Discovering Rules for Clustering and Predicting Asynchronous Events

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

Abstract

A wide variety of complex systems generate asynchronous events, including nuclear power plants, computer networks, governments, relational database systems and operating systems. We present Multi-Event Dependency Detection (MEDD), a novel algorithm for acquiring event correlation rules from historical logs of asynchronous events. Given a new stream of events generated in real time, the rules enable two important activities: clustering sets of related events and predicting events that will occur in the future. The former activity supports data reduction so that human monitors can more easily understand the state of the system generating the events, and the latter activity facilitates prediction of future states of the system by reasoning about events that are likely to occur. MEDD's utility is evaluated in experiments with event logs generated by a simulated computer network.

Keywords: events, correlation, prediction, rule learning

1 Introduction

A large number of diverse real-world systems generate asynchronous events, electronic status reports, usually in response to abnormal conditions. For example, computer networks and nuclear power plants generate events when individual components fail (Feldkuhn & Erickson 1989). Political systems generate events in the form of media reports when atypical circumstances arise (Schrodt 1994). Complex computer programs, such as relational database systems and operating systems, maintain logs that record a variety of information such as user activity and error conditions (Feldkuhn & Erickson 1989). For the sake of concreteness, the remainder of the discussion will focus on events generated by computer networks.

This paper describes the Multi-Event Dependency Detection (MEDD) algorithm for discovering temporal patterns in historical event logs. These patterns, which are represented in the form of event correlation rules, support two kinds of inference: determining which reported events "go together", and predicting the occurrence of future events. The latter type of inference is clearly important for reasoning about how the state of a system is likely to evolve over time given the events that it is currently generating. The former type of inference facilitates data reduction. For example, a single fault in a network, such as the failure of a router, can generate dozens of events, flooding the manager's console with information. If one or more faults occur simultaneously, it may be difficult to detect that separate faults have occurred based on the single resultant stream of events. Event correlation groups related events into *clumps*, reducing the number of separate reports arriving at a manager's console and organizing that information into meaningful units.

Algorithms to automatically construct event correlation rules face at least three challenges. First, event correlation rules must express important generalizations while abstracting over unimportant details. For example, an event correlation rule might specify the types of network elements that generate the events, but abstract over particular event times and physical locations.

Second, rules must be constructed from incomplete information. The same fault can produce different event patterns, depending on the operating conditions of the network. Although a fault might disrupt communication between two devices, the devices will not report an event unless they attempt to communicate. Thus, all potential events produced by a particular fault are unlikely to be reflected in any one event log. This affects judgments about the quality of event correlation rules. If a rule forms a clump from two abstract events, A and B, then the occurrence of an event of type A without an event of type B is not necessarily negative evidence for the rule. Instead, it may indicate that the conditions for generating an event of type B never arose over the duration of the fault.

Third, rule construction cannot rely on precise temporal ordering of events, especially in computer networks. Events are often reported out of sequence because of non-deterministic messaging delays and the vagaries of clock synchronization. Thus, exact temporal ordering cannot be used to induce causality. Instead, general temporal proximity is used to induce which events are associated.

2 The MEDD algorithm

With these constraints in mind, we designed and implemented MEDD (Multi-Event Dependency Detection). MEDD finds *dependencies* between patterns of network events recorded in event logs.[1] Dependencies are unexpectedly frequent or infrequent co-occurrences of patterns of events, and can be expressed as rules of the following 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 in close temporal proximity with probability p."[2] 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 historical event logs) is very different from the probability of seeing a co-occurrence of x and y under the assumption that they are independent. Strong dependencies capture structure in event logs because they tell us that there is a relationship between their constituent event patterns, that occurrences of those patterns are not independent.

Performing event correlation with MEDD is a two-step process. First, MEDD is used off-line to find strong dependencies between patterns of events in existing event logs. Second, the resulting rules are matched against new events as they are generated in real time. Because strong dependencies indicate that occurrences of precursors and successors are not independent, any co-occurrence of the constituent event patterns of a rule can be reported to the network manager as a single clump, thereby reducing the volume of events that reach the manager's console. That is, all of the events in the precursor and all of the events in the successor can be collapsed into a single unit.

MEDD finds strong dependencies between patterns of events by performing a general-to-specific, best-first, systematic search over the space of all possible pairs of event patterns. Each of the patterns in such a pair may be composed of representation of one or more events. In the remainder of this section, we explain exactly what that means. Section 2.1 defines the space of pairs of event patterns that MEDD searches. Section 2.2 discusses systematic search in general, and Section 2.3 describes the details of systematic search in MEDD. Finally, Section 2.4 describes a post-processing phase that the rules returned by MEDD undergo, and explains in more detail how the rules are used for event correlation in real time.

2.1 The Space of all Possible Event Patterns

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 assigned 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 $\mathcal{E} \subset \mathcal{P}$.

[1] MEDD is based on our earlier work with a similar algorithm named MSDD (Oates & Cohen 1996; Oates, Schmill, & Cohen 1997; Oates *et al.* 1995).

[2] We rigorously define "close temporal proximity" in Section 2.3.

Consider an extremely 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. Therefore, the event pattern $\{$**(* router), (* host)**$\}$ matches the following event log fragment containing three events, whereas the event pattern $\{$**(down *), (* router)**$\}$ does not:

```
up   host
up   host
down router
```

2.2 Systematic Search

MEDD's search for dependencies among events is *systematic*, leading to search efficiency (Oates, Gregory, & Cohen 1994; Riddle, Segal, & Etzioni 1994; Rymon 1992; Schlimmer 1993; Webb 1996). Systematic search non-redundantly enumerates the elements of search spaces for which the value or semantics of any given node are independent of the path from the root to that node. Webb calls such search spaces *unordered* (Webb 1996). Consider the space of disjunctive concepts over the set of literals $\{A, B, C\}$. Given a root node containing the empty disjunct, $false$, and a set of search operators that add a single literal to a node's concept, a *non*-systematic elaboration of the search space is shown on the left in Figure 1. Note that the concept $A \vee B \vee C$ appears six times, with each occurrence being semantically the same as the other five, yet syntactically distinct. In the space of disjunctive concepts, the semantics of any node's concept is unaffected by the path taken from the root to that node.

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

Systematic search of unordered spaces generates no more than one syntactic form of each semantically distinct concept, and is therefore much more efficient than naive search. That is accomplished by imposing an order on the search operators used to generate the children of a node, and applying only those operators at a node that are higher in the ordering than all other operators already applied along the path to the node. Let op_A, op_B and op_C be the operators that add the literals A, B and C respectively to a node's concept. If we order those operators so that $op_A < op_B < op_C$, then the corresponding space of disjunctive concepts can

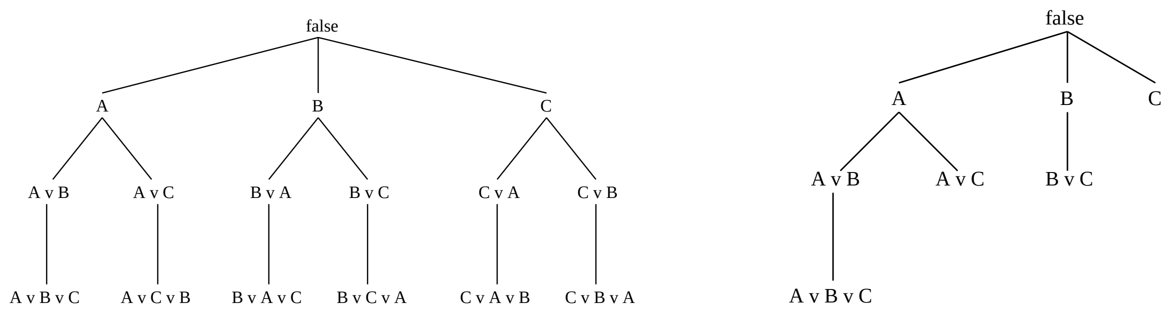

Figure 1: On the left, a naive elaboration of the space of disjunctive concepts over the set $\{A, B, C\}$ generated by applying all valid search operators at every node. Naive search generates multiple syntactic variants of individual concepts. On the right, a systematic elaboration of the space of disjunctive concepts over the set $\{A, B, C\}$. Only one syntactic form of each semantically distinct concept is expanded.

be enumerated systematically as shown on the right in Figure 1. Note that each semantically distinct concept appears exactly once. The concept A is obtained by applying operator op_A to the root node. Because $op_B > op_A$ and $op_C > op_A$, both op_B and op_C can be applied to the concept A, generating the child concepts $A \vee B$ and $A \vee C$. In contrast, the concept C, which is obtained by applying op_C to the root node, has no children. Because all other operators (op_A and op_B) are lower in the ordering than op_C, none will be applied and no children will be generated.

The trees in Figure 1 represent general-to-specific elaborations of the space of disjunctive concepts. That is, the root node is the most general concept $false$, and children of a node are generated by making that node's concept more specific through the addition of a single literal. All of the concepts at depth d contain exactly d literals.

2.3 Search in MEDD

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. This section describes the search in detail, and the next explains how the returned nodes are processed and used for event correlation.

2.3.1 The Structure of the Search Space

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 in Section 2.1, 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 than the original node — that is, descendants specify more non-wildcard values for fields than the original node.

Consider the node `{(up *)}` $\Rightarrow$ `{(* *)}`. Several of the children of that node are shown below:

```
{(up host)} => {(* *)}            {(up *), (down *)}   => {(* *)}
{(up *)}    => {(down *)}         {(up *), (* host)}   => {(* *)}
{(up *)}    => {(* router)}       {(up *), (* router)} => {(* *)}
```

The three children in the left column were generated by specifying a value for a single field that was wildcarded in the parent. The three children in the right column were generated by adding a new PIE containing a single non-wildcard to the precursor.

The search is made systematic, thereby obtaining the efficiency gains described in Section 2.2, 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 *)}` $\Rightarrow$ `{(down *)}`. The right-most non-wildcard in this rule is `down` in the successor. Therefore, `{(up *)}` $\Rightarrow$ `{(down router)}` is a valid child, but `{(up router)}` $\Rightarrow$ `{(down *)}` would not be generated because it requires adding a non-wildcard to the left of `down`.[3]

2.3.2 Counting Co-occurrences

MEDD's search through the space of dependencies is guided by a best-first heuristic. Each time it generates a node, MEDD scans its historical event logs, counting the number of times the precursor and successor of that node co-occurred. 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.

When should an occurrence of precursor x and an occurrence of successor y count as a co-occurrence? Clearly, the temporal proximity of the occurrences of x and y is important. We are more inclined to believe that x and y are related if they occur within a few seconds of each other rather than a few hours. One approach to determining when x and y co-occur is to specify a fixed-size temporal window of width δ, pass the window over the historical event log that MEDD uses to generate rules, and count any occurrences of x and y within the window as co-occurrences. That is, if x occurs at time t, and y occurs between $t-\delta$ and $t+\delta$, then x and y are said to have co-occurred. This approach has the disadvantage of treating events that fall just outside the temporal window as being unrelated to those that are inside the window. Suppose event e_1 occurs at time t, e_2 at $t+\delta$, and e_3 at $t+\delta+\delta/1000000$. The method above would treat e_1 and e_2 as related, and e_2 and e_3 as related, but would not treat all three events as related.

To avoid the problem with fixed-size windows described above, MEDD takes a slightly different approach. Sets of related events are constructed such that each event in a set occurs within δ time steps of at least one other event in that set. Given a historical event log, let e_i be the i^{th} event in the log, and let t_i be the time at which the i^{th} event occurred. To

[3] Interested readers are referred to (Oates & Cohen 1996) for a detailed discussion of the use of this type of operator ordering to achieve systematicity.

construct sets of temporally proximal events, we start with e_1 by creating a set containing only that event and asking whether $t(e_2) - t(e_1) \leq \delta$. If so, then e_2 is added to the set containing e_1. Then we ask whether $t(e_3) - t(e_2) \leq \delta$. If so, then e_3 is added to the set. This continues until $t(e_{j+1}) - t(e_j) > \delta$, where e_j is the last event added to the set. Events e_1 through e_j form a complete set, and a new set is created containing only e_{j+1}. The process repeats (comparing $t(e_{j+2})$ and $t(e_{j+1})$, etc.) until the last event in the log is assigned to a set. Consider the following event log, which gives the times of occurrence of six events:

```
(e1  10) (e2  15) (e3  24) (e4  50) (e5  75) (e6  80)
```

Given $\delta = 10$, MEDD would form the following sets of related events: $\{e_1, e_2, e_3\}, \{e_4\}, \{e_5, e_6\}$. MEDD forms sets of temporally related events in the historical event logs prior to starting its search. When counting co-occurrences of precursors and successors for the purpose of guiding the search, MEDD simply checks for occurrences inside those sets.

Counting co-occurrences of precursors and successors is complicated by the fact that they can comprise multiple PIEs. Recall that an event pattern is said to match a fragment of a log (i.e. occur in that fragment) if each of the PIEs in the pattern can be matched on a different event. That is, we are only concerned with whether a set of PIEs co-occur within one of the sets of temporally proximal events, not whether they occur in any specific order. Log fragments are obtained for matching by finding all sets of events within the logs that span no more than δ time steps. 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 then 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, 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 avoid this combinatorial problem by simply trying to match the elements of the precursor sequentially and, if a complete match is found, doing the same with the successor. This scheme may miss some matches, but it is computationally efficient.

2.4 Rule Post-Processing and Application

MEDD returns all 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 and set at 200.

To perform event correlation in real time, the remaining rules are matched against incoming events in sorted order. That is, the rule representing the strongest dependency that matches a new set of events is used to cluster those events.

3 Performance

To test the performance of MEDD, we created simulated networks with a modified version of the Netsim network simulator (which is publicly available from MIT (Heybey & Robertson 1994)), introduced faults into those networks, and generated event logs based on those faults. For each experimental trial, we created an event log, applied MEDD to the log, produced event correlation rules, and then tested those rules on a second event log created from the same simulated network. We examined MEDD's performance under baseline conditions and under variations in the size of the training log (N), the size of the window (δ), and the size of the simulated network (T). Baseline parameter values were $N = 100$, $\delta = 0.175$ seconds, and $T = 16$ nodes.

We evaluated MEDD by recording various measures of performance for ten different runs of the simulator and averaging the results. MEDD identified an average of 39.1 clumps, sets of events that were each generated in response to a different single fault, in logs of 100 events. In actuality, the logs contained an average of 54.3 clumps. That is, there were about 54 faults that each generated two or more events that could be grouped into a clump. MEDD correctly identified 37.8 of those clumps (true positives), missed 16.5 clumps (false negatives), and mistakenly identified 1.3 clumps (false positives). On average, the correctly identified clumps contained 85.8 events out of a possible total of 100 events. Of those, 79.8 were correctly included in clumps (true positives), 20.2 were missed (false negatives), and 6 were incorrectly included in a clump (false positives). To summarize: in the baseline trials, MEDD correctly identified 70% of the true clumps, and less than 5% of its identified clumps were spurious.

Next, we evaluated MEDD's performance in relation to this baseline by varying the number of events in the training log (N), the size of the window used to locate co-occurring events (δ), and the size of the network (T). Each of these factors was varied individually, and the effects on performance were recorded.

Surprisingly, MEDD can construct accurate event correlation rules from very small event logs. When we examined the rules constructed by MEDD and how they are used during event correlation, we discovered that a few rules often formed the majority of the clumps. As long as the event log is sufficiently large to support construction of these dominant rules, a large number of true positive clumps will be found.

Equally surprisingly, very large samples (e.g., $N = 500$) do little to increase the percentage of true clumps found. A manual review of the event logs indicates that this "ceiling effect" is partially caused by the representational limitations of MEDD's event correlation rules. Some types of relationships cannot be represented by MEDD rules, and thus cannot be located by the algorithm. For example, MEDD cannot represent the proximity of network elements. Thus, MEDD can only construct rules that apply to all hosts or to one specific host. Similarly, MEDD cannot represent classes of applications, and thus can only construct rules that apply to all applications or one specific application. In both these examples, the inability to form PIEs at the correct level of abstraction reduces the G value of rules that use the only alternative — PIEs that are too general or too specific.

Despite its relatively modest effect on the number of true positive clumps found, increasing N does reduce the number of false positive events in true positive clumps. Using very small event logs more than doubles the average number of false positive events per clump.

However, this effect quickly levels out, so that training logs with more than 100 events have no discernible effect on the average number of false positive events per clump.

The window size δ can have a dramatic effect on the percentage of true positive clumps found by MEDD. Window sizes that are too small make it difficult to form any useful event correlation rules, because related events are not included within a single window, and thus cannot be used to form a rule. Instead, MEDD constructs many spurious rules that associate unrelated events that occur at nearly the same time in the training log. Window sizes that are too large produce rules that form a few very large clumps, missing many smaller clumps that should be identified separately.

4 Conclusion

In the base case, applying MEDD rules produced a 25% reduction in the number of reports that would arrive at a manager's console. That is, when MEDD clumps were substituted for their constituent events, the resulting number of items (events and clumps) was only 75% of the original number of events in the original testing log. MEDD requires relatively small training logs (e.g., 100 events) to form the event correlation rules responsible for these reductions. In nearly all cases, only one percentage point of this reduction was due to false positive clumps (in the worst case examined, with very small window sizes, MEDD reduced the overall number of events by 15%, and four percentage points this reduction were due to false positive clumps).

However, there is substantial room for future work in two areas. First, MEDD should be evaluated much more thoroughly. We currently have little idea whether MEDD will perform well on actual network data. Actual networks may have very different failure patterns than those we were able to produce in simulated networks. Real networks may also be much larger and more varied in their topology, interconnectedness, and types of elements than were our simulated networks. Event logs of failures in actual networks will be needed before MEDD can be realistically evaluated.

Second, MEDD itself could benefit from additional development. The ability of MEDD rules to detect certain types of event correlations could be improved if PIEs could abstract events in a greater variety of ways. For example, providing a hierarchy of network element types and providing information on network proximity would greatly expand the types of abstraction possible in PIEs.

Similarly, additional characteristics of hosts, applications, and communication links could be used by MEDD to expand the range of rules it can consider. Finally, methods for automatic determination of window size should be explored because MEDD's accuracy depends critically on this parameter. We can reasonably expect managers to provide some information about the likely temporal proximity of related events, but automatic determination would still be desirable.

References

Feldkuhn, L., and Erickson, J. 1989. Event management as a common functional area of open systems management. In *Proceedings of the First IFIP Symposium on Integrated Network Management*, 365–376.

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

Oates, T., and Cohen, P. R. 1996. 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.; Gregory, D. E.; and Cohen, P. R. 1994. Detecting complex dependencies in categorical data. In *Preliminary Papers of the Fifth International Workshop on Artificial Intelligence and Statistics*, 417–423.

Oates, T.; Schmill, M. D.; and Cohen, P. R. 1997. Parallel and distributed search for structure in multivariate time series. To appear in *Proceedings of the Ninth European Conference on Machine Learning.*

Riddle, P.; Segal, R.; and Etzioni, O. 1994. Representation design and brute-force induction in a boeing manufacturing domain. *Applied Artificial Intelligence* 8:125–147.

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

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

Schrodt, P. A. 1994. Event data in foreign policy analysis. In Neack, L.; Hey, J. A. K.; and Haney, P. J., eds., *Foreign Policy Analysis: Continuity and Change.* Prentice-Hall. 145 – 166.

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