Efficient Mining of Statistical Dependencies

Tim Oates, Matthew D. Schmill and Paul R. Cohen
Experimental Knowledge Systems Laboratory
Department of Computer Science
Box 34610 LGRC
University of Massachusetts
Amherst, MA 01003-4610
{oates, schmill, cohen}@cs.umass.edu

Abstract

The Multi-Stream Dependency Detection algorithm finds rules that capture statistical dependencies between patterns in multivariate time series of categorical data [Oates and Cohen, 1996c]. Rule strength is measured by the G statistic [Wickens, 1989], and an upper bound on the value of G for the descendants of a node allows MSDD's search space to be pruned. However, in the worst case, the algorithm will explore exponentially many rules. This paper presents and empirically evaluates two ways of addressing this problem. The first is a set of three methods for reducing the size of MSDD's search space based on information collected during the search process. Second, we discuss an implementation of MSDD that distributes its computations over multiple machines on a network.

1 Introduction

Multi-Stream Dependency Detection (MSDD) is an algorithm for finding rules that capture statistical dependencies in databases. Past applications of the algorithm include finding dependencies in multi-variate time series [Oates and Cohen, 1996c], learning probabilistic planning operators [Oates and Cohen, 1996b], and acquiring rules for correlating and predicting asynchronous events [Oates *et al.*, 1998]. In this paper, we describe three methods for reducing the size of the search space that MSDD considers and empirically evaluate their utility. In addition, we present a version of the algorithm, called D-MSDD, that distributes the search for rules over multiple machines on a network. The remainder of this section discusses the core MSDD algorithm. Section 2 describes the three search space reduction methods and section 3 summarizes our empirical work with them. Section 4 presents D-MSDD. Finally, section 5 summarizes.

Let D be a database containing T *records*: $D = \{R_1, \ldots, R_T\}$. Each record is a set of unique tokens taken from the alphabet Σ, and the number of tokens may vary from record to record: $R_i = \{\sigma_1, \ldots, \sigma_{n_i} | \sigma_j \in \Sigma, 0 \leq n_i \leq |\Sigma|\}$. Let a *pattern* be defined in exactly the same manner as a record. We say that pattern p occurs in record R if $p \cap R = p$. A *rule* (also called a dependency) consists of a pair of patterns, p and s, which are called the *precursor* and *successor* respectively.

For any given rule, we can construct a 2x2 contingency table that describes the frequency of co-occurrence of the corresponding patterns in a database. Let COUNT(p, s) denote the number of records in D that contain both p and s, i.e. for which $(p \cap R = p) \wedge (s \cap R = s)$. If either of the arguments to COUNT is negated, then that argument must not appear in the records. For example, COUNT$(p, \overline{s})$ denotes the number of records in D that contain p but do not contain s, i.e. for which $(p \cap R = p) \wedge (s \cap R \neq s)$. The following contingency table describes the frequency of co-occurrence of p and s:

	s	$\overline{s}$	Totals
p	n_1	n_2	r_1
$\overline{p}$	n_3	n_4	r_2
Totals	c_1	c_2	T

In the table above, $n_1 = \text{COUNT}(p, s)$, $n_2 = \text{COUNT}(p, \overline{s})$, $n_3 = \text{COUNT}(\overline{p}, s)$ and $n_4 = \text{COUNT}(\overline{p}, \overline{s})$. Also, $r_1 = n_1 + n_2$, $r_2 = n_3 + n_4$, $c_1 = n_1 + n_3$, $c_2 = n_2 + n_4$, and $T = n_1 + n_2 + n_3 + n_4$.

Rule strength is measured by the G statistic. G is a statistical measure of association, with large values indicating that p and s co-occur more or less frequently than one would expect by random chance [Wickens, 1989]. G is computed for the table above 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 row margins and the table total. For example, $\hat{n}_1$ is the probability that p and s will co-occur in a database record, given that they are independent, times the number of records in the database. The probability of an occurrence of p is r_1/T, the probability of an occurrence of s is c_1/T, and the probability of the joint event given independence is $(r_1/T)(c_1/T)$. Therefore, $\hat{n}_i = (r_1/T)(c_1/T)T = r_1 c_1 / T$. Strong dependencies, as indicated by large G

values, capture structure in the database because they tell us that there is a relationship between their constituent patterns, that occurrences of those patterns are not independent.

MSDD performs a general-to-specific search in the space of all possible pairs of patterns defined over Σ and returns the k strongest dependencies found, where k is supplied by the user. It is perhaps more correct to say that MSDD returns all of the rules associated with the top k values of G found in the search space. The current version of MSDD, in contrast to all other implementations reported previously, may return a number of rules much larger than k. The reason is that if multiple rules have exactly the same G value they are all retained, regardless of how different the rules themselves are.

The root of MSDD's search tree contains two empty patterns, $\{\} \Rightarrow \{\}$, and all rules at depth n have the property that $|p| + |s| = n$. The children of a node are generated by adding a token from Σ to either the precursor or the successor of that node. Despite the fact that the current implementation of MSDD is highly optimized C code which is capable of processing more than 100,000 rules per second (on some databases), the size of the search space is exponential in $|\Sigma|$ and simply cannot be explored exhaustively. However, because MSDD returns a list of the k strongest dependencies, it is possible to use an upper bound on the value of G for a rule's descendants to prune the search. If none of the descendants of a rule can have a G value higher than that of any of the current k best rules, then that rule can be pruned. In previous work we derived such an upper bound on G, making it possible for MSDD to find the k strongest dependencies in an exponential space [Oates and Cohen, 1996c].

To ensure that good rules are found early in the search, and thus that pruning becomes effective early as well, MSDD performs iterative deepening. This also causes the algorithm's memory requirements to be relatively meager.

2 Improving Search Efficiency

The first method for reducing the number of rules that MSDD searches is based on the observation that the size of the fringe as a function of search depth often has the shape shown in Figure 1. For this dataset (the `vote` dataset taken from the UC Irvine collection), the number of fringe nodes initially rises sharply, reaching a peak of 7,304,048 nodes at depth nine, and then falls off just as sharply. There are two competing forces at work causing this behavior: the size of the search space and MSDD's ability to prune. As search depth increases, the size of the search space grows dramatically. At shallow depths few good rules are found and MSDD's pruning cannot stop this growth. Eventually, though, a depth is reached at which most of the k best rules have been found, and the vast majority of the rules at subsequent depths have low G values due to the presence of extraneous tokens and can be pruned. To take advantage of this phenomenon, MSDD stops performing iterative deepening when the size of the fringe falls below a user-specified fraction of its maximum size and instead performs a single depth-first search with no depth limitation. For the search shown in Figure 1, the fact that the size of the fringe has peaked can be determined at the end of the search to depth ten, resulting in one final iteration of depth first search rather than the five iterations required by iterative deepening to terminate at depth fifteen.

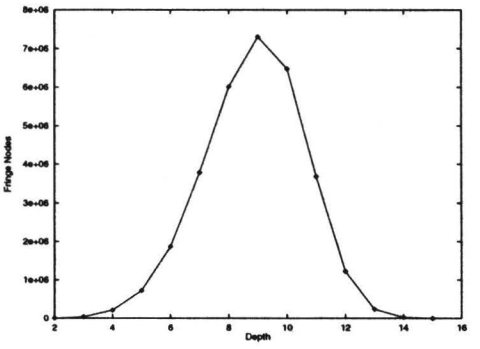

Figure 1: The number of fringe nodes as a function of search depth on the `vote` dataset.

The second method for reducing the number of rules that MSDD considers involves reordering the elements of Σ at each new depth. As with most implementations of depth-first search, search operators are applied in a fixed order to generate the children of a node. In the case of MSDD, search operators add an element of Σ to a rule to generate a child, and operators are ordered so that the operator that adds σ_i is applied before the one that adds σ_j for $i < j$. The result is that children for which the last token added has a lower index in Σ are expanded before children for which that index is higher. If the tokens in Σ are ordered so that the ones with low indices are the ones that appear in rules with high G values, then those rules will be generated early and pruning becomes more effective early. Under the assumption that tokens that appeared in the k best rules at depth d will continue to be the most useful tokens at depth $d+1$, we reorder the tokens in Σ after each search depth so that the most frequently used tokens have the lowest indices in Σ.

The final performance enhancement takes advantage of the fact that the size of MSDD's search space is exponential in $|\Sigma|$. Let Φ be the set of tokens that appear at least once in the final set of the k best rules. If Φ could be determined a priori, then the other tokens could be eliminated from Σ, leading to a potentially large reduction in the size of the search space. Although it is impossible to determine Φ prior to searching, it is often possible to identify that set by iteratively looking at subsets of Σ. Even when Σ is very large, exhaustive search to a shallow depth is feasible. The tokens that appear in the k best rules found during that search serve as an initial approximation to Φ, which we will denote Φ'. Hereafter, all searches are performed using Φ' as the token set rather

than Σ. After the initial shallow search, the following procedure is repeated some fixed number of times:

1. Perform a depth-unlimited run of the standard iterative deepening search (using Φ').
2. Remove all variables in Φ' that do not appear in any of the k best rules.
3. Add some small number of randomly selected variables in $\Sigma - \Phi'$ to Φ'.

The search in step 1 is very fast because $|\Phi'|$ is typically much smaller than $|\Sigma|$, making it possible to perform many iterations of this procedure in the time normally required to perform one search on the space defined by Σ. Unless the true k best rules individually contain very large numbers of tokens and rules containing subsets of those tokens all have low G scores, the procedure outlined above will eventually converge on Φ. There is no guarantee of such convergence in practice, especially when the number of iterations through the procedure is small. However, empirical results show that it performs quite well both in terms of CPU requirements and ability to find a good approximation to the set of rules found when searching over Σ.

3 Experiments

To evaluate the utility of the performance enhancements described above, we ran MSDD on several datasets taken from the UC Irvine collection. For each dataset, we ran MSDD five times in each of the conditions below, with each iteration using a different random ordering of the variables:

STANDARD - no performance enhancements turned on

FRINGE - switching from iterative deepening to depth-unlimited DFS when the fringe falls below 75% of its maximum size

ORDER - reordering Σ after each search

SAMPLE - five iterations of random sampling of five tokens in $\Sigma - \Phi'$

ALL - applying each of FRINGE, ORDER and SAMPLE at the same time

All experiments were run on a 500MHz DEC Alpha, and in each case $k = 10$.

Results for six different datasets taken from the UC Irvine repository are shown below. Table 1 lists those datasets along with the number of records and the number of unique tokens ($|\Sigma|$) they contain. For each dataset, Table 2 shows the mean number of CPU seconds and search nodes required to find the k best rules. In addition, the number of rules returned and their mean length (as measured by the sum of the number of tokens in their constituent patterns) is reported. Finally, the *disparity* between the true k best rules and the actual rules found is shown, where disparity is the mean number of tokens by which each rule in the true rule set differs from its best match in the actual rule set returned. Note that the STANDARD, ORDER and FRINGE conditions are all guaranteed to find the optimal rule set, and thus always have a disparity of zero.

Dataset	Records	Unique Tokens
flare	323	45
lymphography	148	64
mushroom	8124	128
promoters	106	230
tic-tac-toe	958	29
vote	435	45

Table 1: Datasets and their features that are relevant to MSDD's run time.

On the vote dataset, STANDARD expanded nearly one quarter of a billion nodes on average, requiring a little over one hour of CPU time. Interestingly, the amount of search required by ORDER and STANDARD were virtually identical on this dataset. However, compared to STANDARD, FRINGE was more efficient by a factor of two, and both SAMPLE and ALL were more efficient by an order of magnitude. t tests comparing mean CPU seconds and mean nodes expanded confirm these results, with all differences being highly significant. Despite the fact that SAMPLE and ALL are not guaranteed to find the same rule set as the other conditions, they did so unfailingly (i.e. had a disparity of zero).

On the promoters dataset, STANDARD expanded almost 300 million nodes on average, requiring a little under 23 minutes of CPU time. Each of ORDER, FRINGE, SAMPLE and ALL were significantly more efficient (as indicated by t tests) than STANDARD. Again, ALL was the least expensive condition, requiring 1/6 the amount of search required by STANDARD. Although the rules found in the SAMPLE and ALL conditions differed from the rules found in the other conditions on this dataset, there was substantial overlap. The mean G of the rules returned in the former two conditions was 63.09 and 64.37 respectively, whereas the mean G of the true k best rules was 68.25. On average, the rules returned in the SAMPLE and ALL conditions differed from the optimal rule set by 2.5 and 3.23 tokens respectively.

Results for the lymphography dataset show similar results, with significant speedups and low disparity (less than one token per rule). However, the results are rather different for flare, mushroom and tic-tac-toe. In each of these datasets, the ORDER condition was virtually identical to the STANDARD condition, with FRINGE performing significantly better. In each case, though, both SAMPLE and ALL performed significantly worse. The reason for this is made obvious by inspecting the final rule sets. Almost every token in Σ appears in the list of rules returned. Therefore, $|\Phi'|$ rapidly approaches $|\Sigma|$, causing each iteration of the sampling procedure to be almost as costly as a single run of STANDARD. Future work will include attempting to detect this condition as the search proceeds so as to stop iterating through the sampling procedure.

4 Distributed Search

Expansion of a node by MSDD requires knowledge of the node itself (to determine which search operators are

Dataset	Condition	CPU Seconds	Nodes	# Rules	Length	Disparity
flare	STANDARD	57.90	3,359,849	912.00	5.09	0
	ORDER	57.87	3,346,089	912.00	5.09	0
	FRINGE	29.86	1,836,863	912.00	5.09	0
	SAMPLE	287.5	16,581,715	912.00	5.09	0
	ALL	146.64	8,949,947	912.00	5.09	0
lymphography	STANDARD	18125.34	2,871,877,000	66.00	5.48	0
	ORDER	18191.70	2,869,929,700	66.00	5.48	0
	FRINGE	7898.37	1,311,838,200	66.00	5.48	0
	SAMPLE	3506.80	451,658,020	55.33	5.11	0.91
	ALL	4346.57	634,634,100	62.67	5.32	0.42
mushroom	STANDARD	18,809.59	189,895,730	9402.00	9.50	0
	ORDER	18,962.20	189,547,940	9402.00	9.50	0
	FRINGE	10,225.68	105,555,256	9402.00	9.50	0
	SAMPLE	44,523.57	393,258,140	9402.00	9.50	0
	ALL	17,984.33	172,908,180	7554.00	9.33	0.25
promoters	STANDARD	1364.32	297,523,940	418.00	7.67	0
	ORDER	1257.40	271,428,540	418.00	7.67	0
	FRINGE	621.92	142,537,070	418.00	7.67	0
	SAMPLE	421.72	72,410,270	274.00	7.57	2.50
	ALL	214.47	39,952,100	101.60	7.12	3.23
tic-tac-toe	STANDARD	673.97	21,553,110	60	4.00	0
	ORDER	673.90	21,553,088	60	4.00	0
	FRINGE	499.11	16,173,695	60	4.00	0
	SAMPLE	3370.47	107,787,840	60	4.00	0
	ALL	2495.93	80,890,920	60	4.00	0
vote	STANDARD	4005.03	225,861,500	20	3.3	0
	ORDER	3999.81	225,844,530	20	3.3	0
	FRINGE	2228.59	131,412,384	20	3.3	0
	SAMPLE	269.21	12,648,121	20	3.3	0
	ALL	144.63	7,782,805	20	3.3	0

Table 2: Performance of MSDD and its various enhancements on several datasets.

valid), Σ (the list of all search operators), the dataset (to build contingency tables), and the current list of the k best nodes (to prune). Only the list of the k best nodes changes dynamically during the search, making it possible to distribute the search space over multiple machines on a network as long as those machines have access to the same k best list. However, an out-of-date k best list will only result in *underestimates* of the pruning threshold, so the algorithm will not suffer a loss of admissibility if local copies of the k-best list are updated lazily.

D-MSDD uses a centralized model of communication to coordinate its distributed search. A single 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.

The distributed search begins with the server initiating the distribution of data. Once complete, the server expands the root of the search space to generate a single ply of the search space, and distributes it among the connected clients. Work can begin at a searching machine as soon as there are nodes to be evaluated, and continues until all participating searchers report that they have processed their entire workloads. During the search, should a participant decide to add a rule to it's k-best list, the rule and its rating are broadcast to all of the other participants.

The major advantage of distributing the search for dependencies across multiple computing resources is obvious: in the ideal case, 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, this idealized speedup is difficult to obtain, but it is the goal of parallel and distributed computation to come as close to it as possible. The key to realizing this goal is to keep all of the distributed resources as busy as possible while reducing message passing and other overhead to a minimum.

4.1 Load Balancing

Some studies have been made of provably optimal load-balancing policies. Most, if not all such studies, such as [Gao *et al.*, 1995], 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 D-MSDD.

Many solutions to the load balancing problem have been proposed for problems for which optimality results do not apply, such as IDA* search [Cook, 1996]. 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 portions of the computational space are distributed among processing elements. In *data decomposition*, the data is 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 D-MSDD 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 D-MSDD, static load balancing equates to dividing the first ply of the search space among the distributed processing elements. Dynamic load balancing takes place while the search is in progress. An example of dynamic load balancing in D-MSDD would be a processor with a large agenda offloading some of its work to a processor with few nodes on its agenda. Good static analysis can make dynamic load balancing unnecessary, reducing communication overhead and idle CPU cycles.

We explored the utility of two static load balancing policies in D-MSDD. A *capacity sensitive* policy ensures that each searcher receives a number of nodes proportional to its processing capacity by consulting a database of known clients and architectures containing estimates of their processing capacity. The capacity estimates in the database reflect the mean number of nodes expanded per second over a fixed reference trial. A potential improvement on this policy is to take into account the number of children a search node can parent. We will call the number of search operators that apply at a rule its *rank*. Rank can be used to compute the size of the unpruned search space parented by rule r. Let *spacesize* be the maximum number of nodes a searcher will have to expand if it is given rule r as its workload. Certainly, the maximum amount of work a searcher could have to do is much greater 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. D-MSDD's *rank-based* policy attempts to balance, in a capacity sensitive manner, the total spacesize it allocates to different searchers.

Dynamic load balancing schemes are a class of algorithms that perform load balancing after work begins. For D-MSDD, dynamic load balancing is initiated when a client detects that its agenda is about to become empty. In such a situation, the client sends a message to the server indicating 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.

When the server receives the work request, it first checks its own agenda to see if there is enough work there to offload some minimum number of nodes. If there is, the server invokes a 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 request for work to all connected clients, who themselves invoke the static load balancing algorithm when possible.

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 more recent estimates of its own processing capacity. Updates to the capacity lookup table are sent to the server along with each request for more work.

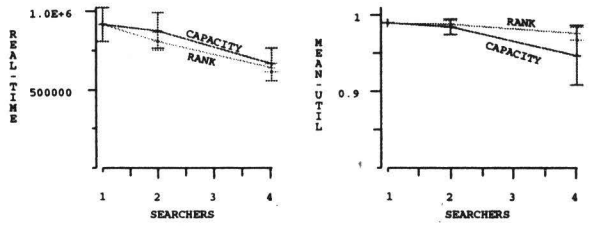

Figure 2: Results for the solar flares data after adding dynamic load balancing to D-MSDD. **(a)** the effect on the time to completion for unloaded machines **(b)** the effect on mean CPU utilization.

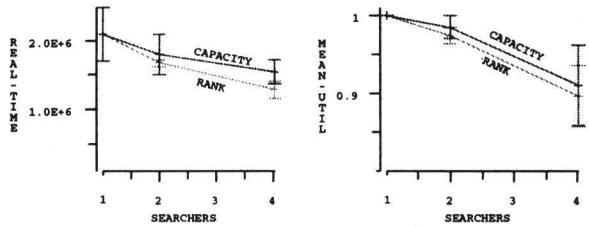

Figure 3: Results for the chess data after adding dynamic load balancing to D-MSDD. **(a)** the effect on the time to completion for unloaded machines **(b)** the effect on mean CPU utilization.

4.2 Evaluation

The basic MSDD algorithm has been shown to be effective in terms of the quality of the rules it discovers [Oates and Cohen, 1996a] and efficient in its search of very large spaces. Our goal in evaluating D-MSDD is to test the hypothesis that efficiency increases proportionally to the computing resources that are added to the search.

We measure performance gain (or loss) through four variables: the total number of *nodes expanded*, *CPU time*, *CPU utilization*, and the number of *messages generated*. The number of nodes considered in the search is a raw measure of computational expense. CPU time is measured in milliseconds as the sum of system and user time spent on behalf of MSDD . All results reported here are for machines in which D-MSDD is the primary load. 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. Finally, the number of messages generated is simply a tally of the TCP/IP messages sent from any searcher to another during the search.

The datasets used for evaluation of D-MSDD were the solar flare dataset and the chess endgame dataset, both from the UC Irvine repository. In all cases, k was set to 20. The number of machines involved in the search varied from one to five, and included one 500MHz Alpha, three 175MHz Alphas, and one 40MHz Sparc10.

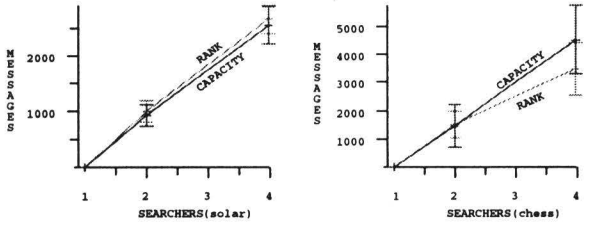

Figure 4: The number of network messages generated during search with dynamic load balancing turned on. **(a)** the solar flare dataset **(b)** the chess dataset.

It should be noted that D-MSDD was built on an old version of MSDD that was implemented in Lisp. Therefore, direct comparisons of running times and numbers of nodes expanded between D-MSDD and the current implementation of MSDD are impossible. However, we believe that the results in this section will be qualitatively the same when D-MSDD is re-implemented on the C version of MSDD.

Performance results with dynamic load balancing working in conjunction with both the capacity and rank static load balancing algorithms are shown in Figures 2 and 3. The graphs of mean CPU utilization show the effect that we had hoped for. For both the solar flares and chess datasets, mean CPU utilization show only slight decreases as processors are added, and are generally within the 90-95% range. The minimum CPU utilization, not shown, exhibited similar behavior, in most cases between 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 in the single processor case is an Alpha approximately 4.75 times faster than the machines added in the 2 and 4 processor cases. In the ideal case, then, the performance increase would be around 163%. With the rank based load policy, the mean speedup in our trials was 162% for the solar flares and 143% on the chess data. The dynamic load balancing scheme achieves high levels of CPU utilization despite the relatively poor scheduling information available to the static policies.

Figure 4 shows the number of network messages generated under the dynamic load balancing scheme. The number of messages generated while searching the solar flare and chess datasets appears linear with respect to the number of searchers and in the thousands.

5 Discussion

Although the core MSDD algorithm is capable of finding complex dependencies in exponential search spaces, we demonstrated the utility of three methods for further reducing the number of rules that MSDD considers. The impact of the methods depends to a large extent on the nature of the database, but in all cases at least one of the methods resulted in significant reductions in running time. If one is willing to forgo MSDD's optimality guarantees (with respect to the G values of the final rule set), then reductions in CPU time and nodes expanded of up to an order of magnitude are possible while still finding high quality rules. In addition, we demonstrated that it is possible to distribute MSDD's search for structure over multiple networked machines and achieve an almost linear speedup in the number of machines used.

Acknowledgements

This research is supported by DARPA/AFOSR under contract number F49620-97-1-0485. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding 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 Defense Advanced Research Projects Agency, Air Force Office of Scientific Research or the U.S. Government.

References

[Cook, 1996] 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.

[Gao *et al.*, 1995] 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.

[Oates and Cohen, 1996a] 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.

[Oates and Cohen, 1996b] Tim Oates and Paul R. Cohen. Searching for planning operators with context-dependent and probabilistic effects. 1996.

[Oates and Cohen, 1996c] 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.

[Oates *et al.*, 1998] Tim Oates, David Jensen, and Paul R. Cohen. Correlating and predicting asynchronous events. In *Working Notes of the AAAI-98 workshop on Predicting the Future: AI Approaches to Time-Series Analysis*, pages 73–79, 1998.

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