open*)
            i. SEND-TO-SERVER("*waiting*")
```

this idealized speedup is simply not possible, but it is the goal of parallel and distributed computation to come as close to it as possible.

The most important factor behind the ultimate speedup of a distributed algorithm is ensuring that all available processors are doing work until the computation is complete. Ideal load balancing ensures this property in a distributed system. Our goal with DMSDD is to design such a load balancing scheme.

Some studies have been made of provably optimal load-balancing policies. Most, if not all such studies, such as [GRS95], require *a priori* knowledge about the structure of the search space. The MSDD search space can indeed be enumerated and reasoned about, but due to pruning, the *effective* search space (that space which is actually searched) cannot be determined a priori. For this reason, optimality results based on tree sweep procedures do not directly apply to DMSDD.

Many solutions to the load balancing problem have been proposed for problems for which optimality results do not apply, such as IDA* search [Coo96] and plasma reactor simulation [WRT96]. In general, these load balancing algorithms can be distinguished in two ways. The first distinction can be made based on *what* is partitioned (and subsequently distributed): the computational space, or the data. In *functional decomposition*, distinct computations are distributed among processing elements. In *data decomposition*, partitions of the data are distributed. With MSDD, the systematic nature of the search space allows disjoint sets of nodes to be evaluated independently. The same process if the data were partitioned would not allow the searchers to operate independently; every result generated by a host would need to be synchronized with every other host. As such, the logical approach to partitioning (and the one we take) with DMSDD is functional decomposition.

The second distinction among distributed algorithms is made between *static load balancing* and *dynamic load balancing*. Static load balancing attempts to divide the data prior to the beginning of the distributed computation. For DMSDD, static load balancing equates to dividing the first ply of the search space among the distributed processing elements in line

	Number of Streams	Training Instances	Size of First Ply	Nodes Expanded	CPU Time
Solar	13	1066	47	9,085	912,858msec
Chess	17	500	38	24,168	2,090,196msec

Table 1: Properties of the machine learning datasets used for evaluation.

3 of 3.1. Dynamic load balancing takes place while the search is in progress. An example of dynamic load balancing in DMSDD would be a processing element with a large agenda offloading some of its work to an element with few nodes on its agenda. Good static analysis can make dynamic load balancing unnecessary, reducing communication overhead and idle CPU cycles. The remainder of this section is devoted to developing and evaluating an effective approach to load balancing for MSDD.

3.2.1 Evaluation Criteria

The basic MSDD algorithm has been shown to be effective in terms of the quality of the rules it discovers [Oat94, OC96a, OSGC96] and efficient in its search of very large spaces [OSJC97]. Our goal in evaluating DMSDD is to ensure that effectiveness is not compromised and efficiency increases proportionally to the computing resources as they are added to the search.

Effectiveness is not compromised; the systematicity of the algorithm ensures that exactly the same set of dependencies is generated regardless of where the nodes are evaluated. Further, the expansion of extraneous nodes are generated as a result of propagation delays on the BEST messages will never effect admissibility of the search. Our evaluation efforts are primarily concerned, then, with efficiency gains bought by distributing the search and effective load balancing.

We measure performance gain (or loss) through four variables: the total number of *nodes expanded*, *CPU time*, *real time*, and *CPU utilization*. The number of nodes considered in the search is a raw measure computational expense. CPU time is measured in milliseconds as the sum of system and user time spent on behalf of MSDD, and is a measure of processing time independent of system load. Real time takes the system load into account; it is the real time duration of the search in milliseconds. Finally, CPU utilization measures the percentage of real time that the open list of a machine is non-empty. In our experiments, we record the mean CPU utilization across the nodes in a search as well as the minimum utilization.

The data used for evaluation can be found at the UCI machine learning repository. We chose two datasets: solar flare data, and chess endgame results. In each case, the attribute vectors were duplicated and presented as pairs to detect correlated attribute values as described in section 2.1. The value of k was set to 20 for all experiments reported here.

Table 1 shows some properties of the datasets along with results from applying basic MSDD to them. The column "size of first ply" reflects the number of children of the root node, and is the size of the work load that is used for static load balancing. The number of nodes expanded and CPU time are results for basic MSDD.

Machine	Processor	Capacity
`goddard`	`Alpha`	257$\frac{msec}{node}$
`lindy`	`Alpha`	1251$\frac{msec}{node}$
`earhart`	`Alpha`	1249$\frac{msec}{node}$
`gagarin`	`Alpha`	1370$\frac{msec}{node}$
`hinden`	`SPARC`	1550$\frac{msec}{node}$

Figure 2: An excerpt from DMSDD's capacity lookup table.

3.2.2 Static Load Balancing

Static load balancing is an approach to load balancing that attempts to evenly distribute work up front, by static analysis of the initial problem space. In the prior discussion of load balancing, the remark was made that optimality results do not directly apply to DMSDD. This is not to say that devoting effort to static load balancing is not a worthwhile task; however, one cannot expect optimal static load balancing for tasks that involve dynamic changes to the problem space.

The initial problem space DMSDD works with are the direct children of the root dependency. Lines 2 through 4 of algorithm 3.1 show this initial expansion, its partitioning, and delivery of initial workloads to the client searchers. The third step, a call to PARTITION, is where static analysis may take place and static load balancing begins.

A naive static load balancing policy makes no effort to balance the load in a disciplined way. This policy, which we will call the *naive* condition simply takes the set of dependencies, splits it into n equal sized chunks (where there are $n-1$ clients and one server), and distributes them. Essentially, the naive load balancing condition represents the minimum requirements of a load balancing policy, and we will consider it to be a baseline result.

The naive policy fails to take into account all but one piece of information, the size of the initial search space (the children of the root node), and even then does a poor job of using that information. Dividing the search space into even chunks assumes a homogeneous (in terms of processing capacity) group of workstations. For a distributed architecture, this is assumption can be costly. Our local network, for example, has machines running at three different speeds, including a SPARC10, three Alphas running at 2 to 3 times as fast, and one Alpha running 5 to 9 times faster than the others. To address this concern, we must revise the naive policy to take capacity into account. We will call this revision the *capacity sensitive* policy.

The capacity sensitive policy uses a database of known clients and architectures to access rough estimates of processing capacity. The capacity estimates in the database reflect the mean time in milliseconds to expand the root node of a known dataset. Example entries in the capacity database for the machines used in the experiments reported here are shown in table 2. Using these baseline estimates of processing speed, the capacity sensitive policy no longer needs to make the homogeneous architecture assumption, and can send workloads of sizes more appropriate to the recipient architecture.

The graphs in figures 3 and 4 show the effects of adding processors to the search of the

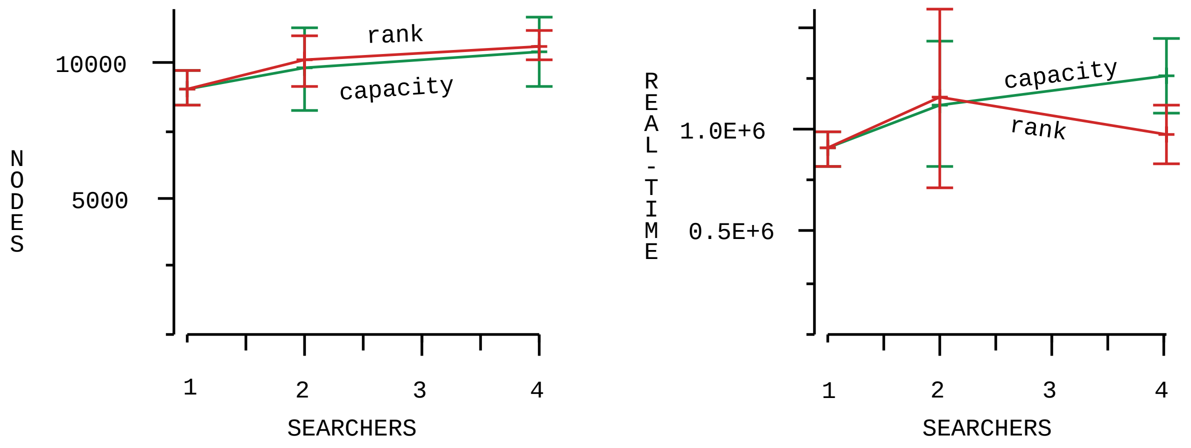

Figure 3: The effects of adding more workstations to the search of the solar flare dataset with the capacity sensitive and rank-capacity sensitive load balancing policies. Each data point is based on 5 samples and is shown with 90% confidence intervals. **(a)** the effect on the total number of nodes expanded **(b)** the effect on the time to completion for unloaded machines.

solar flare dataset. The plots labeled "capacity" correspond to DMSDD operating using only the capacity sensitive load balancing policy. Each data point represents the mean of five trials with 90% confidence intervals. In all cases, the machines were unloaded, and selected from the Alpha machines listed in figure 2.

The number of nodes expanded increases a small amount as a result of distributing the search. The effects are very small, and most likely due to the fact that the k-best list is subject to latency in updating. Because a BEST message may experience propagation delay, a small number of nodes may temporarily escape pruning. Time to completion, shown in figure 3b, behaves somewhat differently than expected, though. The time to completion actually increases as processors are added to the search!

One need only look to the graphs of figure 4 to get an indication of how the search could actually take longer when additional resources are available. As the nodes are added, overall CPU utilization declines, indicating that mistakes are being made in the static partitioning phase. The effect is most dramatic in graph 4b, which shows the minimum CPU utilization. In the case with 4 workstations, the capacity sensitive policy performs quite poorly, creating a partition that in the average case results in at least one machine operating for only 30% of the total search time. The result is that the total time to completion will be equal to the time taken by the *worst assignment of nodes to processor*. In the results shown here, the machine **goddard** was used in the single processor case, and is 4 or 5 times faster than any of the other alphas we used. By adding slower processors to the search, we allow DMSDD the opportunity to overallocate nodes to them. As figure 4b shows, as more processors are added to the search, the capacity sensitive load balancer is more probable to create an assignment of nodes that will be slower than goddard alone. Intuitively, there should be a saddle point at which the addition of processors overpowers this "slow as the slowest" effect, although it appears that that point is greater than four processors for the capacity sensitive load balancer.

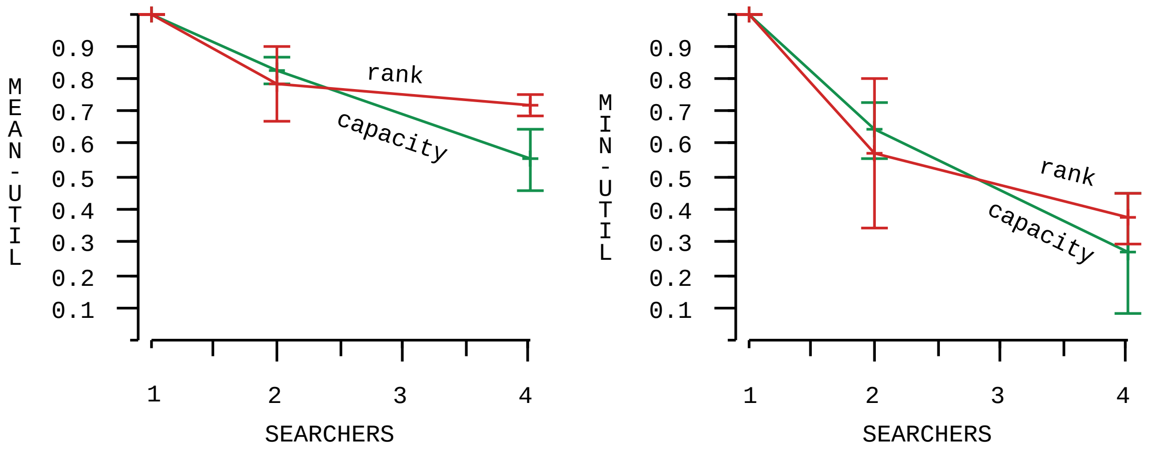

Figure 4: The effects of adding processors on CPU utilization for the solar flares dataset. **(a)** the effect on mean CPU utilization **(b)** the effect on the minimum CPU utilization

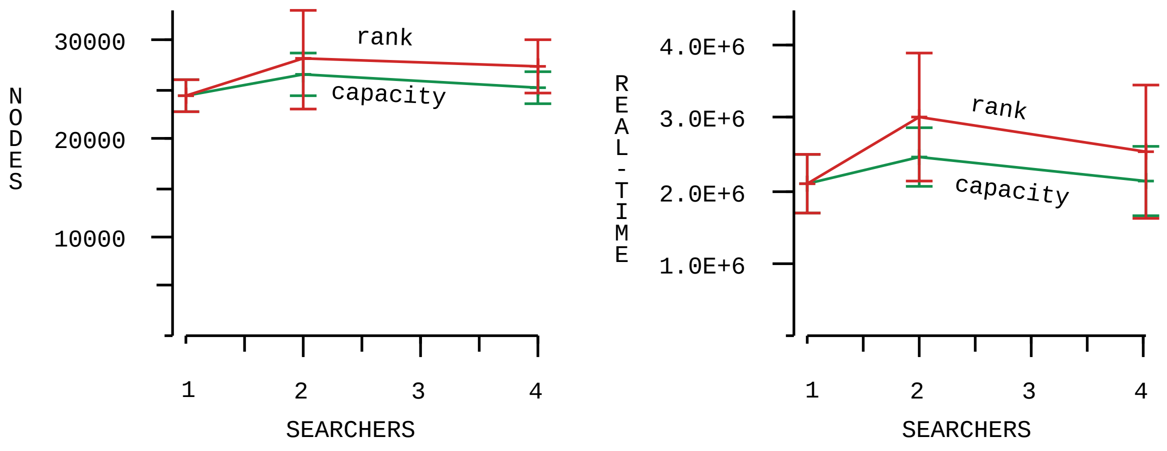

Figure 5: The effects of adding processors to the search of the chess endgame dataset. **(a)** the effect on the total number of nodes expanded **(b)** the effect on the time to completion for unloaded machines.

The graphs of figures 5 and 6 show similar effects for the chess endgame dataset, which has an effective search space roughly 2.5 times the size of the solar flares dataset. In these graphs, however, the "saddle point" is visible with only four processors. In the graphs of nodes expanded and real time, there is an increase from 1 to 2 processors, and then each appears to begin a decline. In both cases, this represents an effect of diminishing penalties; for nodes expanded, the latency effect as described for the solar flares data is overtaken by a different effect. Because there are additional searchers, greater breadth of search can be achieved in less time, making BEST messages available earlier than in the 1 and two node cases, allowing for more aggressive pruning earlier. As the graphs in figures 4 and 6 show the same effect, it seems likely that the saddle point in real time is reflection of what is going on with the number of nodes expanded.

The capacity sensitive policy seems far from ideal. Our main goal of maximizing mean CPU utilization has not been accomplished. One reason for this is likely that the capacity

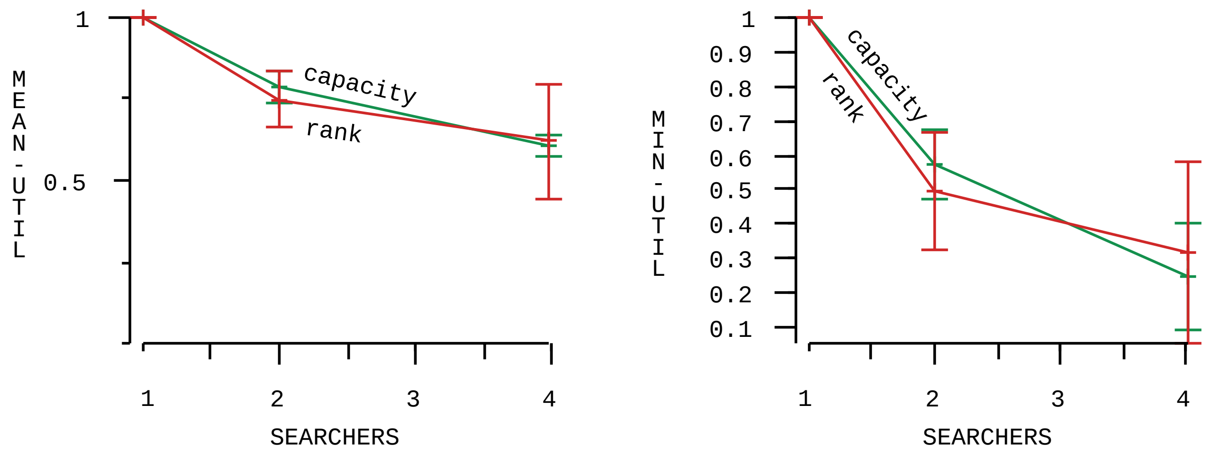

Figure 6: The effects of adding processors on CPU utilization for the chess endgame dataset. **(a)** the effect on mean CPU utilization **(b)** the effect on the minimum CPU utilization

sensitive policy does not exploit any knowledge about the structure of the state space. Of particular interest is the fact that MSDD's systematic expansion algorithm, in which children are generated by instantiating a token to the right of the righmost instantiated token, results in a heavily skewed search tree. Consider the following two rules:

$$(*\ X\ *) \Rightarrow (*\ *\ *)$$
$$(*\ *\ *) \Rightarrow (*\ *\ 6)$$

Both rules reside one level deep into the search space, but the first rule can be parent to $\sum_{i=3}^{6} |\mathcal{V}_i|$ children, while the second rule will parent none at all. We will call the number of uninstantiated tokens to the right of the righmost instantiated token for a rule its *rank*. The rank of the above two rules are 4 and 0, respectively.

Rank can be used to compute the size of the unpruned search space starting at rule r. The calculation is a recursive summation, and is a function of the rule r's rank, which is x, and the number of streams m, in the following formula:

$$\textit{spacesize}(x) = \sum_{i=x}^{m}(|\mathcal{V}_i| + (|\mathcal{V}_i| \times \textit{spacesize}(x+1)))$$

Spacesize computes the maximum amount of work a searcher would have to as a result of being assigned rule r to expand as part of its workload. Certainly, the maximum amount of work a searcher could have to do is much different than the work a search *will* do on most datasets. Rank and spacesize, however, are statistics that can be computed *a priori*, while effective spacesize is not.

DMSDD's static load balancing policy might attempt to balance the total rank or spacesize it allocates to different searchers. Such policies we will call *rank-based*. Our implementation of rank-based load balancing uses a fast estimate of spacesize by estimating each $\mathcal{V}_i$ to be 2. As a result, estimated spacesize is equal to $2^{m-x} - 1$.

The plots labeled "rank" of figures 3- 6 show the effects of adding processors to the search with rank-based load balancing. The performance of rank-based load balancing appears to scale slightly better than the capacity-only policy for the solar flares set. In the case of four processors, the rank based scheme shows an improvement in CPU utilization of roughly 15% over capacity-only load balancing, and has less variance in the results. Results for CPU utilization trials are not so convincing, as rank based load balancing appears to increase the minimum utilization, but perform worse or similarly to capacity based balancing in all other measures. The problem here appears to be related to the location of the k best rules in the unpruned search space. The working assumption of the rank-based policy is that rules are uniformly distributed across the working search space, and it appears that the chess dataset violates that assumption.

Overall, static load balancing does not appear to be feasible as a stand alone load balancing policy. Both policies fail to use one last available piece of data, the G-max estimate of the initial search space. This requires further investigation, but as heuristics are added to the static load balancing policy, it becomes less a priori than dynamic. To be sure, neither capacity nor capacity-rank based balancing alone are sufficient to do an effective job.

3.2.3 Dynamic Load Balancing

The major fault of static load balancing is that with DMSDD, the information useful in load balancing appears as the search progresses. Before any nodes are rated and the k-best list starts filling out, DMSDD has little information to base its work assignments on. Mistakes appear to be imminent, and static load balancing offers no advice on what to do once mistakes have been made. One solution to this problem is to allow processors to correct the mistakes of the static policy by dynamically rebalancing their workloads.

Dynamic load balancing schemes are a class of algorithms that perform load balancing after the work has already begun. For DMSDD, dynamic load balancing begins when a client detects that its agenda is about to become empty. In such a situation, the client sends a new type of message to the server to indicate that it can take on more work. This is referred to as *receiver initiated* load balancing, as the eventual recipient initiates the transfer of work.

- A WAITING message indicates that a client's agenda is empty, and it is able to take on more work.

When the server receives a WAITING message, it first checks its own agenda to see if there is enough work there to offload some nodes. If there is, the server invokes its static load balancing policy to rebalance its load with respect to the client. If the server does not have enough nodes to offload to a waiting client, or its own agenda becomes empty, it broadcasts a RECALL message to all working clients.

- A RECALL message advertises to the clients that the server is willing to recall some work. Any client with more than some threshold of nodes on its agenda can respond by rebalancing its load with respect to the waiting client (or server). Offloaded nodes are sent to the server, who either keeps them or supplies them to a waiting client.

The message passing associated with dynamic rebalancing also provides an opportunity to obtain more up-to-date information for use in load balancing. In particular, by the time a searcher has expended its agenda, it will have new estimates of its own processing capacity. We introduce the CAPACITY message as a way for clients to update the server's capacity estimates dynamically.

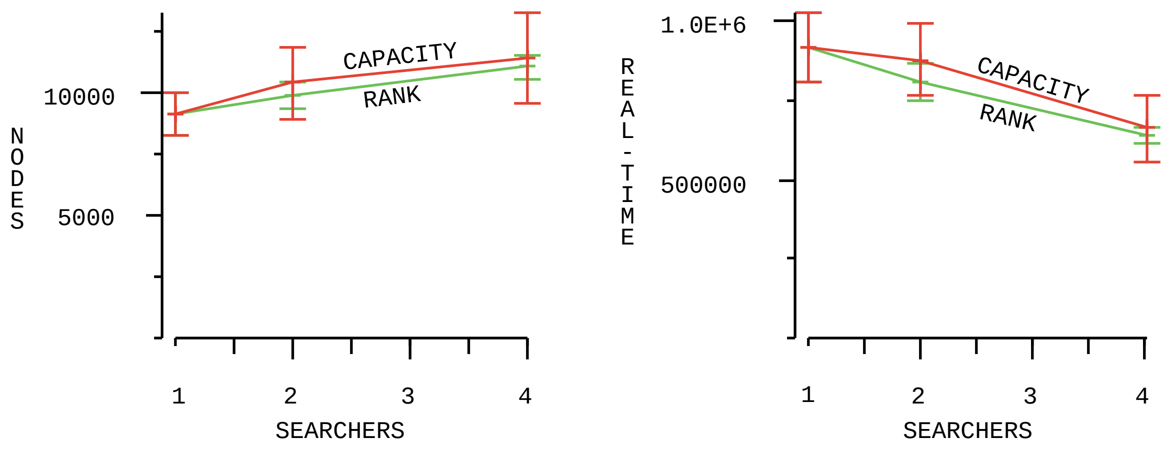

Figure 7: Results for the solar flares data after adding dynamic load balancing to DMSDD. Each data point is based on 5 trials and is shown with 90% confidence intervals. **(a)** the effect on the total number of nodes expanded **(b)** the effect on the time to completion for unloaded machines.

- The CAPACITY message communicates the effective number of nodes per second a client is processing during the current search.

Performance results with dynamic load balancing enabled are shown in figures 7- 10. Not surprisingly, the number of nodes expanded continue to show small, but noticeable increases as a result of adding processors. The graphs of CPU utilization, though, show the effect that we had hoped for. For both the solar flares and chess datasets, mean CPU utilization increased dramatically to the 90-95% range. The minimum CPU utilization, not shown, showed similar improvements, in most cases hovering around 80-95%. As a result, the mean completion time decreases in an apparently linear fashion as processors are added to the search. Recall that the machine goddard, an alpha 4 to 5 times faster than the others, was used in the single processor case. In the ideal case, then, the performance increase would be somewhere between 160 and 175 percent. With the rank based load policy, the mean speedup in our trials was 162% for the solar flares and 143% on the chess data. Thus, the dynamic load balancing scheme seems to be working as advertised: it is achieving high levels of utilization despite the relatively poor processor scheduling being done by the static policies.

The poor performance of the static load balancing policies does have downstream effects, however, that are most apparent in figures 8b and 10b. As the number of processors increase, the number of network messages increases proportionally. On our local network, latency is low, and this caused only minor degradation in system utilization. In a heavily loaded medium, though, where latencies can be large, the relationship between the number of searchers and the number of messages sent could result in longer idle periods and a reduction in scalability.

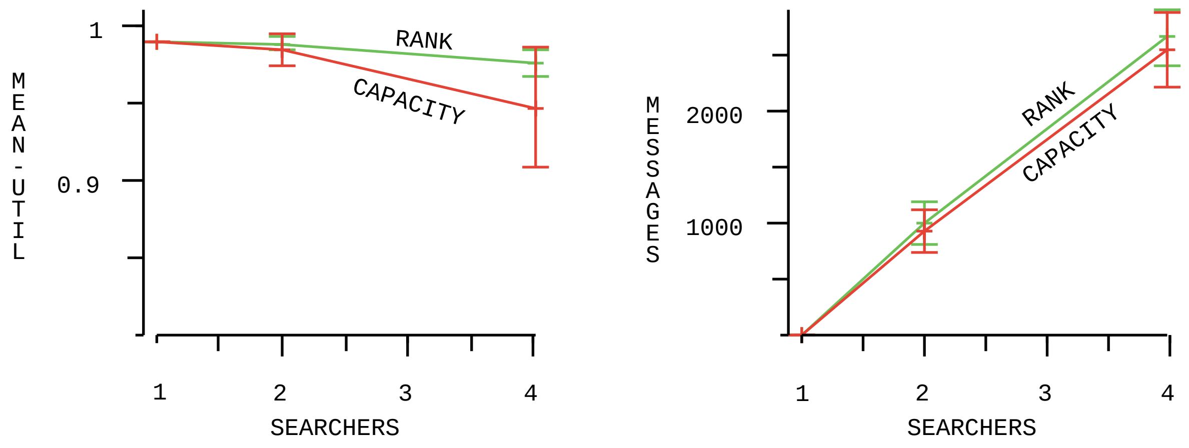

Figure 8: More results for the solar flares set with dynamic load balancing turned on. **(a)** the effect on mean CPU utilization **(b)** the effect on the total number of messages sent.

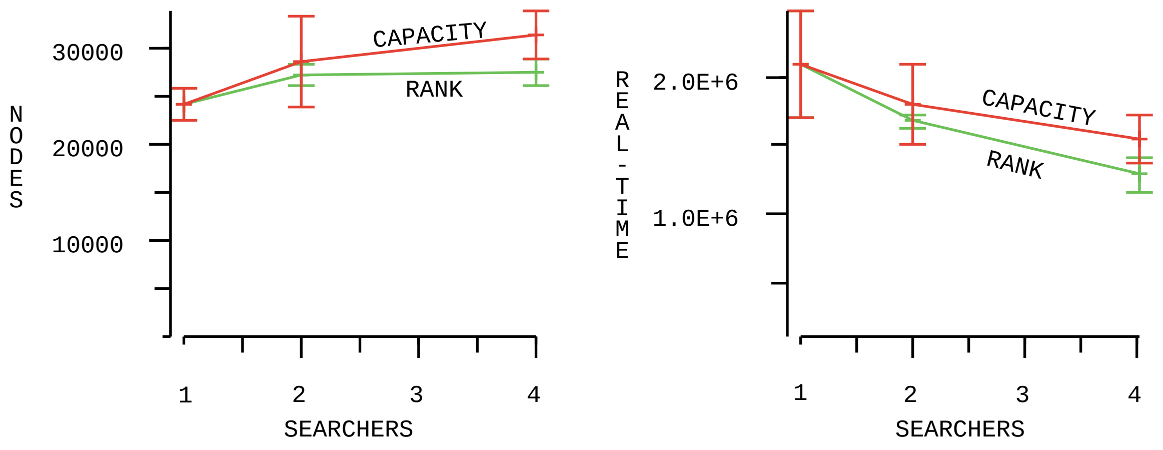

Figure 9: Results for the chess data after adding dynamic load balancing to DMSDD. **(a)** the effect on the total number of nodes expanded **(b)** the effect on the time to completion for unloaded machines.

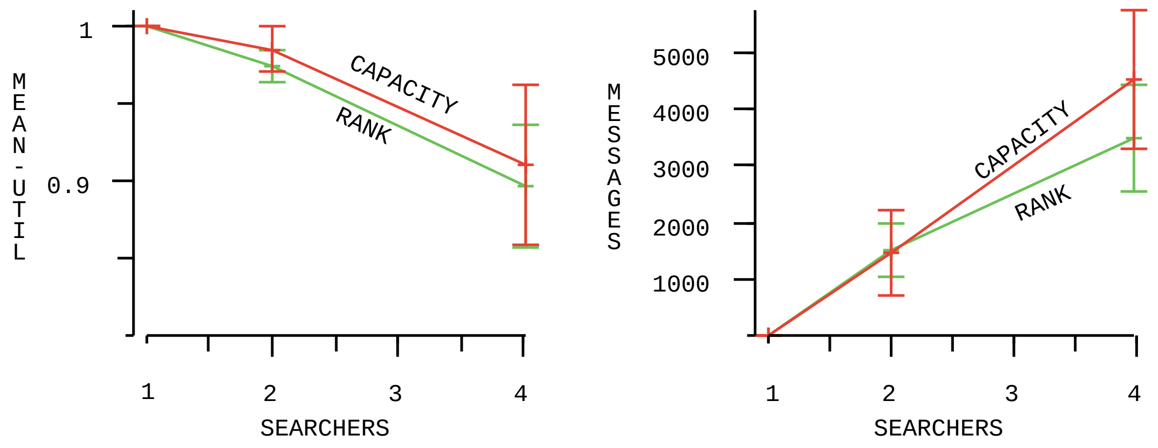

Figure 10: CPU utilization and message passing results for the chess set with dynamic load balancing turned on. **(a)** the effect on mean CPU utilization **(b)** the effect on the total number of messages sent.

4 Conclusions

We presented MSDD, an algorithm that searches for strong dependencies in multivariate, categorical data. By limiting the scope of the search to a set of k best dependencies, and evaluating nodes with G statistic, a measure with a provable upper bound for the children of a rule, MSDD can perform substantial pruning that makes the search tractable in very large spaces. MSDD's systematic expansion procedure also ensures that fringe nodes are computationally independent of one another. As a result, a distributed implementation of MSDD is a straightforward extension of the algorithm.

The dynamic nature of MSDD's search presents a challenge to effective load balancing. A priori analysis of the search space can give little indication of what parts will actually need to be searched and which will be pruned. As a result, our efforts to produce static load balancing policies were quite ineffective as standalone components. In fact, if slower processors were added to the search, the total time to completion was observed to actually increase in some cases even though the total processing capacity increased. As a result, it was determined that dynamic load balancing was necessary to recover from the misallocations that the static policies were invariably making. The dynamic policy was a significant improvement, exhibiting high levels of CPU utilization and linear speedups as processors were added. The dynamic policy also generated a number of messages that increased at a linear rate with respect to the number of processing elements, though. This result indicates that speedup and scalability could be reduced significantly for large numbers of processors or loaded networks. These conditions will be the topic of future consideration.

References

[Coo96] Diane J. Cook. A hybrid approach to improving the performance of parallel search. In J. Geller, editor, *Parallel Processing for Artificial Intelligence*. Elsevier Science Publishers, 1996.

[GRS95] Li-Xin Gao, Arnold L. Rosenberg, and Ramesh K. Sitaraman. Optimal architecture-independent scheduling of fine-grain tree-sweep computations. In *7th IEEE Symposium on Parallel and Distributed Processing*, pages 620–629, 1995.

[Oat94] Tim Oates. MSDD as a tool for classification. EKSL Memorandum 94-29. Department of Computer Science, University of Massachusetts at Amherst, 1994.

[OC96a] Tim Oates and Paul R. Cohen. Learning planning operators with conditional and probabilistic effects. In *Proceedings of the AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems*, pages 86–94, 1996.

[OC96b] Tim Oates and Paul R. Cohen. Searching for structure in multiple streams of data. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 346–354. Morgan Kaufmann Publishers, Inc., 1996.

[OGC95a] Tim Oates, Dawn Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In *Preliminary Papers of the Fifth International Workshop on AI and Statistics* [OGC95b], pages 417–423. Also available as University of Massachusetts Computer Science Department Technical Report 94-81.

[OGC95b] Tim Oates, Dawn Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In *Preliminary Papers of the Fifth International Workshop on AI and Statistics*, pages 417–423, 1995. Also available as University of Massachusetts Computer Science Department Technical Report 94-81.

[OSGC96] Tim Oates, Matthew D. Schmill, Dawn E. Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In D. Fisher and H. Lenz, editors, *Learning from Data: Artificial Intelligence and Statistics*, pages 185–195. Springer Verlag, Inc., New York, 1996.

[OSJC97] Tim Oates, Matthew D. Schmill, David Jensen, and Paul R. Cohen. A family of algorithms for finding temporal structure in data. In *Preliminary Papers of the Sixth International Workshop on Artificial Intelligence and Statistics*, pages 371–378, 1997.

[RSE94] 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.

[Rym92] Ron Rymon. Search through systematic set enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[Sch93] 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.

[Tuk77] John W. Tukey. *Exploratory Data Analysis.* Addison-Wesley Publishing, Co., Reading, MA, 1977.

[Web96] Geoffrey I. Webb. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:45–83, 1996.

[Wic89] Thomas D. Wickens. *Multiway Contingency Tables Analysis for the Social Sciences.* Lawrence Erlbaum Associates, 1989.

[WRT96] Jerrell Watts, Marc Rieffel, and Stephen Taylor. Practical dynamic load balancing for irregular problems. In *Parallel Algorithms for Irregularly Structured Problems: IRREGULAR '96 Proceedings*, pages 299–306, 1996.