A Distributed Approach to Finding Complex Dependencies in Data

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

March 16, 1998

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 strives 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.

Keywords: prediction, dependency detection, time series, distributed algorithms, load balancing

1 Introduction

We are concerned with the analysis of dependencies in time series data. 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 a certain stock always seems to drop several points the third Friday of every month. *Dependencies* express unexpectedly frequent occurrences of one pattern (called the *successor*) following another (the *precursor*) in 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 $(A \ * \ 7) \stackrel{3}{\Rightarrow} (* \ W \ 3)$ 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. The systematicity of the search ensures that no node can ever be generated more than once [OGC95a, 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. In this paper, we explore some possibilities for efficiently distributing the search for dependencies in multivariate, time series data.

2 The MSDD Algorithm

MSDD finds the k strongest dependencies in a dataset 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 $x \stackrel{\delta}{\Rightarrow} y$, 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. The successor pattern for the sample rule in Section 1 is denoted $(* \; W \; 3)$.

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.

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. 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 in the training data. 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. As a consequence, 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. Oates has shown that an optimistic upper bound on the G score ($Gmax$) for the descendents of a dependency can be established. The derivation is omitted here for the sake of brevity, but can be found in [OC96b]. The optimistic upper bound on G is the basis of a simple but powerful pruning heuristic. The value $Gmax$ is computed for each node, as it is evaluated, and compared it 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 considering any of the current node's children, and MSDD prunes the search space at that point. Experiments

with both real and artificial data show that the use of this G-based pruning heuristic results in very focused exploration, with the algorithm often considering only tiny fractions of the complete space of dependencies [OSJC97].

3 Distributed Search

The systematic expansion of nodes performed by MSDD guarantees that evaluating any two frontier nodes can be done without the need for synchronizing the computations. Further, these operations only require access to two data structures: the dataset, and the list of the k best dependencies. Because access to the training data is read-only, it can simply be replicated at each of the hosts involved in the search[1]. 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 local copies of the k-best list are updated lazily. In the sections that follow, we will describe the design of the DMSDD algorithm, describe our criteria for evaluating it, and then move on to the more challenging issue of efficiently balancing the distributed search.

3.1 DMSDD

DMSDD 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 to 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.

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.

[1]For massive datasets, this may not be an option. Distributed search of massive datasets is a topic for future research.

Many solutions to the load balancing problem have been proposed for problems for which optimality results do not apply, such as IDA* search [Coo96]. 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 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. Dynamic load balancing takes place while the search is in progress. An example of dynamic load balancing in DMSDD 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.

3.1.1 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 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 DMSDD 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 data used for evaluation uses MSDD for the task of feature correlation. In this task, each multivariate observation is presented as both the precursor and successor during training. The resulting rules express strong correlations among features within (rather than between) observations in the training data. We chose two datasets from the UCI machine classification repository: solar flare data, and chess endgame results.

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 MSDD run on a single 500MHz Alpha processor. As with all the trials reported here, MSDD was configured to find the 20 strongest

	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.

Machine	Configuration	Capacity
goddard	500MHz Alpha/512M	$3.79\frac{nodes}{sec}$
lindy	175MHz Alpha/128M	$0.80\frac{nodes}{sec}$
earhart	175MHz Alpha/96M	$0.80\frac{nodes}{sec}$
gagarin	175MHz Alpha/80M	$0.73\frac{nodes}{sec}$
hinden	40MHz SPARC10/128M	$0.65\frac{nodes}{sec}$

Table 2: DMSDD's capacity lookup table for our local network.

dependencies ($k = 20$).

3.1.2 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.

Our first offering for a static load balancing policy considers the relative speeds of the machines when dividing the initial problem space. This *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. Entries in the capacity database for our local network are shown in Table 2.

The graphs in Figures 1 and 2 show the effects of adding processors to the search using static load balancing. 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. For all trials reported here, machines were added to the search in the order in which they appear in Table 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 update messages to local k-best lists may experience propagation delays, a small number of nodes may temporarily escape pruning. Time to completion, shown in Figures 1b and 2b, behaves somewhat differently than expected, though. The time to completion can actually increase as processors are added to the search!

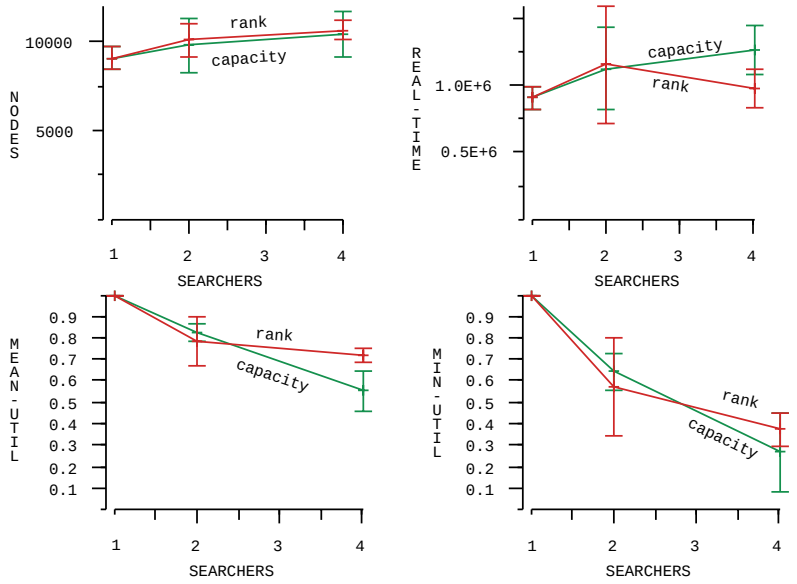

Figure 1: The effects of adding more workstations to the search of the solar flare dataset with the capacity sensitive and rank sensitive load balancing policies. Each data point is based on 5 samples and is shown with 90% confidence intervals. **(a)** the effect on the total number of nodes expanded **(b)** the effect on the time to completion for unloaded machines **(c)** the effect on mean CPU utilization **(d)** the effect on the minimum CPU utilization.

One need only look to the graphs of CPU utilization to get an indication of how the search could actually take longer when additional resources are available. As processors are added, overall CPU utilization declines, indicating that mistakes are being made in the static partitioning phase. The problem is stems from the systematic expansion of the search space. The capacity sensitive algorithm does not take into account how prolific each offloaded node may be – the number of children can vary greatly from search node to search node. The result is a search that lasts as long as it takes the machine allocated the most overall work to complete, leaving the other processors idle for as much as 70% of the total time to completion. In our experiments, the machine in the 1 processor case is the fastest of the bunch. Thus, adding slower machines to the search can lengthen the overall search time.

The static policy should improve, then, if it would only take into account the number of children a search node can parent. We will call the number of uninstantiated tokens to the right of the righmost instantiated token for a rule its *rank*. Rank can be used to compute the size of the unpruned search space parented by rule r. The calculation is a recursive summation, and is a function of the number of data streams m, and the rule r's rank, indicated in the formula below as x:

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

Spacesize computes the maximum amount 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.

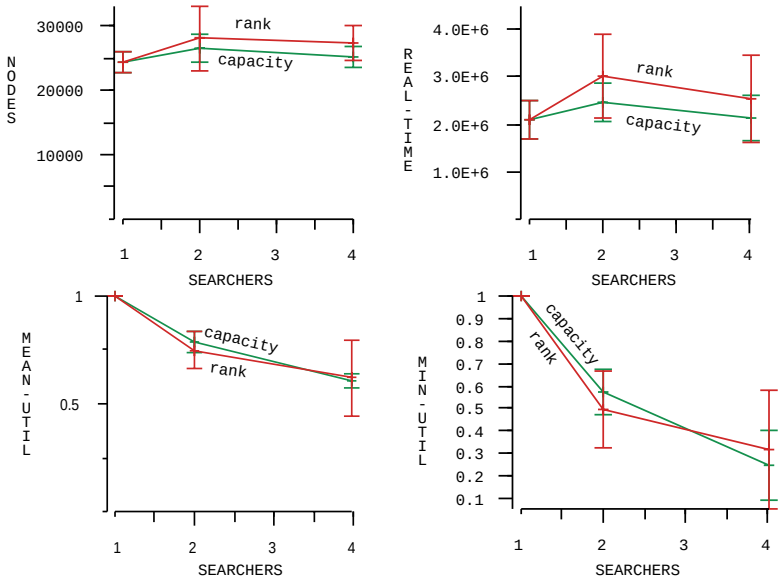

Figure 2: The effects of adding processors to the search of the chess endgame dataset. **(a)** the effect on the total number of nodes expanded **(b)** the effect on the time to completion for unloaded machines **(c)** the effect on mean CPU utilization **(d)** the effect on the minimum CPU utilization.

DMSDD's *rank-based* policy attempts to balance, in a capacity sensitive manner, the total spacesize it allocates to different searchers. 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 1 and 2 show the effects of adding processors to the search with rank-based load balancing. The performance of rank-based load balancing appears to scale somewhat better than the capacity-based policy for the solar flares set. In the case of four processors, the rank based scheme shows an improvement in mean CPU utilization of roughly 20% over capacity-only load balancing, and has less variance in the results. Results on all other measures are mixed. The persistent problem 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, a seemingly unsafe assumption for the attribute correlation task.

3.1.3 Dynamic Load Balancing

Overall, static load balancing does not appear to be feasible as a standalone load balancing policy for DMSDD. The major fault of static load balancing is that the information useful to load balancing becomes available only as the search progresses. Before any nodes are rated and the k-best list starts filling out, DMSDD has little information to base its load balancing on. The solution to this problem is to allow processors to correct the misallocations of the static policy by dynamically rebalancing their workloads.

Dynamic load balancing schemes are a class of algorithms that perform load balancing after work begins. For DMSDD, 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

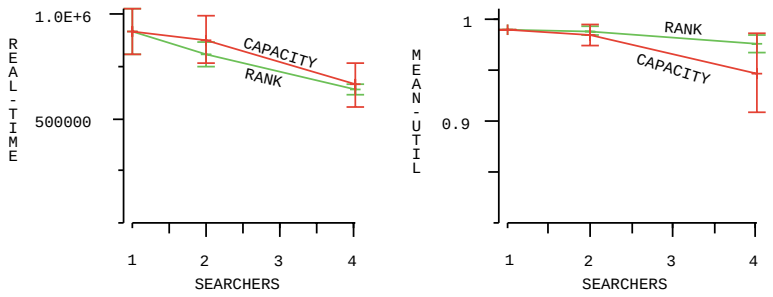

Figure 3: Results for the solar flares data after adding dynamic load balancing to DMSDD.**(a)** the effect on the time to completion for unloaded machines **(b)** the effect on mean CPU utilization.

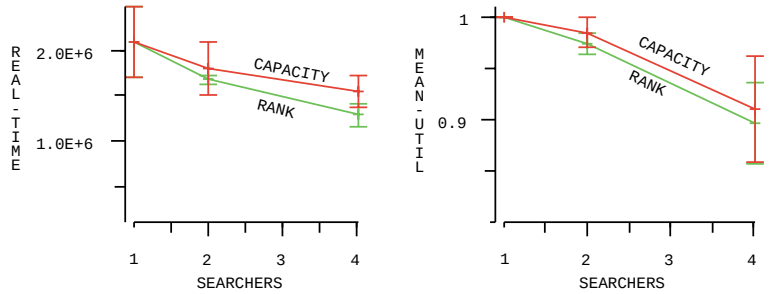

Figure 4: Results for the chess data after adding dynamic load balancing to DMSDD. **(a)** the effect on the time to completion for unloaded machines **(b)** the effect on mean CPU utilization.

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.

Performance results with dynamic load balancing working in conjunction with both the capacity and rank static load balancing algorithms are shown in Figures 3 and 4. 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,

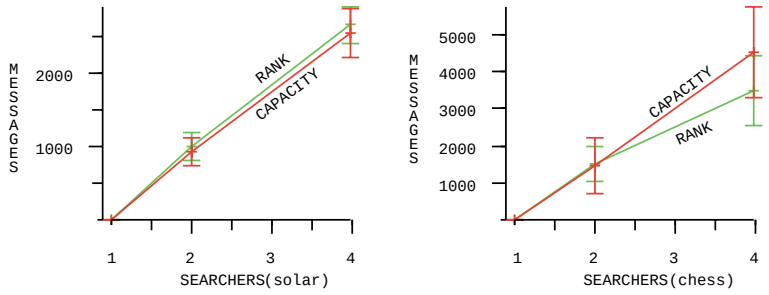

Figure 5: The number of network messages generated during search with dynamic load balancing turned on. **(a)** the solar flare dataset **(b)** the chess dataset.

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.

Better load balancing does not come without a cost, though. Figure 5 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. For networks with large propagation delays, or large numbers of processors this performance degradation could be significant.

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 the G statistic, a measure with a provable upper bound for the children of a rule, MSDD performs 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 space 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 static load balancing policies are based on the assumption that the k strongest dependencies will be uniformly distributed throughout the search space, an assumption clearly violated by the feature correlation task. In such cases where the assumptions of static analysis cannot be met, we found that dynamic load balancing was necessary. Our dynamic policy, working in conjunction with the static ones, provided significant improvements, producing high levels of CPU utilization and apparently 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. This result indicates that speedup and scalability might be compromised for large numbers of processors. 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.