Discovering Causal Chains by Integrating Plan Recognition and Sequential Pattern Mining

Shreeram Sahasrabudhe, Héctor Muñoz-Avila

Department of Computer Science and Engineering
19 Memorial Drive West
Lehigh University
Bethlehem, PA 18015, USA
{sas4,munoz}@cse.lehigh.edu

Abstract

In this paper we define the notion of causal chains. Causal chains are a particular kind of sequential patterns that reflect causality relations according to background knowledge. We also present an algorithm for mining causal chains from a collection of action traces. We run this algorithm on a real-world domain and observe that causal chains can be computed efficiently by quickly identifying inter-related actions.

Introduction

Developing algorithms for mining sequential patterns over a collection of data has been the subject of an increasing research effort (e.g., (Agrawal & Srikant, 1995; Faloutsos *et al*., 1994; Zaki *et al*., 1998; Han *et al*., 1999; Zaki, 2001)). In these approaches, patterns consist of sequences of events, where the order of these events reflects the order in which they frequently occur in the input data.

We present a new kind of sequential patterns called causal chains. Like sequential patterns, events in causal chains reflect the order in which they frequently occur in the action traces. The key characteristic of casual chains is that each event causes the event that immediately follows it in the pattern. These cause-effect relations are determined by background information.

Causal chains are motivated by work on plan recognition (e.g., (Kautz, 1991)). Plan recognition aims at predicting goals, actions or plans from a set of observed actions performed by an observing agent. The particular kind of plan recognition necessary for computing causal chains is the so-called "keyhole" recognition since the observed actions are independent of the observing agent (e.g., (Albrecht et al, 1997)). The causal chains can be seen as plans that occur frequently in traces of actions. As such, causal chain can be used to predict goals (i.e., the resutls of the plans), and thus, provide a global picture of the actual intention of the actions in the target domain.

In this paper, we make the following contributions: (1) formally define causal chains, (2) present an algorithm for discovering causal chains from input action traces, (3) present results of experiments with a real-world complex domain illustrating that causal chains can be mined efficiently, and (4) observe that sequential patterns are a worst-case scenario for causal chains where every action causes every other action. In realistic situations, however, the number of causal chains is substantially less than the number of sequential patterns.

Target Domain

Although our algorithm for extracting causal chains is domain-independent, we selected the domain of UNIX command traces to focus our experiments for several reasons: (1) it is a complex domain yet the actions (i.e., the UNIX commands) are well defined. (2) It is a realistic domain. UNIX is a wide spread multi-tasking operating system with a broad range of users and commands, and (3) large, realistic data sets are available.

We used the collection of UNIX command traces collected reported in (Greenberg, 1988). It contains command histories of 168 users, totaling thousands commands. Each trace is divided in one of four categories depending on the expertise of the user: novice programmer, experienced programmer, computer scientists and non-programmers. We used the operator definitions for UNIX commands reported in (Golden, 1997). Although these definitions by no means cover the whole spectrum of UNIX commands, it does show that it is feasible to represent the conditions and effects of UNIX commands in a formal language. We did some variations of these commands and added definitions for several that were unavailable.

Problem Formulation

Background Knowledge

In our framework the **background knowledge** is a list of operators. An **operator** consists of (1) a name, (2) a list of arguments, (3) a list of preconditions and (4) a list of effects. The **preconditions** are a list of literals indicating the conditions that must be valid for the operator to be applicable. The **effects** are a collection of literals to be added or deleted; they indicate the changes in the conditions as a result of applying the operator. Table 1 shows the definition of the operator *rm*, which indicates the effects of removing one or more files. The name of each file is an argument. The preconditions indicate that the existence of each file. The effect of *rm* is to delete the condition about existence of the file. We use brackets in Table 1 to represent that the number of arguments, preconditions and effects is not fully known until the operator is matched against a command in the traces. we adopted the principles of the SADL language (Golden, 1997). SADL was conceived for expressing actions whose conditions and effects may only be known during the execution of the action. We define an **action** as an instance of an operator. We view an **action trace** as an ordered collection of actions. For the UNIX command traces data, we refer to the actions as commands

Table 1: Example of the operator rm

```
Operator: rm
   Arguments: [<filename><type = file>]
   Precondition:
         [<filename><type = file><status = exists>]
   Effects:
         [<filename><type = file><status = not exists>]
```

Causal chains

Given a sequence of elements A = ($a_1 \ldots a_n$), a subsequence of k elements in A, ($a^1, \ldots a^k$), is a **k-subsequence** of A if each a^j occurs before a^{j+1} in A.

An action trace T = ($a_1 \ldots a_n$) **supports** a sequence of actions P = ($b_1, \ldots b_m$) if there is a m-subsequence S = ($a^1 \ldots a^m$) of T such that there is a mapping from the arguments of each b_j in P to the arguments of each a^j in S such that if the mapping is applied to P, P and S are identical. In this situation, we say that S is a sequence supporting P.

An action trace ($b_1 \ldots b_m$) is a **Causal chain** if:

($b_1 \ldots b_m$) is supported by at least a certain predefined percentage, called the **minimum support**, of the input action traces.

The m-subsequence ($a^1 \ldots a^m$) of each supporting action trace ($a_1 \ldots a_n$) meets the following conditions:

At least one of the effects of a^i is a literal l_i that is present in the preconditions of a^{i+1}.

There is no action a_k in the action trace that is **clobbering** the literal l_i linking a^j and a^{j+1}. That is, there is no a_k between a^j and a^{j+1} deleting l_i.

The first requirement ensures that the Causal chain and each of its supporting sequences are equivalent applications of the same operator. It is not sufficient to require actions in the pattern and the supporting sequences to be instances of the same operators. The reason is that operators have a variable number of arguments, preconditions and effects. Thus, two instances of the same operator may result in actions that do not match. The next two conditions ensure the causality relation between the actions in the Causal chain. In particular, it is crucial that no actions are clobbering these causality relations. We refer to the m-subsequence ($a^1 \ldots a^m$) of the definition of Causal chain as a **chain**. *The requirement for these causality relations is typical for plan recognition*.

Our definition of Causal chain follows the *principle for mining sequential patterns* stated in (Agrawal & Srikant, 1995) and others (e.g., (Zaki, 2001)) of counting support from each trace once even if the same pattern appears more than once. Also from (Agrawal & Srikant, 1995; Zaki 2001), we will identify Causal chains that are maximal: A Causal chain, C, of length k is a **maximal** if there is no other Causal chain, C', of size h, such that C is a k-subsequence of C' and h > k.

Example of a Causal Chain

Table 2 presents two sample action traces. The first trace consists of 4 commands: (1) the file *thesis* is renamed as *thesis.bak*, (2) a symbolic link, by the name *thesis*, is made to the file *thesis* contained in the directory *docs*, (3) the contents of *docs* are listed, and (4) the file *thesis* in the directory *docs* (pointed by the symbolic link) is copied into the file named *thesis1*. The second trace consists of two commands: (1) a symbolic link, by the name *paper*, is made to the file *paper* contained in the directory *papers*, and (2) a copy of the file *paper* in the directory *papers* (pointed by the symbolic link) is made into the file named *paper1*.

Table 2: Two samples of action traces

1	(mv thesis thesis.bak, ln –s docs/thesis, ls docs, cp thesis thesis1)
2	(ln –s papers/paper, cp paper paper1)

If the minimum support is 100%, the only Causal chain is (ln –s papers/paper, cp paper paper1). This pattern is supported by the 2-sequence (ln –s docs/thesis, cp thesis thesis1) of the first trace and the 2-sequence (ln –s papers/paper, cp paper paper1) of the second trace. The pattern (ln –s docs/ thesis, cp thesis thesis1) is considered equivalent to (ln –s papers/paper, cp paper paper1) since there is the mapping of their arguments, {docs → papers, thesis → paper, thesis1 → paper1}, making these two patterns identical.

Algorithm for mining causal chains

We refer to the m-subsequence (a^1 … a^m) of the definition of Causal chain in Section 3.2 as a **chain**. A maximal Causal chain can be seen as a chain that is maximal (i.e., the chain is not a k-subsequence of another chain with more than k elements) and that has the minimum required support. The idea of the algorithm is to follow a bottoms-up process whereby chains are constructed for each action trace independently and then chains from different traces are matched to find those that have the required minimum support.

The algorithm for mining maximal Causal chains begins by processing the input action traces to instantiate the commands with their corresponding operators (Section 4.1). The next step is where the chains are identified for each trace (4.2). To improve efficiency and remove redundancy we perform a cleansing step where equivalent chains are detected and removed (4.3). Finally, maximal Causal chains are mined by determining which maximal chains have the required minimum support (4.4).

Action Traces Instantiation

The data files contain the input action traces. In this phase, these files are parsed. During parsing, each command is instantiated with its matching operator to determine the arguments, preconditions and effects. The output of this phase, are the traces consisting of the instantiated operators.

Table 3: Example of a command in input traces

C rm sam
D /user/grads/xxx
A NIL
H NIL
X NIL

Table 3 shows an example of a UNIX command entry in the input action traces as they appear in (Greenberg, 1988). Every command is annotated with 5 fields labeled with initials C, D, A, H and X. The field C corresponds to the command as typed by the user. In this example the command type is remove *sam* (*rm sam*). D indicates the current directory where the command was executed (*/user/grads/xxx*). A indicates the alias of the command (*NIL* indicates that no alias was used). H indicates if the line was retrieved through history. X indicates if the command was executed successfully (anything different than NIL indicates failure). Thus, the reading of the command in Table 3 is that the file *sam* was removed from the directory */user/grads/xxx*

Table 4: Instantiated operator *rm*

Name: rm
---Arguments:---
Name: sam; Type: F
---Preconditions:---
Name: /user/grads/xxx; Type: P; Status: E
Name: sam; Type: F; Status: E
---Effects:---
Name: sam; Type: F; Status: NE

Our program parsing the commands first checks if there is no error. If no error occurs, we lookup (1) the command name (e.g. rm), (2) any command options (i.e., labeled by a - sign following the command name), and (3) the number of arguments. These three elements determine what the command is actually doing and how the matching operator instantiates the command. For example, the move command (mv) has different effects if called with two arguments than if called with more than two arguments; when called with two arguments, the first one is renamed to the second. But when called with more than two arguments e.g. "*mv f1 f2 … fn d*", the last argument, *d*, is a directory to which the files *f1 … fn* are moved.

The arguments and the directory field (labeled D in Table 3) are used to instantiate the arguments, preconditions and effects of the operator. Each instantiated operator is added to the traces. The traces containing the sequence of instantiated operators are the output of this phase. Table 4 shows the instantiated operator for the command shown in Table 3 and the operator shown in Table 1. The label F in Type indicates that the argument is a file (other possibilities include D for directory, P for the current path of the user, SL for soft link). The status can be E meaning that the file exists or NE meaning that the file does not exists.

An issue in this phase is when parsing commands where no operator definition is available. A typical example of such a command is the UNIX command *make*. This command executes a script issuing other commands typically used in connection with compiling pieces of a code. We currently ignore these commands but we acknowledge that this is a limitation and intend to address this in the next phase of our research. One possibility is to assume that any command for which no operator definition is known can cause any subsequent command.

Identification of Chains

This phase receives as input traces of instantiated operators. The goal of this phase is to identify chains for each trace. For each command, C, we construct what we called a chain-set for C. The **chain-set** for a command C, is the set of all chains that start with C.

We perform this chain identification process for each trace independently. The basic algorithm for our implementation is presented below:

1. For each command C_j ($1 \le j < N$) in the trace we construct a chain-set, initially consisting of a single 1-sequence containing C_j. Then for each Chain Set we perform Step 2.
2. For each command N_j occurring after C_j (i.e., $i+1 \le j \le N$) we perform Steps 3 and 4
3. Compare N_j with the last command L of each chain CH_m in the chain-set of C_j. If any effect e of L is a precondition of N_j and e is not clobbered between L and N_j then create a new chain, $CH_m + N_j$, and add it to the chain-set of C_j.

The algorithm will basically create a chain-set for each command. We only create chain-sets for new commands (i.e., there are no two chain-sets for the same command).
Given a trace of instantiated operators, the cause-effect relations define a partial order between the actions in the trace. Algorithms have been proposed in the AI planning literature to compute such partial order for an input plan (e.g., (Veloso and Carbonell, 1993)). The chain identification process computes all total ordered subsequences.

Removing Equivalent Chains

In this phase we remove equivalent chains from each of the k-Chain Sets ($1 \le k < N$). A chain $C = (a_1 \dots a_k)$ is **equivalent** to another chain $C' = (b_1 \dots b_k)$ if there is a mapping from the arguments of each a_i in C to the arguments of each b_i in C' such that if the mapping is applied to C, then C and C' are identical. When an equivalent chain is removed its predecessor chains are removed as well. The reason for this is that, any h-subsequence of C (with $h < k$) must be equivalent to a h-subsequence of C'.

Determining Support for Maximal Chains

In the final stage of our algorithm, the maximal chains for the different traces are compared looking for equivalent chains. If for a given chain $(a_1 \dots a_n)$, the required minimum support is not met, the algorithm tries with the predecessor chain $(a_1 \dots a_{n-1})$ and continues until 20 chains $(a_1 a_2)$.
We now give a concrete equivalent example of equivalent chains**.** Suppose that the following chains from two different traces are being compared:

mv file1 file2 [pre_1/eff_1], cp file2 file3 [pre_2/eff_2], rm file3 [pre_3/eff_3]

mv file40 file23 [pre_4/eff_4], cp file23 file33 [pre_5/eff_5], rm file33 [pre_6/eff_6]

Where pre_i/eff_i are the preconditions and effects of each command according to the background knowledge. For example:

eff3 = [Name: file3; Type: F; Status: NE; Tag: d]

eff6 =[Name: file33; Type: F; Status: NE; Tag: d]

These two chains are equivalent with the mapping:

{file1 → file40, file2 → file23, file3 → file 33}

Experiment

To evaluate our algorithm for extracting causal chains we performed experiments with the UNIX user command traces. The purpose of our experiments was to find all maximal Causal chains for non synthetic domain.

Sequential Mining

A **sequential pattern** is a sequence of actions $P = (b_1 \dots b_m)$ for which a certain predefined percentage of the input action traces (i.e., the minimum support) meet the following condition: there is a m-subsequence $S = (a^1 \dots a^m)$ of the action trace such that the action names, options and the number of arguments of each b_i and a^i are the same.
For the experiment we implemented a simple-minded algorithm for mining sequential patterns that is correct (i.e., it finds sequential patterns only) and complete (i.e., it doesn't leave out any sequential pattern). The purpose is to have a baseline for comparing the number of patterns that can be found with the CES approach.
The algorithm for mining sequential patterns begins with the action traces instantiation process (See Section 4.1). But for sequential mining we only take into account the command names, options and the number of arguments. That is, we don't consider the actual arguments, the preconditions and the effects. The reason is that sequential patterns are mined without the system having the background knowledge. Hence, the comparison between any two sequences is simply done by verifying that they have the same command name, the same option and the same number of arguments. We make sure that there are no repeated sequences. For doing this, we have to check for an exact match in the arguments.

Table 5: User categories for input traces

Categories	Input Action Traces	Trace Length (average)
Computer scientists	17	586.35
Experience Programmers	34	691.41
Novice Programmers	55	348.90
Non-Programmer	25	472.96

Experimental Setup

The experiments were performed on a single Pentium 4 1.6GHz machine with 512MBytes of RAM. Table 5 summarizes the input action traces used in the experiments. The first column describes the user categories. The second column the number of traces for each category and the third column the average size of the traces (number of commands).

We ran experiments for computing Causal chains and sequential patterns for each of the four user categories. For each run we computed the number of maximal Causal chains and sequential patterns. We ran the experiments for support percentages of 100, 70, 40 and 10, for a total of 16 runs.

Users / Support (%)	100	70	40	10
Computer Scientists	31	29	28	26
Experience Programmers	90	80	77	64
Novice Programmer	70	68	65	55
Non-Programmer	40	38	37	33

Table 6: Time (in seconds) for finding the causal chains

The readings for mining sequential patterns were taken for a 1-hour run because of the combinatorial factor would require too much time and resources to compute all sequential patterns. We also mined the Causal chains first. Thus, we knew the length of the maximal patterns for each user category. When running the sequential mining approach we stopped the sequential pattern mining process when we reached sequences of length larger than the length of the maximal Causal chain length. In most runs however, we reach the 1-hour limit before being able to compute all sequence patterns of the maximal length. Thus, the total number of sequential patterns computed is a lower bound for the total number of possible sequential patterns.

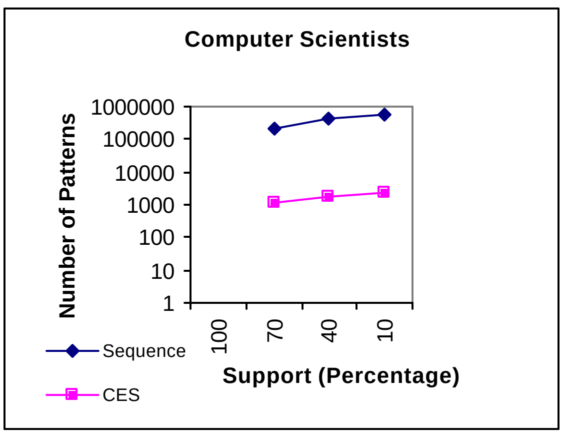

Figure 1: Results for computer scientists

Results

Table 6 shows the run times for the different users and support percentages for mining Causal chains. The support percentages, 100, 70, 40 and 10, are spaced to cover a wide support range. Despite that we didn't emphasize efficiency in our implementation of the algorithm for mining Causal chains, the time that it took for each run was relatively low for the hardware we used and the size of the input data. The worst time was 1.5 minutes but more than half of the runs took less than a minute. Our strategy of constructing chains for each trace independently and then comparing the chains seem to have worked well for this particular input data. In this data (see Table 6), there are relatively few traces considered in each run (up 20) but the traces contained a large number of commands (up to 691).

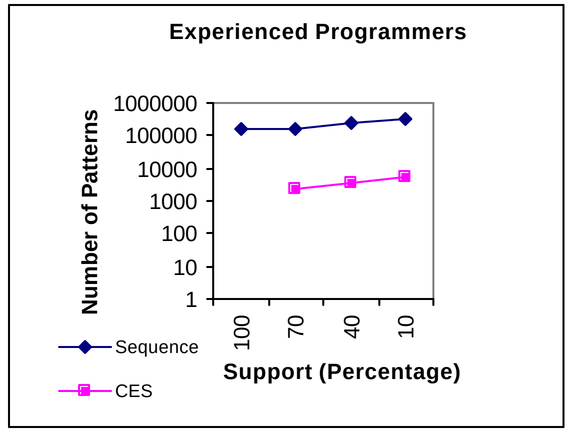

Figure 2: Results for experienced programmers

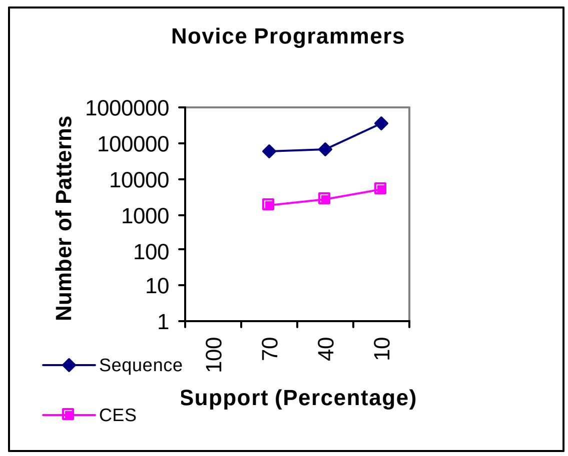

Figure 3: Results for novice programmers

Figures 1-4 show the total number of Causal chains found for each of the four user categories on the logarithmic scale. These figures also show the number of sequential patterns for the same input data and support. As explained before, the sequential patterns found is a subset of the sequential patterns that can potentially be mined from this data. Still, from these readings we observe that the number of Causal chains is substantially less than the number of sequential patterns. Sequential patterns are a worst-case scenario for Causal chains where every action can cause another action. As expected, the number of Causal chains is several orders of magnitude less than the number of sequential patterns for this domain. This illustrates that is feasible to learn all Causal chains even for very large data sets. The crucial point being that non relevant data from the point of view of the cause-effect relationships can be quickly discarded allowing the learning algorithms to concentrate on inter-related actions.

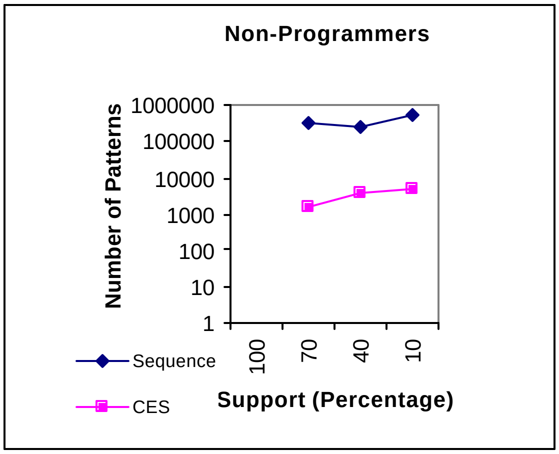

Figure 4: Results for non-programmers

Discussion and Final Remarks

Sequential patterns are a worst-case scenario for Causal chains where every action can cause another action. In our experiments, the number of Causal chains is several orders of magnitude less than the number of sequential patterns. Mining causal chains is motivated by plan recognition work where the aim is to predict goals. In a collection of traces, the goals are the effects of the causal chains. We saw that even for a real-world domain such as UNIX command traces, the number of chains generated is relatively small. Thus, they provide a global picture of the actual intend (i.e., all of the resulting goals) of the traces.

Mining Causal chains can have important applications. Particularly, Causal chains can be used to find patterns from a collection of intrusion logs. These patterns can be used to detect security flaws in a system. Since Causal chains describe structural patterns reflecting causal relations, inter-related actions can be rapidly identified. As a result, Causal chains can be computed efficiently for large collections of input data.

References

Agrawal, R. Srikant, R. Mining Sequential Patterns. In Proceedings of the International Conference on Data Engineering, 1995

Albrecht, D. W., Zukerman, I., Nicholson, A. and Bud, A., Towards a Bayesian Model for Keyhole Plan Recognition in Large Domains, *Proceedings of the 6th International Conference on User Modelling*, 1997, pp. 365-376.

Chien, S.A; Hill, R.W; Wang, X; Estlin, T; Fayyad, K.V. & Mortenson, H.B. (1996) Why Real-world Planning is Difficult: a Tale of Two Applications. In: New Directions in AI Planning, M.Ghallab & A.Milani, eds, IOS Press, Washington DC 1996 pp 287-98

Davison, B., and Hirsch, H. (1998). Predicting sequences of user actions. In Notes of the AAAI/ICML 1998 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis.

Faloutsos, V., Ranganathan, M. and Manolopoulos, Y. "Fast Subsequence Matching in Time-Series Databases". Proc. ACM SIGMOD, May 25-27, 1994, Minneapolis, MN. pp. 419-429

Fikes, R. E., Hart, P. E., and Nilsson, N. J. Learning and executing generalized robot plans. Artificial Intelligence, 3:251-288, 1972..

Golden, K. 1997. Planning and Knowledge Representation for Softbots. Ph.D. Dissertation, University of Washington. Available as UW CSE Tech Report 97-11-05.

Han, J., Yang, Y., and Kim, K. Plan Mining by Divide and Conquer. In Proceedings of the 1999 SIGMOD Workshop on Data Mining. Philidelphia, PA. June 1999.

Kautz., H., A Formal Theory of Plan Recognition and its Implementation, *Reasoning About Plans*, Allen, J., Pelavin, R. and Tenenberg, J. ed., Morgan Kaufmann, San Mateo, C.A., *1*991, pp. 69-125.

Korvemaker, B. and Greiner, R. Predicting UNIX command lines: Adjusting to user patterns. In Adaptive User Interfaces: Papers from the 2000 AAAI Spring Symposium, pages 59-64, 2000

Mannila, H., Toivonen, H., and Verkamo, A.I. "Discovery of frequent episodes in sequences" First International Conference on Knowledge Discovery and Data Mining (KDD'95) 210 - 215, Montreal, Canada, August 1995.

Martelli, A. and Montanari, U. "An Efficient Unification Algorithm". ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2 (April 1982), pp. 258-282.

Veloso, M.M., Carbonell, J.G. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization, Machine Learning, 10:249-278, 1993.

Zaki, M. J., Lesh, N., & Ogihara M. PLANMINE: Sequence Mining for Plan Failures. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98), 1998

Zaki, M.J., SPADE: An Efficient Algorithm for Mining Freque