A Family of Algorithms for Finding Temporal Structure in Data[1]

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

Abstract

Finding patterns in temporally structured data is an important and difficult problem. Examples of temporally structured data include time series of economic indicators, distributed network status reports, and continuous streams such as flight recorder data. We have developed a family of algorithms for finding structure in multivariate, discrete-valued time series data (Oates & Cohen 1996b; Oates, Schmill, & Cohen 1996; Oates *et al.* 1995). In this paper, we introduce a new member of that family for handling event-based data, and offer an empirical characterization of a time series based algorithm.

1 Introduction

Dependency detection is an approach to finding patterns in time series or event data based on locating unexpectedly frequent or infrequent co-occurrences of patterns in the data. We call these co-occurrences *dependencies* that can be expressed as rules of the following form: "If an instance of pattern x is observed at time t, then an instance of pattern y will be observed after some delay with probability p." Dependency rules are denoted $x \Rightarrow y$; x is called the *precursor* pattern, and y is called the *successor* pattern. A dependency is strong if the empirically determined value of p (obtained by counting actual co-occurrences of x and y in the data) is very different from the probability of seeing a co-occurrence of x and y under the assumption that they are independent. Our algorithms find strong dependencies between patterns by performing a general-to-specific, systematic search over the space of possible dependencies.

We begin by introducing Multi-Event Dependency Detection (MEDD), an algorithm for finding patterns in event-based data. We report on its performance in a computer network event correlation task. Next, we give an empirical characterization of the Multi-Stream Dependency Detection (MSDD) algorithm for time series based dependency detection. We conclude with a discussion of the general efficiency and applicability of the family of dependency detection algorithms.

2 Searching for Structure in Event-based Data

Computer networks produce large amounts of event-based data, and management of such networks is largely driven by the generation and interpretation of events. A problem that plagues network managers is the large number of events of different types from disparate locations in the network that result from network faults (Oates 1995). Finding patterns in those events to form clusters of related events is important for reducing the amount of information that must be interpreted and

[1]This research is supported by DARPA/RL F30602-93-C-0076, and by a subcontract from Hughes Information Technology Corp.RFQ 96-ECS-UMass-001. 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 necessarioy representing the official policies or endorsements either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

for understanding the state of the network (e.g. by identifying whether a set of events represents the effects of a single fault or multiple, concurrent faults).

Multi-Event Dependency Detection searches for frequently co-occurring patterns of events recorded in event logs. Because strict temporal ordering cannot be enforced in network logs, MEDD searches for dependencies among groups of events recorded within some temporal window δ. Thus, MEDD dependencies can be expressed by rules of the form: "If an instance of event pattern x is recorded in the log at time t, then an instance of event pattern y will be recorded between $t - \delta/2$ and $t + \delta/2$ with probability p."

Although the goal of our work with MEDD is the development of a tool that automatically acquires knowledge from network event logs for the purpose of event correlation (Jensen, Oates, & Cohen 1996; Oates, Jensen, & Cohen 1995), we believe that the approach is very general. The algorithm's ability to find dependencies in event data may be useful in many other fields that manage large, interconnected systems; e.g. healthcare, transportation, logistics, telecommunications, etc.

2.1 Multi-Event Dependency Detection

MEDD accepts as input a set of historical event logs, searches for dependencies between patterns of events in those logs until a user defined limit on the number of nodes to expand is reached, and returns a list of the nodes explored. The nodes are then filtered to identify those that represent strong dependencies as measured by G, a statistical measure of non-independence (Wickens 1989). The precursors of the remaining rules can then be matched against new event data to predict occurrences of their successors.

Precursor and successor event patterns contain one or more *partially instantiated events* (PIEs). In general, events recorded in logs comprise multiple fields, and each field takes a value from a set of allowable values specific to that field. For example, the `status` field might take values from the set {`up, down`}. Assuming that events contain f fields, and that field i takes values from the set $\mathcal{V}_i$, then the space of all possible events is given by $\mathcal{E} = \times_{i=1}^{f} \mathcal{V}_i$ (that is, the cross product of all of the $\mathcal{V}_i$ – every possible combination of field values). Any event e that appears in an event log is an element of $\mathcal{E}$. PIEs simply leave the value of one or more fields unspecified, which is denoted by assigning those fields the wildcard value $*$. Therefore, the space of all possible PIEs is given by $\mathcal{P} = \times_{i=1}^{f}(\mathcal{V}_i \cup \{*\})$. Note that $\mathcal{E} \subset \mathcal{P}$. Consider a simple event structure containing two fields – `status` and `element` – such that $\mathcal{V}_{status}$ = {`up, down`} and $\mathcal{V}_{element}$ = {`host, router`}. Then $\mathcal{E}$ and $\mathcal{P}$ are as follows:

$$\mathcal{E} = \left\{ \begin{array}{ll} \texttt{(up host)} & \texttt{(up router)} \\ \texttt{(down host)} & \texttt{(down router)} \end{array} \right\} \qquad \mathcal{P} = \left\{ \begin{array}{lll} \texttt{(up host)} & \texttt{(up router)} & \texttt{(up *)} \\ \texttt{(down host)} & \texttt{(down router)} & \texttt{(down *)} \\ \texttt{(* host)} & \texttt{(* router)} & \texttt{(* *)} \end{array} \right\}$$

A PIE $p \in \mathcal{P}$ is said to *match* an event $e \in \mathcal{E}$ if every non-wildcard field in p has the same value as the corresponding field in e. For example, the PIE p = (`up *`) matches event e_1 = (`up router`), but it does not match event e_2 = (`down host`) or event e_3 = (`down router`). Event patterns, precursors and successors, are defined to be sets of PIEs; i.e. $x = \{p_1, \ldots, p_n | p_i \in \mathcal{P}\}$ is an event pattern. Precursors and successors are said to match a fragment of an event log if each of their constituent PIEs can be matched on a *different event* in the fragment.

MEDD's traversal of the space of dependencies between event patterns is both general-to-specific and systematic. Each node in the search space corresponds to a dependency rule, and the root of that space is the completely general rule in which both the precursor and successor contain only wildcards. For the simple two-field event structure introduced earlier, the root node contains the rule {(`* *`)} $\Rightarrow$ {(`* *`)}. The children of a node are generated by modifying either the precursor or

successor of that node in one of two ways: by filling in the value of a field that contains a wildcard in an existing PIE, or by adding a new PIE containing a single non-wildcard field. In either case, the descendants of a node are always more specific – they specify more non-wildcard values for fields – than the original node.

The search is made systematic, and therefore more efficient (Rymon 1992; Webb 1996), by only adding non-wildcards and PIEs to the right of the right-most non-wildcard in a node when generating that node's children. Consider the node {`(up *)`} ⇒ {`(down *)`}. The right-most non-wildcard in this rule is `down` in the successor. Therefore, {`(up *)`} ⇒ {`(down router)`} is a valid child, but {`(up router)`} ⇒ {`(down *)`} would not be generated because it requires adding a non-wildcard to the left of `down`. (The interested reader is referred to (Oates & Cohen 1996b) for a detailed discussion of the use of this type of operator ordering to achieve systematicity.)

MEDD's search through the space of dependencies is guided by a best-first heuristic. Each time a node is generated, MEDD scans its historical event logs, counting the number of times the precursor and successor of that node co-occurred within a temporal window of size δ (specified by the user). Frequency of co-occurrence becomes the node's heuristic value, biasing the search to prefer rules with frequently occurring precursors and precursor/successor pairs that frequently co-occur. The search proceeds by iteratively selecting the node with the highest value, generating that node's children, and adding them to the list of nodes under consideration.

Counting co-occurrences of precursors and successors is not as straightforward as it might appear. For a log fragment with n events and a pattern with k PIEs, there are n choose k ways that the pattern might match. If the precursor matches a fragment, then the successor must also be checked for a match. That process is complicated by the fact that the successor may not match because certain events are "taken" by the precursor (the precursor and successor must match on *different* events), but the precursor could match on a different set of events allowing the successor to match. In the worst case, all possible matches of the precursor must be tried to find a match for the successor. We currently use an approximate counting scheme, and are investigating ways to efficiently implement more accurate counting algorithms.

MEDD returns all of the nodes that it explores in the space of dependencies. To find the strongest dependencies among those explored, a 2x2 contingency table that describes the frequency of co-occurrence of each rule's precursor and successor is built. (Actually, the complete table is built during the search as each node is expanded. The first cell of the table is used as the node's heuristic value to guide the search.) Then, the G statistic, a statistical measure of non-independence, is computed for each rule, and the rules are sorted in non-decreasing order of G. We then remove generalizations of the strongest rules that were generated as the search descended through the tree to find those rules. Finally, the top k rules are retained. Currently, the choice of k is ad hoc. We are investigating automated methods for choosing "good" values for k.

2.2 Experiment

This section describes an experiment designed to test both MEDD's ability to find strong dependencies between patterns of events and the utility of those dependencies with respect to event correlation. Event logs were generated by a modified version of the Netsim network simulator which is publicly available from MIT (Heybey & Robertson 1994). The simulator was modified to allow randomly selected components to fail. In addition, a monitor component was added to the simulator to act as a Simple Network Management Protocol (SNMP - the Internet standard for network management) proxy agent. That component periodically polls the other network components and reports on their reachability.

Netsim generated two separate event logs for a simulated network containing 17 components.

Each event contained six fields: the ID of the component reporting the event, that component's type, the time at which the event was reported, the event type, and two additional fields whose semantics and contents depend on the event type. The first event log, which covered 261 network errors and 237 reported events, was used by MEDD to search for dependency rules. Not all errors generated events, and some errors generated multiple events. The latter errors are the ones that MEDD attempts to correlate. The second event log, which covered 248 network errors and 243 reported events, was used as a source of new events to test the rules generated from the previous log. The rules were used to find and report correlated events.

MEDD generated 30,000 search nodes, and the total CPU time required by the search and post-processing was 344 seconds. Post-processing removed all but 2761 rules, and the top 200 were retained (rather arbitrarily) for correlating future events. Given knowledge of the actual errors introduced into the simulated network while generating the test log, it is possible to form 56 groups of events for which the events in each group are causally related to a single error. Events not contained in these groups should not be correlated with other events because they are not causally related to any other events. Of the 56 possible groups, MEDD's rules found all or part of 35. In six of those cases, an extraneous event that belonged to a different error was included. Of the 21 groups that MEDD failed to identity, 17 involved an indirect manifestation of an error and a report from the monitor component identifying the cause of that manifestation. That is, the missed groupings almost entirely involved diagnostic relationships, a problem that may be alleviated by recourse to domain knowledge. Finally, MEDD formed groupings of causally unrelated events 8 times. In several of those cases, the groupings pair an event reporting one error with an event reporting a manifestation of a second (different) error that could have been caused by the first error. Our conjecture is that deepening the search for rules (i.e. increasing the number of nodes that MEDD explores) will lead to more specific rules that will be able to distinguish such cases.

2.3 Discussion

The experimental results reported in the previous section are very encouraging. For a simulated network experiencing multiple (often almost simultaneous) errors, rules generated by MEDD can successfully correlate events caused by a single root error. The rules rarely incorporate unrelated events, and they rarely form completely erroneous groupings. Future work will involve reducing the number of errors that the rules commit, perhaps allowing a user-defined tradeoff between aggressiveness in grouping events and the potential for different types of errors.

3 Empirical Characterization of MSDD for Time Series Data

Multi-Stream Dependency Detection (MSDD) is an algorithm for finding the strongest dependencies in multiple synchronized streams of discrete time series data. As opposed to MEDD, MSDD assumes that its data are strictly ordered, and thus its rules predict future patterns from observed precursor patterns: "If an instance of pattern x beginning at time t is observed, then an instance of pattern y will be observed beginning at time $t + \delta$ with probability p." Consider three streams:

Stream1	**A**	C	F	B	L	A	A	B	B	B	B	C	**A**	F	F	L	B
Stream2	V	**W**	W	**W**	V	X	W	X	Y	X	W	Y	W	**W**	X	**W**	X
Stream3	**7**	3	1	**3**	1	7	6	6	2	5	3	5	**7**	5	5	**3**	1

The boldfaced tokens highlight a dependency between two patterns in these streams. The following rule represents a co-occurrence that was observed twice in the streams above:

$$\begin{bmatrix} \text{A} & * \\ * & \text{W} \\ 7 & * \end{bmatrix} \overset{3}{\rightarrow} \begin{bmatrix} * \\ \text{W} \\ 3 \end{bmatrix}$$

This rule says, "When you see A in Stream 1 and 7 in Stream 3 on timestep t, and W in Stream 2 on timestep $t+1$, 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 spanned by the precursor or successor is called the *block size*. The precursor and successor above have block sizes of 2 and 1, respectively.

Rules like this are hard to find; the search space is exponential (Oates *et al.* 1995). MSDD treats dependency detection as an efficient search for the k most predictive rules in the search space.

3.1 The MSDD Algorithm

MSDD accepts as input a dataset of *streams*, each of which comprises a list of categorical *tokens*. It returns the k strongest dependencies between precursor and successor patterns. MSDD performs a general-to-specific, systematic search over the space of possible dependency rules, starting with a root node that specifies all wildcards for both the precursor and the successor. The children of a node are generated by adding a single token to its precursor or successor. By imposing a total ordering on the adding of tokens, MSDD ensures that each dependency rule is generated at most once (Oates *et al.* 1995; Webb 1996). To evaluate a rule $R_p \rightarrow R_s$, MSDD counts co-occurrences of R_p and R_s, and also occurrences of other precursors $\overline{R_p}$ and other successors $\overline{R_s}$ in the dataset. The following contingency table summarizes these counts:

	R_s	$\overline{R}_s$
R_p	n_1	n_2
$\overline{R}_p$	n_3	n_4

MSDD is interested in unusually high or low cell counts in n_1; this is the cell that represents the number of times $R_p \rightarrow R_s$ held true in the data. For the purpose of rating nodes, MSDD uses the G statistic for measuring nonindependence in 2x2 contingency tables (Wickens 1989). MSDD maintains a list of k rules sorted by the G statistic. Because it is possible to derive an optimistic upper bound on the G statistic for the children of a given rule, MSDD is able to prune large portions of the search space when the upper bound on G for a node's children is found to be less than that of the worst rule present in the k-best list (Oates & Cohen 1996b). The systematic search for rules continues until all of the unvisited nodes at the fringe of the search tree have no chance at being among the k strongest dependencies.

3.2 Empirical Evaluation

MSDD has been tested in several domains, including learning planning operators (Oates & Cohen 1996a), predicting pathologies in a simulating shipping network, and classification (Oates & Cohen 1996b). Our purpose here is to provide an empirical assessment of its performance when its parameters and the structure of datasets are manipulated. Each dataset comprised n streams of length l tokens. The tokens were selected from an alphabet of a tokens. Each *target rule* $R_p \rightarrow R_s$ involved c non-wildcards in both R_p and R_s; we call this value c the rule's *complexity*. The block size for R_p and R_s was always 1. We constructed r target rules and inserted them into otherwise random

streams. For each experiment we set k and measured the internal CPU time of the trial, the number of nodes expanded in the search, and the proportion of tokens that were actually predictable in the streams. Each trial was replicated 5 times to generate estimates of mean performance and variance.

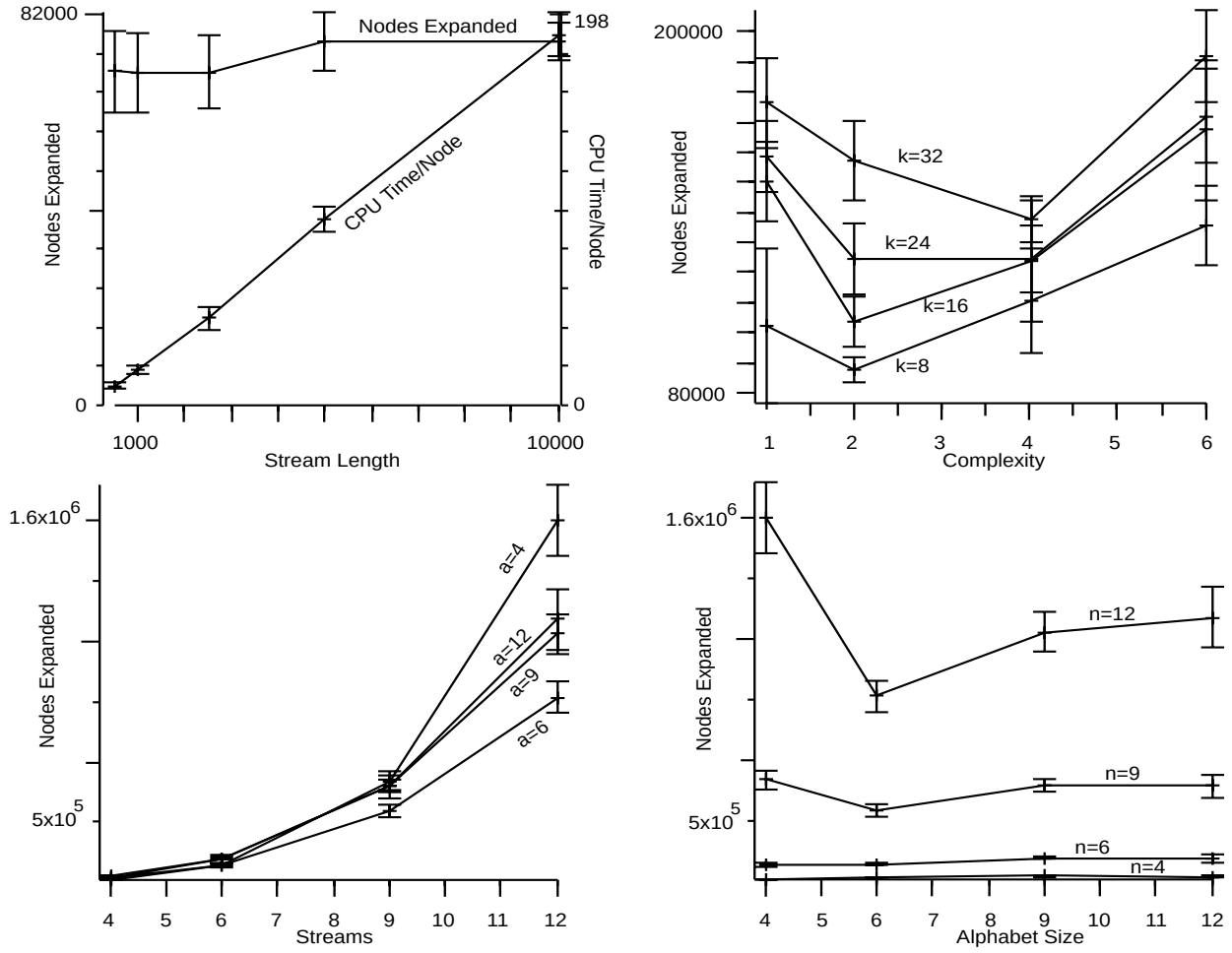

Figure 1: Top, left(**a**): trend of CPU time per node and nodes expanded as stream length l increases; top, right(**b**): nodes expanded at various levels of k over increasing complexity; bottom, left(**c**): the effect of n for various levels of a on nodes expanded; bottom, right(**d**): the effect of a for several levels of n. Each datapoint represents 5 trials with 95% confidence intervals.

3.2.1 Stream Length

Figure 1a shows the trends of CPU time per node expanded and total nodes expanded as l grows from 500 to 10,000 training instances, with all other measures held constant. Nodes expanded remains quite stable as stream length increases, while CPU time per node ascends in a linear fashion to its maximum at $l = 10000$. Because the data is recounted during the evaluation of each node, CPU time is proportional to stream length. (Other parameters, a, n, k, and r, have constant effects on CPU time.) However, stream length has no significant effect on the number of nodes expanded.

3.2.2 K and Complexity

The choice of k specifies the number of strongest dependencies that MSDD will report at the conclusion of the search, and implicitly specifies how aggressively MSDD will be able to prune. Choice of a low value for k ensures that the k-best list will fill rapidly with highly-rated dependencies, making it easier for MSDD to prune weak rules; however, MSDD may miss less accurate (but valid) dependencies when there are more than k stronger ones in the data. A large value for k results in more nodes being searched and more subtle dependencies detected. Figure 1b plots the number of nodes expanded as a function of rule complexity c—the number of non-wildcards in R_p and R_s—for various levels of k, in a dataset created with seven streams and seven target rules.

An interesting feature of graph 1b is that as c varies from 1 to 6, the number of nodes expanded first decreases and then begins to grow. In cases where $k > r$, the k-best list cannot be filled by target (strong) dependencies. MSDD must then search for rules to fill out the $k - r$ "free" spaces in the k-best list. This gives rise to an interaction effect between c and k. Consider a rule $R_p \rightarrow R_s$ with two non-wildcards in R_p and R_s (i.e., $c = 2$). A candidate rule with one correct token in R_p and two correct tokens in R_s is a *generalization* of the target rule; it might have a high G value, indicating structure below it in the search tree. As the complexity of the target rules increases, the probability increases that some generalizations of the target rules will be highly-rated. Thus, the k-best list tends to become populated with highly rated generalizations, and spurious rules are quickly pruned. For low values of c ($c = 1$ in particular), there are not enough generalizations to fill the k-best list, and thus some number of slots in the k-best list will be contested by spurious rules. The result is that MSDD cannot prune as significant a portion of the search space. These results are reflected in graph 1b; the downward slope at the outset of the plots represents more abundant generalizations as c increases overcoming an unnecessarily large choice of k.

Irrespective of k, complexity affects the depth of the effective MSDD search space. Because the space is structured from general to specific, more complex target rules are situated more deeply in the search tree. We see this effect in the ascending portions of the plots of graph 1b. Experiments holding k constant, and varying c for trials with larger numbers of streams indicate that the effect of c on nodes expanded may be superlinear.

3.2.3 Alphabet Size and Number of Streams

Token alphabet size, a, and number of streams, n, are the main contributors to the size of the unpruned search space. Analysis of the space's organization indicates that n affects the search tree's depth and both a and n effect the branching factor. Graph 1c plots the observed effect of n on the number of nodes expanded. This plot shows a superlinear growth of nodes expanded as n grows. Most interesting, though, is the effect of varying a in these trials. The trial where $a = 4$ grows most rapidly; even more quickly than when $a = 12$.

Figure 1d plots the number of nodes expanded versus a in the same trials as figure 1c. For very low values of a, there is little possible token variance in the streams to distinguish between random co-occurrences and actual dependencies. As a result, some very general, yet random co-occurrences may appear somewhat frequently in the data, and MSDD must consider many of these highly rated spurious associations before the search can terminate. As a increases, the probability of seeing any given random co-occurrence decreases, lengthening the distance between the real dependency ratings and spurious ones. We see this "distinguishing effect" when a increases from 4 to 6. As a increases beyond 6, this effect becomes insignificant compared to the increase in branching factor, and we see an increase in the number of nodes expanded.

3.3 Discussion

By systematizing its search and pruning the children of all but the k best nodes, MSDD efficiently searches an exponential space of dependency rules. Experiments indicate that the effects of complexity and number of streams on the number of nodes expanded appear to be superlinear; however, MSDD still only expands a tiny fraction of its search space. In our experiments, MSDD explored as few as 10^{-21} of the possible rules in the $n = 12, a = 12$ case. For a search space of depth 7, the minimum depth required to correctly identify rules of complexity 3, MSDD prunes more than 98% of the space in the $n = 12, a = 12$ case. Additionally, MSDD appears to be affected little by choice of conservatively large choices of k in all but the $c = 1$ case, and exhibits only small increases in nodes searched as a becomes larger.

4 Conclusions

We have described a family of algorithms for finding dependencies in temporally structured, multivariate datasets. Based on a systematic, general-to-specific search procedure, MEDD locates correlated event patterns in event-based data, and MSDD determines the k strongest dependencies in time series data. The MEDD algorithm provided encouraging results by uncovering much of the underlying structure present in a simulated network application. Results of a controlled set of experiments indicate significant pruning of the exponential space of dependencies is possible due to a heuristic based on optimistic upper bounds on the G statistic for a dependency's children. This pruning allows MSDD to conduct its search in a tiny portion of the overall search space. Because these two algorithms are both based on the same principles, we expect the empirical findings on MSDD to apply to a k-best MEDD algorithm.

References

Heybey, A., and Robertson, N. 1994. The network simulator version 3.1.

Jensen, D.; Oates, T.; and Cohen, P. R. 1996. Prototype for ECS fault identification: Requirements and design. Report Prepared for Hughes Information Technology Corporation.

Oates, T., and Cohen, P. R. 1996a. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence.*

Oates, T., and Cohen, P. R. 1996b. Searching for structure in multiple streams of data. In *Proceedings of the Thirteenth International Conference on Machine Learning*, 346 – 354.

Oates, T.; Schmill, M. D.; Gregory, D. E.; and Cohen, P. R. 1995. Detecting complex dependencies in categorical data. In Fisher, D., and Lenz, H., eds., *Finding Structure in Data: Artificial Intelligence and Statistics V.* Springer Verlag. 185 – 195. Includes work on an incremental algorithm not contained in workshop version.

Oates, T.; Jensen, D.; and Cohen, P. R. 1995. Technologies for fault management. Report Prepared for Hughes Information Technology Corporation.

Oates, T.; Schmill, M. D.; and Cohen, P. R. 1996. Parallel and distributed search for structure in multivariate time series. Technical Report 96-23, University of Massachusetts at Amherst, Computer Science Department.

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

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

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

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