



The University of Sydney

A Simple and Efficient Algorithm for Hypercycle Detection

TECHNICAL REPORT NUMBER 538

August 2003

Gaurav Pandey
ITT Kanpur, CSE Department

Sanjay Chawla
School of Information Technologies, University of Sydney

ISBN 1 86487 583 6

**School of Information Technologies
University of Sydney NSW 2006**

A Simple and Efficient Algorithm for Hypercycle Detection

Gaurav Pandey² and Sanjay Chawla¹

¹ Knowledge Management Research Group
School of Information Technologies
University of Sydney, NSW, Australia
chawla@it.usyd.edu.au

² Department of Computer Science
Indian Institute of Technology
Kanpur, India
gpandey@cs.iitk.edu.in

Abstract. We present an $O(s^2)$ algorithm for hypercycle detection in a Backward Directed Hypergraphs(BDH), where s is the size of the hypergraph. BDHs provide a representation model for many-one relationships and have found applications in several areas within Computer Science and Operations Research. Our interest in BDHs arise from there application within data mining where they can be used to represent a certain class of association rules, namely rules whose consequents are singletons.

Keywords: Directed Hypergraphs, Association Rules, Cycle-Detection

1 Introduction

A hypergraph is a generalization of a graph where the relationship between the vertices(V) and edges(E) is induced by the power-set on the vertices V , i.e., $E \subset 2^V$. In a traditional graph $E \subset V \times V$.

The advantages of a hypergraph is that they allow a clean representation of many-many, one-many and many-one relationships. Thus hypergraphs are used for modeling circuits on VLSI chips, Bus routes in operations research, and dynamic networks.

2 Motivation

2.1 Association Rules

Association rules are generally described in the framework of market basket analysis. Given a set of items I and a set of transactions T consisting of subsets of I , an Association Rule is a relationship of the form $A \xrightarrow{s,c} B$ where A and B are subsets of I while s and c are the minimum support and confidence of the rule. A is called the *antecedent* and B the *consequent* of the rule. The support $\sigma(A)$ of a subset A of I is defined as the percentage of transactions which contain A and the confidence of a rule $A \rightarrow B$

is $\frac{\sigma(A \cup B)}{\sigma(A)}$. Most algorithms for association rule discovery take advantage of the anti-monotonicity property exhibited by the *support* level: If $A \subset B$ then $\sigma(A) \geq \sigma(B)$.

Our focus is to discover association rules in a more structured and dense relational table. For example suppose we are given a relation $R(A_1, A_2, \dots, A_n)$ where the domain of A_i , $dom(A_i) = \{a_1, \dots, a_{n_i}\}$, is discrete-valued. Then an *item* is an attribute-value pair $\{A_i = a\}$. The ARN will be constructed using rules of the form

$$\{A_{m_1} = a_{m_1}, \dots, A_{m_k} = a_{m_k}\} \rightarrow \{A_j = a_j\} \text{ where } j \notin \{m_1, \dots, m_k\}$$

2.2 Directed Hypergraphs

A hypergraph is a pair $H = (N, E)$ where N is a set of nodes $\{n_1, n_2, \dots, n_k\}$ and $E \subset 2^N$ is the set of hyperedges. Thus each hyperedge e can potentially span more than two nodes. Contrast this with a directed graph where the edge set $E \subset N \times N$.

In the context of association rules, each node corresponds to a frequent item and frequent itemsets are mapped to hyperedges.

In a directed hypergraph the nodes spanned by a hyperedge e are partitioned into two parts, the head and the tail denoted by $H(e)$ and $T(e)$ respectively. A hyperedge e is called *backward* if $|H(e)| = 1$. Similarly an edge is called *forward* if $|T(e)| = 1$. A directed hypergraph is called *B-directed* hypergraph if all its hyperedges are backward. In the rest of the paper we will refer to them as B-graphs. Thus the set of association rules whose consequents are singletons map neatly into a B-graph. Each rule r is represented by a hyperedge e , the antecedents of r by $T(e)$ and the consequent by $H(e)$.

We will also consider the antecedent of a rule as a single entity. For that we define the notion of a hypernode. Given a B-graph B with hyperedges $\{e_1, \dots, e_m\}$, the hypernodes induced by the hyperedge e_i are the tail $T(e_i)$ and the head $H(e_i)$ considered as a single entity. The set of all hypernodes is denoted by V .

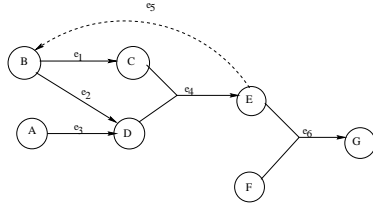


Fig. 1. An example B-graph with all the major features

Example: As can be seen in Figure 1, $N = \{A, B, C, D, E, F, G\}$, $E = \{e_1, \dots, e_6\}$ and $V = \{\{A\}, \{B\}, \{C\}, \{D\}, \{C, D\}, \{E\}, \{E, F\}, \{G\}\}$. Thus F is a node but not a hypernode.

We now define a hyperpath and a hypercycle for a B-graph. A hyperpath is defined as a sequence $P = \{v_1, e_1, v_2, e_2, \dots, e_{n-1}, v_n\}$, where $\forall i, v_i$ is a hypernode and e_i

is a hyperedge. Furthermore for $1 \leq i \leq n - 2, v_i = T(e_i), H(e_i) \in v_{i+1}, v_{n-1} = T(e_{n-1})$ and $v_n = H(e_{n-1})$. A hyperpath is a hypercycle if $v_n \in v_0$.

Again, as can be seen in Figure 1, $P = \{\{A\}, e_3, \{C, D\}, e_4, \{E, F\}, e_6, \{G\}\}$ is a hyperpath and $C = \{\{B\}, e_1, \{C, D\}, e_4, \{E\}, e_5, \{B\}\}$ is a hypercycle.

Finally, we define the size of a hyperpath $|P|$ as the total number of hypernodes appearing on P . Continuing with the previous example, $|P| = 4$ and $|C| = 4$.

2.3 Contributions

Our main contribution is that we propose a $O(s^2)$ algorithm for hyper-cycle detection.

3 Problem Definition

Given A backward directed hypergraph H .

Find All Hypercycles

4 Related Work

5 Preliminaries

5.1 Directed Hypergraphs

5.2 Motivation: Association Rules Network

Association rules are considered a corner-stone of data mining research [?,?]. Association Rules are traditionally described in the framework of market basket analysis. Given a set of items I and a set of transactions T consisting of subsets of I , an Association Rule is a relationship of the form $A \xrightarrow{s,c} B$ where A and B are subsets of I while s and c are the minimum support and confidence of the rule. A is called the *antecedent* and B the *consequent* of the rule. The support $\sigma(A)$ of a subset A of I is defined as the percentage of transactions which contain A and the confidence of a rule $A \rightarrow B$ is $\frac{\sigma(A \cup B)}{\sigma(A)}$. Most algorithms for association rule discovery take advantage of the anti-monotonicity property exhibited by the *support* level: If $A \subset B$ then $\sigma(A) \geq \sigma(B)$.

Our focus is to discover association rules in a more structured and dense relational table. For example suppose we are given a relation $R(A_1, A_2, \dots, A_n)$ where the domain of A_i , $dom(A_i) = \{a_1, \dots, a_{n_i}\}$, is discrete-valued. Then an *item* is an attribute-value pair $\{A_i = a\}$. We are interested in rules of the form

$$\{A_{m_1} = a_{m_1}, \dots, A_{m_k} = a_{m_k}\} \rightarrow \{A_j = a_j\} \text{ where } j \notin \{m_1, \dots, m_k\}$$

The reason we are interested in these specific rules is because we want to characterize projects based on their download activity. Thus our problem is, in the first order, a supervised learning problem where the download activity serves as the dependent variable(class label). But as we have noted in Section 2, the distribution of download activity follows a Zipf-like power law and a direct application of supervised learning techniques will smooth over the skewness. We get around this problem by recursively

constructing a family of interrelated rules. The link between the rules is through their consequent. We begin by fixing an item c and finding all rules whose consequent is c . Then the antecedents of the rules discovered play the role of consequents in the next stage. This way we build a network of association rules. We call this network an Association Rule Network(ARN). The integration of supervised learning and association rules has been the subject of the previous study by Liu, Hsu and Ma [?]. We build upon their work by creating an ARN which flows into an instance of the class label and choosing a different class label in each stage.

Example: Consider a relation $R(A, B, C, D, E, F)$ where the attributes are binary-valued. Assume the following association rules were derived from the relation R using an association rule algorithm

$$\begin{aligned} \{B = 1, C = 1\} &\rightarrow \{A = 1\} \\ \{F = 0\} &\rightarrow \{B = 1\} \\ \{D = 1\} &\rightarrow \{A = 1\} \\ \{F = 0, E = 1\} &\rightarrow \{C = 1\} \\ \{E = 1, G = 0\} &\rightarrow \{D = 1\} \\ \{A = 1, G = 1\} &\rightarrow \{E = 1\} \end{aligned}$$

Suppose we fix the consequent $\{A = 1\}$. Then we can recursively build a network of association rules which flow into $\{A = 1\}$. This is shown in Figure ??(a). Notice the edges of this network are hyperedges. A hyperedge is a generalization of an edge which can span more than two nodes of a graph. As has been noted before [?] a natural way to model itemset transactions is to use hypergraphs. This is because a transaction can consist of more than two items. Also notice that the last rule is not part of the ARN. This is because once we fix the first consequent $\{A = 1\}$ we want to exclude rules in which this consequent appears as an antecedent.

Definition 1. A Hypergraph $G = (V, E)$ is a pair where V is the set of nodes and $E \subset 2^V$. Each element of E is called a hyperedge. A directed hyperedge $e = \{v_1, \dots, v_{k-1}; v_k\}$ is a hyperedge with a distinguished node v_k . A directed Hypergraph is a hypergraph with directed hyperedges.

6 Main Result

Theorem 1. Let $H = (V, E)$ be a backward directed hypergraph. Assume the vertex-list V and the edge-list E of cardinality v and e respectively is given. Then

1. $|HL| = O(e)$.
2. The creation of a list of hypernodes(HL) from the hyperedge-list(EL) is of time-complexity $O(e \log(e))$.
3. The creation of the adjacency-list(AL) has time-complexity $O(e^2)$.
4. The creation of the vertex-matrix(VM) has time complexity $O(ev)$.

Data : Hypergraph H , Superset List SL , Adjacency-List AL , A hypernode list HL , node v (Global)

Result : A list of cycles along paths originating at v

```

let  $S$  be an empty stack;
for each element  $u$  in  $HL$  do
  |  $visited[u] = false$  ;
end
for each element  $u$  in  $HL$  do
  | if  $visited[u] = false$  then
    | |  $visit(u)$ ;
    | end
  | end
end

```

input : A hypernode w

```

 $visit(w)$ ;
 $S.push(w)$ ;
 $o = isPresent(w)$ ;
if  $o \neq null$  then
  |  $findCycle(o, w)$ ;
  |  $visit(o)$ ;
end
 $l_1 = SL(w)$ ;
for each element  $v$  in  $l_1$  do
  | if  $visited[v] = false$  then
    | |  $visit(v)$ ;
    | end
  |  $l_2 = AL(w)$ ;
end
for each element  $v$  in  $l_2$  do
  | if  $visited[v] = false$  then
    | |  $visit(v)$ ;
    | end
  | end
 $visited[w] = true$ 
 $S.pop(w)$ 

```

Data : Hypernode w

Result : The first element e on the stack such that $w \in e$

```

temp = S.pop();
Let S1 be a new stack ;
while S.empty() = false do
  | x = S.pop() ;
  | S1.push(x) ;
  | if w ∈ x then
  |   | while S1.empty() = false do
  |   |   | S.push(S1.pop())
  |   |   end
  |   | S.push(temp);
  |   | return x;
  |   end
end
end
while S1.empty() = false do
  | S.push(S1.pop())
end
S.push(temp) ;
return null ;

```

Data : originating hypernode o and ending hypernode u

Result : H with cycle $[o, u]$ resolved

Let c be the list of edges in the cycle detected ;

```

prev = S.pop() ;
while (true) do
  | curr = S.pop() ;
  | if (e = H.isEdge(curr, prev)) ≠ then
  |   | c.add(e);
  |   end
  | if curr = o then
  |   | break;
  |   end
  | prev = curr;
end
breakCycle(c);

```

5. The creation of the super-set list(SL) has time-complexity $O(e^2v)$.
6. Algorithm HCD (Hyper-Cycle Detection) has time-complexity $O(e^3c)$, where c is the number of cycles in H .

Proof:

1. Follows from the definition of a hypernode when in the worst-case $\cap_{e_h \in E}(T(e_h) \cup H(e_h)) = \emptyset$.
2. The list L consists of the head and tail of all the edges in EL . Construction of HL from L requires a duplication removal operation which takes $O(e \log(e))$, since $|L| = 2e$.
3. For each element $v_h \in HL$ we need to find all edges $e_h \in EL$ such that $v_h = T(e_h)$. Thus AL construction requires $O(e^2)$ time.
4. For each $v_h \in HL$ we create its bit-representation: 1 is assigned against each $v'_h \in V$ if $v'_h \in v_h$, else a 0. Hence the operation is $O(e * v)$.
5. For each element $v_h \in HL$ its bit-representation is subtracted from that of each $v'_h \in HL$ s.t. $v'_h \neq v_h$. If the resulting bit-vector does not contain a -1 then v'_h is added to the $SL(v_h)$. Both the subtraction and search for -1 are $O(v)$ operations. Thus the overall complexity is $O(e^2v)$.
6. Given a vertex v_h on a stack there are two ways in an vertex w_h can appear immediately above it. Either w_h is a superset of v_h or $w_h \in AL(v_h)$. The worst case occurs when these two operations alternate, i.e. in the case of a growing hypergraph of the type shown in Figure ??(a) (This is similar to the situation in standard graphs where the worst-case occurs for Depth-First Search when the graph is a uni-directional chain as shown in Figure ??(b)). In this case the maximum size of the stack will be $2e$. Thus the number of search operations will be $\sum_{i=0}^{2e} i$. Each search operation consists of a subset matching of w' (which is at the top of the stack) against an element v' which appears below it. This is equivalent to testing if $w' \in SL(v')$, an $O(e)$ operation. Also the number of pushes and pops will be $2e$. Furthermore in the worst case, for each cycle detected the complete set of stack operations will be repeated. Thus the overall complexity will be $O(e * e * e * c)$.

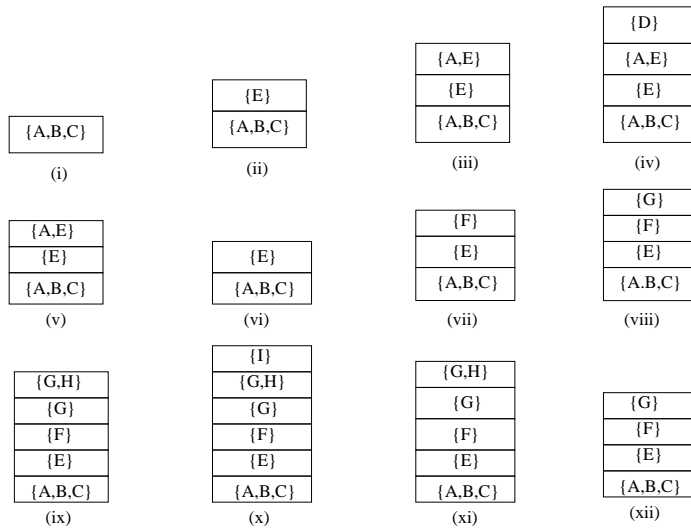
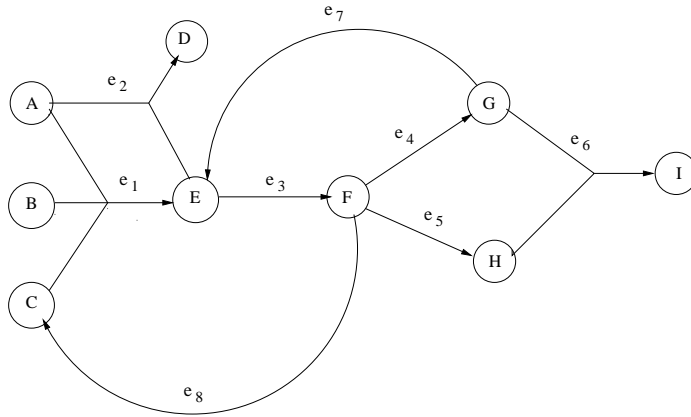
7 Example

- Theorem 2.** 1. If there is a hypercycle in the hypergraph then it will be detected by HCD (completeness).
 2. All hypercycles detected by HCD are indeed hypercycles(correctness).

Proof: Let $C = \{v_0, e_1, \dots, e_n, v_n\}$ be a cycle in the hypergraph H where v_i are the hypervertices and e_i are hyperedges such that $v_n \subset v_0$.

Since C is a hyperpath, for $0 \leq i \leq n - 1$, one of the following must hold:

1. Either $v_{i+1} \in AL(v_i)$ or
2. There exists an element $v \in AL(v_i)$ such that $v_{i+1} \in SL(v)$.



These are also the conditions for v_{i+1} to appear on the stack above v_i . Now, WLOG assume HCD is visiting v_0 , i.e., it has been pushed on the stack. Then by the definition of hyperpaths, v_1, \dots, v_n will appear above v_0 until the condition $v_n \subset v_0$ is met. This is when the hypercycle C will be detected.

{E}
{G}
{F}
{E}
{A,B,C}

(xiii)

{E}
{A,B,C}

(xiv)

{F}
{E}
{A,B,C}

(xv)

{G}
{F}
{E}
{A,B,C}

(xvi)

{F}
{E}
{A,B,C}

(xvii)

{H}
{F}
{E}
{A,B,C}

(xviii)

{F}
{E}
{A,B,C}

(xix)

{C}
{F}
{E}
{A,B,C}

(xx)

{E}
{A,B,C}

(xxi)

{F}
{E}
{A,B,C}

(xxii)

{E}
{A,B,C}

(xxiii)

{A,B,C}

(xxiv)