Finding Dependencies in Event Streams Using Local Search[1]

Adele E. Howe
Computer Science Department
Colorado State University
Fort Collins, CO 80523
howe@cs.colostate.edu

The problem of inferring causality from empirical observations has been well studied. Several approaches are notable for efficiently constructing complex causal models of the inter-relationships between variables (e.g., [8,3,2]). These approaches tend to rely on correlations and co-variances among the variables as the basis for inferring causality. However, for some applications, the available data are categorical observations over time (e.g., event streams or execution traces of programs); for example, patterns in execution traces form the basis of several methods of debugging software (e.g., [1,4]). These applications are less amenable to solution by methods based on correlation and co-variance.

An alternative approach, called *Dependency Detection*, searches the event streams for often recurring sequences [6]. The set of recurring sequences (called *dependencies*) indicates events that commonly co-occur and forms a weak model of causality.

Although promising, dependency detection is limited in several ways. First, the underlying search is exhaustive, looking for all possible dependencies in the data. As a consequence, the computational complexity increases exponentially with the length of the sequences. Second, the sequences are rigid, which means that only exact matches count and that any noise (i.e., insertion of some other unrelated event into the stream) will not count as an example of the sequence. Third, the technique considers only a single stream rather than multiple streams of parallel events; Oates et al. have developed a technique for multi-stream dependency detection [7]. This paper describes how local search with flexible matching has been used to overcome the first two limitations.

1 Finding Significant Sequences

The goal is to find sequences that repeat with unusual frequency in event streams. This section describes the basis for finding these sequences (dependency detection) and enhancements to the basic method that find longer sequences that are insensitive to minor variations (*local search dependency detection*).

1.1 Dependency Detection

Dependency Detection applies contingency table analysis to build a set of simple models of causes of a particular event's occurrence. It was designed to be the first step in a partially automated debugging process, which has been demonstrated on the Phoenix planning system [5]. Consequently, the algorithm and the examples refer to failure recovery actions of the Phoenix planner and the failures that occurred during the planner's execution.

[1]This research was supported by a Colorado State University Diversity Career Enhancement grant and ARPA-AFOSR contract F30602-93-C-0100. I wish to thank Darrell Whitley for his advice on the application of local search, Aaron Fuegi for his programming, Steve Leathard for running tests, and Tim Oates for the synthetic data generator.

	F_b	$F_{\overline{b}}$
R_{aa}	5	10
$R_{\overline{aa}}$	10	100

Table 1: Sample contingency table from Dependency Detection for sequence $[R_{aa}F_b]$

Dependency detection searches Phoenix execution traces for statistically significant co-occurrences of particular precursors leading to a target event (i.e., a failure). The execution traces are sequences of alternating failures and the recovery actions that repaired them, e.g.,

$$F_a \rightarrow R_{aa} \rightarrow F_b \rightarrow R_{zz} \rightarrow F_a \rightarrow R_{aa} \rightarrow F_b.$$

The idea is that the appearance of repetitive sequences, such as $R_{aa} \rightarrow F_b$, may designate early warnings of severe failures or indicate partial causes of the failures.

Dependency detection exhaustively tested the significance of all possible sequences of precursors and failures using contingency table analysis. Precursors combined with failures form three types of sequences: a failure precursor (written [F F]), a precursor of a failure and the recovery action that repaired it ([FR F]), or a recovery action precursor ([R F]); the failure must immediately follow the precursor in the sequence.

The dependency detection algorithm has three steps.

1. Gather execution traces.
2. For each combination of possible instantiations of the three types of precursors and each possible failure, a contingency table is built and tested for the appearance of the sequence in the execution traces.
3. Each significant combination is tested for overlap with other significant sequences.

First, the algorithm gathers the data, which is alternating failures and recovery actions. Second, it steps through all possible combinations of the three types of sequences. For each sequence, it arranges four counts from the execution traces in a 2x2 contingency table and applies a G-test to the table. Figure 1 shows an example contingency table for the sequence $[R_{aa}F_b]$. The four counts are: the number of times the precursor is followed by the target failure (upper left cell) and not followed by the target failure (upper right cell), as well as when other precursors are followed by the failure (lower left cell) and not followed by it (lower right cell). A G-test of the example table produces $G = 6.79, p < .009$, suggesting a dependency between the two events. For each significant combination ($\alpha < .05$), the third step prunes out overlapping combinations (i.e., target failure is the same and one precursor is a subset of the other) using a Homogeneity G-test. The process results in a list of all significant sequences found in the execution traces.

Computationally, dependency detection requires a complete sweep of the execution traces to test each sequence. The number of sequences equals the number of types of failures squared (for FF) plus the number of failures squared times the number of recovery actions (for FRF) plus the number of recovery actions times the number of failures (for RF). In total, this is roughly $M * N^L$ where M is the length of the event traces, N is the number of types of events in the traces, and L is the length of the sequence desired.

1.2 Local Search for Dependencies

Dependency detection is limited primarily by its reliance on exhaustive search of possible sequences. This was feasible initially because the search space included only all possible immediately

preceding influences on failure; the traces contained at most eleven types of failure and eight recovery actions, producing, at most, 1177 possible combinations.

The original algorithm was modified to include more flexible matching and a reduced search through local search. In addition, event streams may be composed of many types of events, which need not alternate as before. The events can be of different values for several types (e.g., in Phoenix, we had recovery actions and failures as types and then each of those had eight and eleven values respectively). Any set of events with time stamps for ordering is a legal event stream.

1.2.1 Improving the Matching

To make the matching insensitive to insertion of random events, the contingency table construction was changed to match a *relative order* sequence within a window. As before, the sequence consists of a precursor and a final target event. For example, we could use a precursor of four recovery actions or failures and a target of a particular failure. For the construction of a contingency table, we need to determine whether the precursor matches (thus, we add one to a count in the top row of the contingency table) and whether the target event matches (in which case we add one to a count in the left column of the table).

In this algorithm, the precursor is a relative order of a specified number of events. A *window* designates an area over which the algorithm will match the precursor. A sequence matches a given window position when the target event is at the last position and when the elements of the precursor appear in the window in the same *order* as they do in the sequence. If the window is exactly the length of the precursor, then the match must be exact as in the original dependency detection algorithm; if the window length is greater than that of the precursor, then the match checks the relative order of the constituents of the precursor. For example, if the precursor is [A B] and the window is of length three, then the precursor will match a portion of the event streams in which A is followed immediately by B or A and B are separated by a single event.

The algorithm for constructing the contingency table is:

1. Initially position the match window at the first position in the event streams in which the window's last position (the target) is over an *event of the same type* and at least $W-1$ events proceed it.
2. Repeatedly re-position the window as follows until the end of the event stream is reached:
   (a) Position the window with its last position over the next event of the same type as the target event.
   (b) If the target event matches the last position, then prepare to add to a cell in the left column; otherwise, prepare to add to the right column.
   (c) If the precursor matches within the window, then add one to the top row and column indicated by the last step of the contingency table; otherwise, add to the bottom row and column indicated as before.

For a given sequence, incidence counts are gathered by sliding an imaginary match window over the event streams. The window defines an area of length W. By default, this length is $2P+1$, which indicates that twice as many elements can be found in the window as are expected to be part of the precursor. In other words, we can allow P spurious events to slip between salient events. The window is re-positioned by moving its last position (the target) to the next event of the same type as the target. As in the original formulation, four counts are tallied for the contingency table: precursor with target, precursor without target, not precursor with target and not precursor without target. Each window positioning contributes to one and only one of the four positions in the contingency table. Figure 1 shows four positionings of the sliding window and their contribution

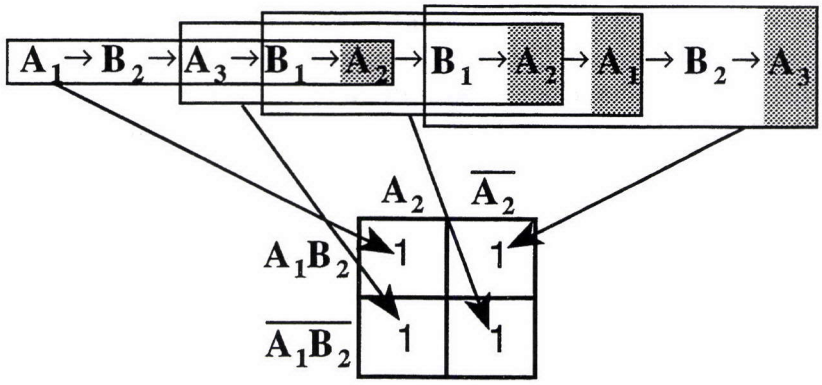

Figure 1: Sliding a window to construct a contingency table.

to the contingency table for the $[A_1\ B_2,\ A_2]$ sequence (the target is of type A and has the value A_2). After sliding the window through all event streams, a G-test is run on the resulting contingency table.

1.2.2 Reducing the Search

The dependency detection algorithm also was modified to apply local instead of exhaustive search. In this case, local search refers to a process of starting with a randomly selected initial sequence and applying operators to find the best sequence from that one; the search continues until no better sequence can be found through operator application. Multiple initial starts of local search are done to collect different dependencies and explore different parts of the search space.

Local search has several advantages over exhaustive search for this application. First, we can control how much time is devoted to search through the operators applied and the number of trials. Second, we can be assured of getting the strongest dependencies; local search prunes the number of dependencies to be considered and usually prunes spurious dependencies based on few data.

The algorithm incorporates local search as follows:

1. Gather event streams of salient events.
2. Select a pattern for the sequence: a specific target event, a length for the precursor (P) and a length for the window (W).
3. For each of T trials
   (a) Construct an initial sequence by randomly selecting P events in some order.
   (b) Find the best sequence by searching from the initial sequence:
      i. Apply the permutation operator to the best sequence found so far.
      ii. For each permutation, test its significance by constructing a contingency table for the relative order matching and applying the G test to it.
      iii. Select the best permutation.
      iv. If the permutation is no better than the previous best, then return it; otherwise, repeat from step i.
4. Return a list of the best sequences found in each trial.

For the second step, as described in the previous section, sequences may be relative order patterns of a specified length with a specific target event. The window length indicates the number of allowed spurious events. For the third step, the search finds the best related sequence: the one

Failure	1	2	3	4	5	6	7	8
# unique	14	10	5	6	8	8	10	14
# $p < .01$	9	2	3	3	8	1	3	7
Avg. depth	5.7	4.8	6.8	5.3	5.8	5.5	5.0	4.8

Table 2: Results of local search for 8 target failures with precursors of length 4, window of length 7 on a Phoenix data set

with the highest G value for its contingency table. For each of the T trials, the initial sequence is composed of a precursor of length P with the elements chosen randomly plus the target event at the end.

New sequences are generated by applying a local search operator to the current best. Several local search operators were considered that permute the sequence by reordering and replacing sequence elements, but the best operator, in terms of efficiency and efficacy, is 1-OPT. 1-OPT optimizes one element in the sequence at a time. Progressing through the sequence positions in order, this operator considers all available elements as replacements; the operator replaces the element originally in the position with the best element for that position by testing the possibilities.

The algorithm continues to apply 1-OPT to subsequent best patterns until the resulting G value cannot be improved. The G values are obtained by constructing contingency tables and applying the G test as was described in the last subsection. The output is a list of the sequences returned in each of the T trials.

2 Results

Earlier examinations of the results of exhaustive dependency detection suggested that dependencies with most of their elements in common tend to both be significant. This characteristic is exploited by local search.

2.1 Finding Dependencies in Phoenix Data

The efficacy of local search was tested on some data sets from Phoenix. The results were similar in each data set and are reported for a data set with 946 events (alternating failures and recovery actions) in its event streams. For each of eight target failures (those that appeared in the event streams at least five times), the algorithm started with 100 randomly generated precursors of length three using a window size of seven and searched for the highest rated neighbor that could be reached from each. Each precursor included either failures or recovery actions for 18 possible values in each precursor position. The searches found a significant dependency in about one-third of the trials, and half were highly significant ($p < .01$). As would be expected, many of the dependencies were found in more than one trial and significant dependencies tended to be related to other significant dependencies. Table 2 summarizes the results for each target failure (designated by number) by number of unique dependencies found, number of dependencies found with $p < .01$, and the average search depth to find the dependencies. The search depth suggests a strong relationship between significant sequences and indicates the number of times the operator was applied. The 100 trials on the eight failures required three hours of CPU time on a SPARC IPX with the code written in Lucid Common Lisp.

Searching for length four precursors exhibited similar, if slightly less successful results. Highly significant dependencies are found in, on average, 18 out of 200 trials. Table 3 summarizes the results for each target failure given local search with precursor of length four and window size of nine. In this case, the search tends to proceed much further, but this is balanced out by many of the initial starts not appearing in the data and many of their neighbors not appearing in the data either. Not too surprisingly, the 200 trials on the eight failures required twice as much computation (six hours).

Failure	1	2	3	4	5	6	7	8
# unique	22	19	18	12	16	20	9	29
# $p < .01$	14	10	12	7	11	4	4	15
Avg. depth	8.5	7.3	7.6	7.5	8.1	7.0	6.9	6.8

Table 3: Results for precursors of length 5, window of length 9

As would be expected, the dependencies found exhibited a high degree of overlap. About 50% to 75% of the length three precursors are substrings of the length four precursors.

2.2 Finding Dependencies in Synthetic Data

In addition to the Phoenix data, the local search algorithm was tested on finding dependencies in synthetic data. The synthetic data was used to probe a concern: Can local search find dependencies even when there is no relationship between similar sequences?

The synthetic data was produced by generating token streams with biases. For this test, the bias was three sequences that were selected in advance. For each sequence, the precursor would appear at each position in the event streams with a likelihood of .05; if the precursor was inserted, then the target event followed it with likelihood of .80. Twenty seven synthetic data sets were generated, which included three numbers of tokens (15, 25, 35), three stream lengths (100, 500 1000), and three sequence lengths (2, 3, 4). Local search found all the sequences of length 2 and 3 with 15 or 25 tokens and at least 500 events in the streams, but none of the sequences of length 4 with 15 or 35 tokens. In addition, few of the sequences were found when there were 35 tokens. These results suggest that local search can detect relative order dependencies even when the neighborhoods have less meaning. However, increasing numbers of tokens and length of precursor means that local search is less likely to effectively search the space.

3 Conclusions

From the point of view of someone interpreting dependencies, one problem with generating long dependencies is that it is easy to be overwhelmed with the number found. Local search addresses this problem in two ways: First, only the most significant dependencies are likely to be found. Second, the search neighborhood in local search provides a natural clustering: the basins of attraction around most locally optimal patterns. These basins ensure that only one of the set of similar dependencies will be returned, and they can be exploited to cluster dependencies that contain the same subsequences.

Subsequences that are repeated across dependencies are probably themselves significant or may indicate cases in which several events exert a similar influence. For example, in the Phoenix data, some of the failures were variants on the same problem (e.g., two of them were failures in calculating

paths), and some of the recovery actions were variants on the same method (e.g., two of them replanned, but from different points in the plan). We would like to cluster related dependencies to find commonalities and to reduce the burden of looking over a long list.

Clusters can be formed by finding locally optimal sequences and then backing out of the basin at individual positions to find other highly significant related sequences. The algorithm was augmented to "back out" of local maxima and gather sequences from the neighborhood that were also significant. If we examine sequences local to the maxima found in the test with precursors of length three and window of length seven, then we find that at each position in the sequence at least one and as many as six out of sixteen other sequences are significant. With local search, we can find clusters of somewhat less significant but highly similar composition.

Local search dependency detection is data directed, finds highly significant dependencies and provides a convenient method of clustering. As demonstrated on the Phoenix data, the local search method is effective at detecting long dependencies in data with considerable overlap in the elements of the dependencies: it finds a large number of the most significant dependencies in a manageable amount of computation time. However, not too surprisingly, when tested on synthetic data, it does not perform nearly as well when there is little relationship between subsequences and the long sequences that form dependencies.

References

[1] Peter C. Bates and Jack C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. Department of Computer and Information Science 83-29, University of Massachusetts, Amherst, MA, March 1983.

[2] Paul R. Cohen, Lisa A. Ballesteros, Dawn E. Gregory, and Robert St. Amant. Regression can build predictive causal models. Technical Report 94-15, Dept. of Computer Science, University of Massachusetts/Amherst., 1994.

[3] C. Glymour, R. Scheines, P. Spirtes, and K. Kelly. *Discovering Causal Structure.* Academic Press, 1987.

[4] N.K. Gupta and R.E. Seviora. An expert system approach to real time system debugging. In *Proceedings of the IEEE Computer Society Conference on AI Applications*, pages 336–343, Denver, CO, December 5-7 1984.

[5] Adele E. Howe. Analyzing failure recovery to improve planner design. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 387–393, July 1992.

[6] Adele E. Howe and Paul R. Cohen. Detecting and explaining dependencies in execution traces. In P. Cheeseman and R.W. Oldford, editors, *Selecting Models from Data; Artificial Intelligence and Statistics IV*, volume 89 of *Lecture Notes in Statistics*, chapter 8, pages 71–78. Springer-Verlag, NY,NY, 1994.

[7] Tim Oates, Dawn Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. To appear in *Preliminary Papers of the Fifth International Workshop on Artificial Intelligence and Statistics*, January 1995.

[8] Judea Pearl and T. S. Verma. A theory of inferred causation. In J.A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pages 441–452, San Mateo, CA, April 1991. Morgan Kaufmann.