Stony Brook University

Automatic Generation of Deductive Spreadsheets Using Inductive Learning

A Thesis Presented

by

Marcela Simona Boboila

to The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

August 2008

Stony Brook University

The Graduate School

Marcela Simona Boboila

We, the thesis committee for the above candidate for the Master of Science degree, hereby recommend acceptance of this thesis.

I.V. Ramakrishnan – Thesis Advisor
Professor, Computer Science Department

David S. Warren - Chairperson of Defense
Professor, Computer Science Department

C.R. Ramakrishnan
Associate Professor, Computer Science Department

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Automatic Generation of Deductive Spreadsheets Using Inductive Learning

by

Marcela Simona Boboila

Master of Science

in

Computer Science

Stony Brook University

2008

This thesis presents a tool for automatic generation and display of logic relations in a visual and interactive format. We address a large category of users, with a tool that enforces the underlying programming language in a transparent way, while providing high usability assurances. We use the familiar Excel interface for storing data and manipulating entity properties, and enhance the traditional spreadsheets with the power of deduction, to define and handle relations. While we benefit from Excel's simplicity and ease of use, we increase its capabilities to model real-world scenarios. Our proposed framework maps well on situations where existing data needs to be explored with minimal user effort. An automatic algorithm based on inductive logic programming and a translation procedure from rules to deductive spreadsheet formulas ensures that the general user does not need be familiar with the deductive spreadsheet language.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1 Motivation

This thesis presents a method for automatic generation of deductive spreadsheets. We use inductive logic programming techniques to learn Prolog rules, and a translation procedure that converts the learned rules to equivalent deductive spreadsheet expressions. Our approach is motivated by usability desiderata: the automatic generation process ensures that the programming effort on behalf of the user is minimal. Moreover, the Excel interface is easy to use and appropriate for data visualization.

The rest of this chapter introduces the fundamental concepts that appear in the system design and implementation: inductive logic programming, used to learn Prolog rules from a few examples and some background knowledge, and the deductive spreadsheet paradigm that facilitates the integration of logic with traditional spreadsheets.

1.2 Deductive Spreadsheets

Deductive Spreadsheets (DSS) [3] represent an enhancement of traditional Excel spreadsheets with the power of deductive rules. Similar to an Excel spreadsheet, the DSS interface is a two-dimensional array of cells. However, the rows and columns are labeled with symbolic names. More specifically, the row names represent objects, and the column names represent properties of the objects. For example, the transitive closure problem could be represented in DSS by labeling the rows with node names and the columns with properties (or relations) of the nodes (e.g. 'edge' between two nodes) (see Section 5.1).

Furthermore, the cells in DSS do not contain a single value, as traditional Excel cells, but sets of values (or sets of tuples). As a result, the language capabilities are extended to operations on sets. The DSS formula, which is built upon Datalog relations [6], is similar to deductive rules in that it constructs relations using other relations. We discuss the construction of DSS formulas in greater detail in Section 4.1. Another important feature of DSS is the tabled logic mechanism [5] employed by the deductive engine, which removes the risk of going into infinite loops in the case of recursive definitions.

1.3 Inductive Logic Programming

Inductive Logic Programming (ILP) [14, 20, 31] lies at the intersection of machine learning and logic programming, where the set of logic programs represent the hypothesis space of the learning process. An ILP system uses a set of positive and negative examples, given as facts, and some background knowledge to derive a hypothesized logic program that covers all the positive examples and none of the negative ones.

Some of the existent ILP systems are Progol [21], Golem [22], Aleph [2], FOIL [30], Lime [25], ACE [26], Warmr [27], RSD [28], and DL-Learner [29]. We use the Aleph ILP system to learn the Prolog rules which are subsequently translated to DSS formulas. We give a detailed description of integrating and using Aleph in our system framework in Section 3.1.

There are two approaches of ILP: One is the bottom up approach, which searches the hypothesis space from general to specific and uses techniques such as least general generalization, inverse resolution, and inverse implication (e.g. Golem). The second approach is represented by top-down methods that search the hypothesis space in the inverse direction, from specific to general (e.g. FOIL, Progol, Aleph). Aleph uses the top-down best first search strategy, starting from a bottom clause.

Chapter 2. Related Work

Combining the traditional spreadsheet with logic represents a central research topic for some other proposals as well [7]. Among them, we mention the Knowledgesheet [12] and PrediCalc [9] projects, which also have labeled cells and columns. Unlike DSS, they use constraint specifications [8] to restrict the cell values in the spreadsheet, and work with single-valued cells (not sets) that do not allow recursive calls.

There have been proposals [11] to increase the functionality of Excel as a programming language with user-defined functions, while maintaining the usability of the Excel interface. This approach is similar to ours in terms of its goals: modeling and programming non-trivial scenarios with as little coding involvement on behalf of user as possible. This desideratum motivates their design of user-defined functions and, in our case, the learning of DSS formulas automatically.

The development of expressive set-based languages has also been the concern in [10, 13]. However, our focus is not on the language expressiveness, but rather on the ability to integrate the set specifications with a highly usable visual interface, such as the Excel interface.

Moreover, inductive learning has been successfully applied in several proposals. LINUS [15] is an inductive logic programming system that uses positive and negative examples specified as tuples, and already defined relations in a deductive database to learn relations from real-life noisy databases. Other directions of inductive learning applications include text classification [16, 17] and natural language processing [19] to create semantic parsers or mine relational data. Inductive logic has been applied in bioinformatics [23, 24], and program analysis [18, 32]. We will see in Section 5.2 an example of how our system can also be applied in program analysis.

The novelty of our approach over previous proposals from the literature is the modeling of relational aspects through the familiar Excel interface. The discovery of inductive rules becomes an automatic process that can be visualized and interacted with easily.

Chapter 3. System Architecture

We present a schematic description of the system's architecture in **Error! Reference source not found.**. The DSS learning system uses Excel as a front end, to provide an interface to users. The back end machinery is built upon the XSB Tabled Logic Programming System [1] - a deductive engine used to compute the DSS semantics - and is engineered as a plug-in to Excel. The enhancement of XSB over a standard Prolog system is its capacity to handle cyclic definitions without going into infinite loops. For this purpose, XSB uses tabling to compute correct and finite answers to resolution-based queries [3, 4].

While the users only see and interact with the Excel interface, there are two main components that work in the background and are integrated using XSB:

- ILP component, for Prolog rule learning
- Translation component, to convert learned rules into DSS formulas

3.1 ILP Component

The user provides the input to the ILP component through the Excel interface. Specifically, the user provides positive and negative examples, and background knowledge for rule learning. The ILP system is represented by the Aleph tool. For a detailed description of Aleph, refer to the Aleph manual [2]. The general algorithm followed by Aleph has four main steps [2]:

1. Select a positive example to be generalized.
2. Construct the "bottom clause", which is the most specific clause that covers the selected example and complies with the specifications (mode and determination specifications, described below).
3. Search for a clause that is more general than the bottom clause.
4. Select and keep the "best" clause and remove examples that are covered. If there are still uncovered examples, return to step 1.

Aleph specifies mode-related information that is an integral part of the background knowledge, in the form [2]:

```
:- mode(RecallNumber,PredicateMode).
```

RecallNumber bounds the non-determinacy of the predicate. It can be: (a) a number specifying the number of successful calls to the predicate; or (b) '*'specifying that the predicate has bounded non-determinacy [2].

PredicateMode specifies a legal form for calling a predicate:

```
p(ModeType, ModeType)
```

ModeType can be: +T, meaning that the parameter is an input, -T, referring to an output, or #T, if it is a constant. T represents the type of the parameter.

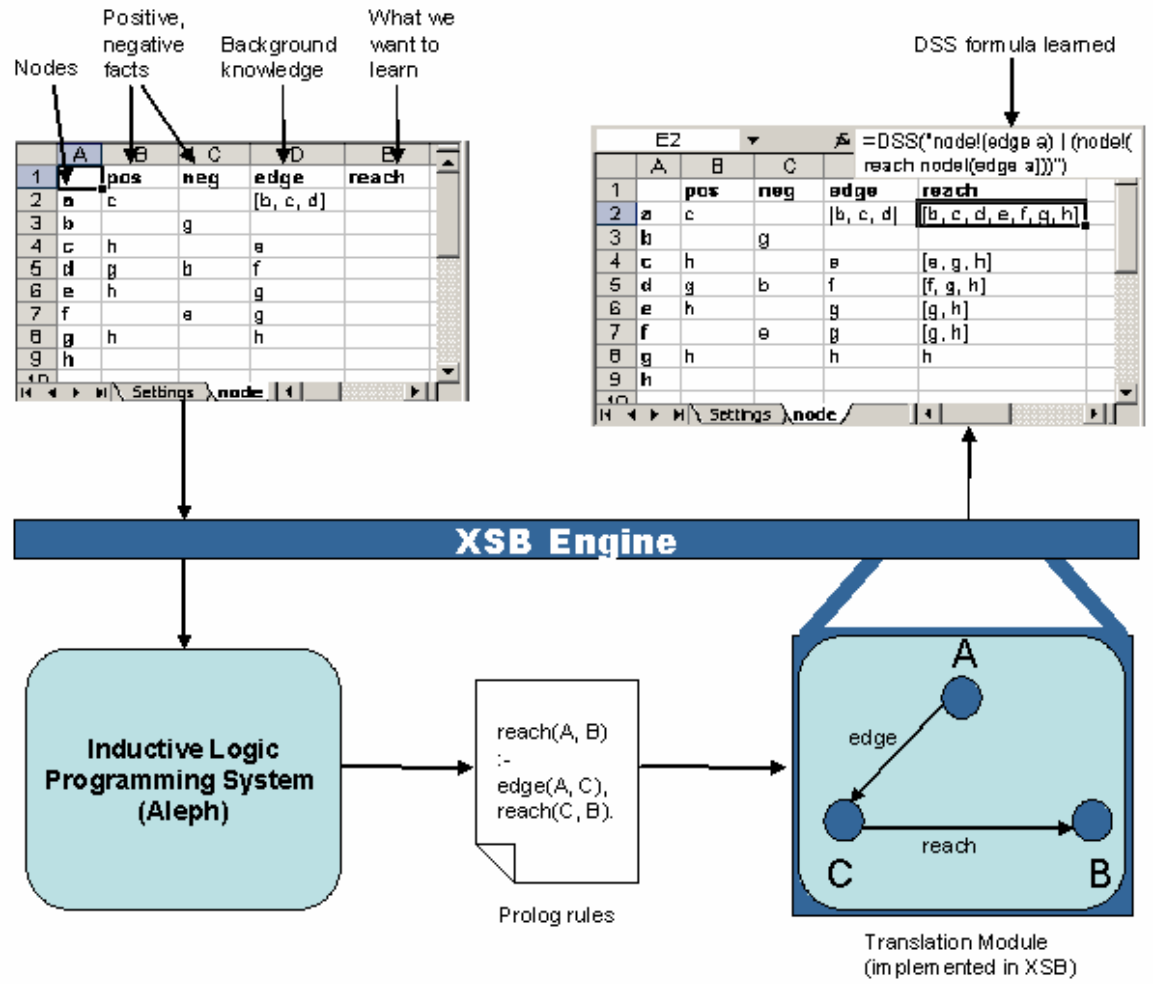


Figure 3.1.1 System architecture

In particular, 'modeh' refers to the head predicate, and 'modeb' refers to body predicates, that appear in the hypothesis. Below are two example of mode declaration for transitive closure, in which 'p' (from 'PredicateMode' described above) denotes a column name, to comply with the spreadsheet representation:

```
:- modeb(*, edge(+node,-node)).
:- modeh (*, reach(+node,-node)).
```

The 'modeh' declaration specifies that the head of the hypothesis that we want to learn is 'reach', which has two parameters, the first one, marked with '+' being an input and the second one, marked with '-', being the output. The 'modeb' declaration specifies that the predicate 'edge' may appear in the body of the learned predicate. Determination statements declare the predicates that will be used to build the hypothesis. They have the following general form:

```
:- determination(TargetName/Arity,BackgroundName/Arity).
```

The TargetName refers to the head predicate, and BackgroundName refers to a predicate that may appear in the body. We must write a determination statement for each predicate that we consider possible to appear in the body of the hypothesis. Moreover, a single target predicate can be learned at a time. Below is an example of determination statement for transitive closure:

```
:- determination(reach/2, edge/2).
```

In this example, we specify that the predicate to be leaned is 'reach', with arity 2, and the predicate 'edge', with arity 2 might appear in the body.

Aleph uses other 'setting' information besides 'mode' and 'determination' as well. One example is setting the 'clauselength', which means imposing an upper bound on the number of literals in an acceptable clause, with the following syntax:

```
:- set(clauselength, V).
```

By default, V is 4. In our applications, we set the clauselength at a higher value, usually 20. Another example is setting the upper bound on the layer of new variables:

```
:- set(i, V).
```

V here defaults to 2. Other settings used by Aleph are detailed in [2]. Please see Chapter 5 for a complete example of specifying the input to Aleph, with positive and negative examples, and background knowledge. The ILP system uses the input information to generate the corresponding Prolog rules.

3.2 Translation Component

The Translation component is implemented integrally in XSB and converts learnt rules into DSS expressions. The input to the Translation module is represented by the Prolog rules provided by the ILP system. At the core of the translation procedure is the dependency analysis of variables in the Prolog rules, modeled with Direct Acyclic Graphs (DAGs). Conceptually, DAGs are related to the Prolog rules in the following way:

- nodes are labeled with the names of variables
- edges are labeled with sheet-column name information, represented by predicate names
- the edge directions depend on the information flow between input-output parameters

A complete description of the translation algorithm is presented in Chapter 4. The output of the Translation module is the DSS formula, automatically generated from the Prolog rule learned with Aleph.

Once the DSS formula is obtained, it is instantiated for each row (and column, in the case of matrix format, see Section 4.1) and written in the spreadsheet interface. The XSB engine evaluates the formula in the background, and outputs the resulting sets in each cell.

Chapter 4. Automatic Translation from Prolog Rules to DSS Expressions

4.1 Preliminary Considerations

Before proceeding to the translation procedure, we introduce some basic concepts that we later use in our approach. If R is a set of row names, C is a set of column names, and s is a sheet name, then s!(C R) is a valid formula whose value is the union of the sets s!(c r), for c Є C and r Є R. The s!(c r) formula gives the content of the cell found at intersection of row r and column c in sheet s. We observe that, with DSS, the cell reference is lifted to sets.

Let us consider the following Prolog predicate: s(C, R, V). The translation is based on the correspondences between the name and parameters of the Prolog predicate and the DSS formula:

1. The Prolog predicate name s represents the sheet name in DSS.
2. The first two Prolog parameters, C and R, represent the set of columns and the set of rows respectively.
3. The last Prolog parameter, V, represents the DSS cell value.

In order to fit the spreadsheet format, we will require the Prolog predicates to have the 3-parameter format described above. While this may seem like a restriction on the number of variables, it actually is not, since any of the 3 parameters may represent tuples of variables:

s((A,B), (C,D,E), (V,W))

In the above example, the column name is given by the (A,B) tuple, the row name is the (C,D,E) tuple, and the value is the (V,W) tuple. Therefore, what may seem like a restriction is simply a format grouping policy to fit the spreadsheet structure.

Back to set expressions, we have the following dependencies:

$V = f_1(C, R) = \{v \in V \mid s(C, R, V)\}$

$R = f_2(C, V) = \{r \in R \mid s(C, R, V)\}$

$C = f_3(R, V) = \{c \in C \mid s(C, R, V)\}$

which say that for a Prolog predicate s(C, R, V):

- the set of V values is a function of the sets C and R
- the set of R values is a function of the sets C and V
- the set of C values is a function of the sets R and V

Our translation method computes the DSS formula incrementally, by looking at each predicate called in the Prolog body of the rule to translate.

Let us consider that after a particular step, the variables C, R and V have the DSS expressions E_C, E_R, and E_V. At this point, the s(C, R, V) predicate is called in the rule body. The E_C, E_R, and E_V formulas will be updated with the following fundamental translation steps:

$E_{V'} = E_V$ & s!(E_C E_R)
$E_{R'} = E_R$ & s!(~E_C E_V)
$E_{C'} = E_C$ & s!(E_V ~E_R)

The first formula returns the set of values in sheet s, at columns C and rows R. The second and third formulas illustrate the reverse lookup operations for computing the set of rows and columns.

DSS allows the definition of two types of sheets:

- vector format, and
- matrix format

With the vector format, we define a relation in a single column: all cells of the column refer to the same relation with respect to the rows. With the matrix representation, the whole sheet refers to a single relation: we define a relation between multiple columns and rows, all belonging to the same sheet. For the vector format, the column name is bound to a single value (we do not have a set of values).

4.2 Translation Algorithm with Direct Acyclic Graphs

We explain the translation procedure starting from a general Prolog predicate. We illustrate the translation steps on concrete examples.

4.2.1 A General Prolog Predicate

Let us assume that we are working with vector representation, and want to translate the following Prolog predicate:

s(p, X, Y) :- s(q_1, X, W), s(q_2, X, U), s(q_3, U, W), s(q_4, W, Y)

For better understanding, let us ignore the sheet name information. Since the column name is unique for each predicate in the body of the rule, we will write the rule in the following, simplified format:

p(X, Y) :- q_1(X, W), q_2(X, U), q_3(U, W), q_4(W, Y)

The information flows from X to Y: X is an input parameter, corresponding to the row names, and Y is an output parameter, corresponding to the cell values at column p. Based on this, we will be able to direct the links in the DAG (Figure 4.2.1) away from X in the two predicates for columns q_1 and q_2, and the link corresponding to q_4 towards Y. We do not have sufficient information to decide the orientation of the link WU, therefore we will consider it to be inout (bidirectional, any direction is possible). The information that we use to decide the orientation of arcs does not come from 'mode' declarations. It is determined by whether a variable is bound or not up to the moment it is processed in the translation process. For the variables that appear in the head of the predicate, we make the following assumptions: X is assumed to be bound in the beginning, since it represents the rows in the spreadsheet. Y is assumed to be not bound, because it represents the value we want to compute. For the variables that appear in the body, any of the two variables may be updated during translation, and we make no assumptions regarding whether they are bound or not.

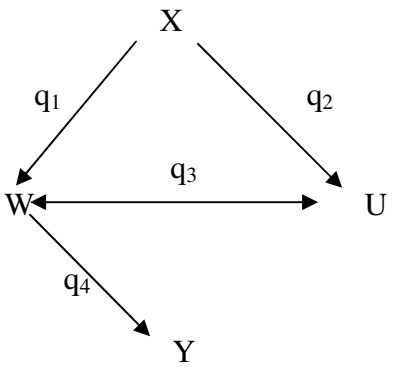


Figure 4.2.1 Modeling the information flow with DAG

The X value is given by the row names, and does not change during translation. The translation procedure looks successively at each predicate in the body, and computes partial DSS expressions for the variables that are encountered. For the predicate body below, we marked the 4 steps that will be carried out during translation.

| q_1(X, W) | q_2(X, U) | q_3(U, W) | q_4(W, Y) |
0 1 2 3 4

Step 1:

At step 1, we determine the value of W (W_1 holds the expression of W after step 1), as depending on X. We used 'f' to indicate that W is a 'function' of X (see Section 4.1). The equivalent DSS expression after this unit step is (q_1 X):

$W = W_1 = f_1(q_1, X) = (q_1\ X)$,

where q_1 represents the column name (identical to the Prolog predicate name), and X (written in upper case letter for generality) is the set of rows.

Step 2:

At step 2, we need to process the variables X and U. As before, X remains unchanged, while the value of U is computed as a function of X:

$U = U_2 = f_1(q_2, X) = (q_2\ X)$.

Step 3:

At step 3, we have a double dependency between W and U. The partial DSS expressions incrementally add to the expressions computed before:

$W = W_3 = W_1 \,\&\, f_1(q_3, U_2) = W_1 \,\&\, (q_3\ U_2) = (q_1\ X) \,\&\, (q_3\ (q_2\ X))$
$U = U_3 = U_2 \,\&\, f_2(q_3, W_1) = U_2 \,\&\, (\sim q_3\ W_1) = (q_2\ X) \,\&\, (\sim q_3\ (q_1\ X))$

Step 4:

At step 4, we determine Y, as a function of W. There is no partial DSS expression for Y computed before step 4, which means that the expression for W will remain unchanged:

$W_4 = W_3$
$Y = f_4(W_3) = (q_4\ (q_3\ (q_2\ X)) \,\&\, (q_1\ X))$

In the final DSS expression, we also make sure that none of the variables gives an empty set, and we return Y:

$IF((X \neq \Phi \,\&\, W \neq \Phi \,\&\, U \neq \Phi),\ Y)$

In DSS, we write the equivalent form:

$IF(X \,\&\, (q_1\ X) \,\&\, (q_3\ (q_2\ X)) \,\&\, (q_2\ X) \,\&\, (\sim q_3\ (q_1\ X)), (q_4\ (q_3\ (q_2\ X)) \,\&\, (q_1\ X)))$

We observe that in this particular case, the emptiness test with IF is not necessary, since the same sets from the IF condition appear as inner expressions in the formula for Y. If they were empty, we would have an empty resulting set for Y, which would have been a correct result. We will see in Chapter 5 that there are particular cases were the IF test is required in order to get a correct result. Pruning the final formula of the IF conditions, we obtain a more clear form:

$Y = (q_4\ (q_3\ (q_2\ X)) \,\&\, (q_1\ X)))$

From the example described before, we can see that the general way of generating the DSS expression at each step is:

$| q_i\ (U, V) |$
$i\text{-}1 \qquad\quad i$

$V_i = V_{i-1} \,\&\, f_1(q_i, U_{i-1})$
$U_i = U_{i-1} \,\&\, f_2(q_i, V_{i-1})$

The translation procedure handles tupes in a limited extent. The limitation appears when some variables are present in the rule body independently of each other in some predicates, and after that, they appear together, as tuples. For example:

$p(X,Y)$:- $q_0(X, (U, V))$, $q_1(U, A)$, $q_2(V, B)$, $q_3(Y, (A, B))$.

In the predicate above, determining A and B initially from q_1 and q_2 and then forming a tuple when processing q_3 would result in an incorrect set for Y. However, we can handle correctly cases where the tuple appears before each of the separate variables it consists of.

4.2.2 Transitive Closure

Next, we describe the translation procedure for the transitive closure example. The Prolog rules are:

reach(X, V) :- edge(X, V).
reach(X, V) :- edge(X, Z), reach(Z, V).

We do a DAG analysis for each rule (Figure 4.2.2):

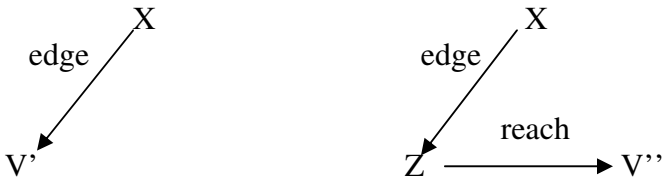


Figure 4.2.2 DAG representation for transitive closure

For the first rule, we can compute the result directly:

$V' = f_1(edge, X) = (edge\ X)$

For the second rule, we will determine the expression for Z, since it appears in the first predicate in the body, and then the expression for Y in the next step:

$Z = f_1(X) = (edge\ X)$

$V'' = f_2(Z) = f_2(f_1(X)) = (reach\ (edge\ X))$

We get the final result, which is a disjunction of the expressions for each rule. The resulting set of nodes given by the 'reach' relation is a union of the sets generated by each rule:

$IF(X \neq \Phi, V') \mid IF((X \neq \Phi \ \&\ Z \neq \Phi), V'')$

which means:

IF(X, (edge X)) | IF((X & (edge X)), (reach (edge X)))

We observe that testing emptiness for X and Z in the IF conditions is not necessary for this particular case, since the IF conditions only test the sets X and (edge X), which also appear in the output expression ('(edge X) | reach (edge X))'), which means that their emptiness will be tested implicitly in the output expression. Pruning the final formula of the IF conditions, we obtain a clearer form:

V = V' | V'' = (edge X) | (reach (edge X))

4.3 Translation Stages

The first part of this chapter presented the translation procedure from a conceptual point of view, and showed how the translation process can be modeled with DAGs. The following sections take us to lower-level details, by describing the implementation stages carried out during translation.

The translation procedure consists of two main parts which will be further detailed:

1. Reading the rules from the input file and generating the equivalent intermediate rules.
2. Transforming the intermediate rules into equivalent DSS expression.

Compared to the Prolog input rules, the intermediate expressions are closer to the spreadsheet representation and contain sheet/row/column information.

For example, let us look again at the Prolog predicate, 'reach':

```
reach(A, B) :-   edge(A, B).
reach(A, B) :-   edge(A, C), reach(C, B).
```

The intermediate representation has the form:

sheet(Sheet, Column, Row, Value)

For a predicate like: reach(A,B), that would be computed in sheet 'node', the intermediate representation is:

sheet(node, reach, A, B)

The intermediate representations for each rule are:

```
sheet(node,reach,A,B) = sheet(node,edge,A,B)
sheet(node,reach,A,B) =  sheet(node,edge,A,C) ,  sheet(node,reach,C,B)
```

where we used the '=' sign to show the correspondence between the rule head and the rule body.

With our format restrictions, a general Prolog predicate has the parameters grouped to result into a three-parameter predicate, as described in Section 4.1. In such a case, the use of intermediate expressions is not necessary, since we have all the sheet/column/row information clearly defined. The need of intermediary expressions appears in two cases of format extension.

First, when using vector representation, the column name is fixed. With the general 3-parameter predicates, we would add the column on the first position (e.g. "s('reach', A, B)") and learn the 'reach' predicate in this form:

s('reach', A, B) :- s('edge', A, B).
s('reach', A, B) :- s('edge', A, C), s('reach', C, B).

where s is the sheet name. It might be cumbersome for the user to give the specifications in the above format. Therefore, we also allow the option of incorporation both sheet name and column name in the predicate name, separated by '__', as below:

s__reach(A, B) :- s__edge(A, B).
s__reach(A, B) :- s__edge(A, C), s__reach(C, B).

With this format, the transformation to intermediate expressions appears as a necessary preprocessing step of extracting sheet/column/row/value parameters.

Second, Prolog also allows one-parameter predicates. For example:

eastbound(A) :- closed(B), has_car(A,B).

To be able to fit the 'eastbound' predicate in the spreadsheet format, we will add the 'value' parameter 1 to represent 'true'. The value 'false' will be empty cell, and there is no need to specify it explicitly. The intermediary expression for 'eastbound', considering that the sheet name is 's', becomes:

sheet(s, eastbound, A, 1) = sheet(s, closed, B, 1), sheet(s, has_car, A, B).

The rule generation is based on the algorithm presented in the previous section, using the DAG model. The final expression is constructed iteratively, adding at each step the portion of the expression that handles the variables from the current term.

Chapter 5. Examples of Applications

The current chapter presents a few examples of using Deductive Spreadsheets for learning.

5.1 Transitive Closure

In the transitive closure example, we have several nodes in a graph, given as row names. We know what pairs of nodes are directly connected, and we determine the set of nodes that are accessible from each node, in any number of hops.

Figure 5.1.1 and Figure 5.1.2 show the input information that is provided by the user and given to Aleph as input. The 'Settings' sheet contains mode relations, where we specify the parameter types (input, output, constant with the symbols '+', '-', '#' respectively), as well as dependency information that relates the predicate to be learned to the predicates that might appear in the body.

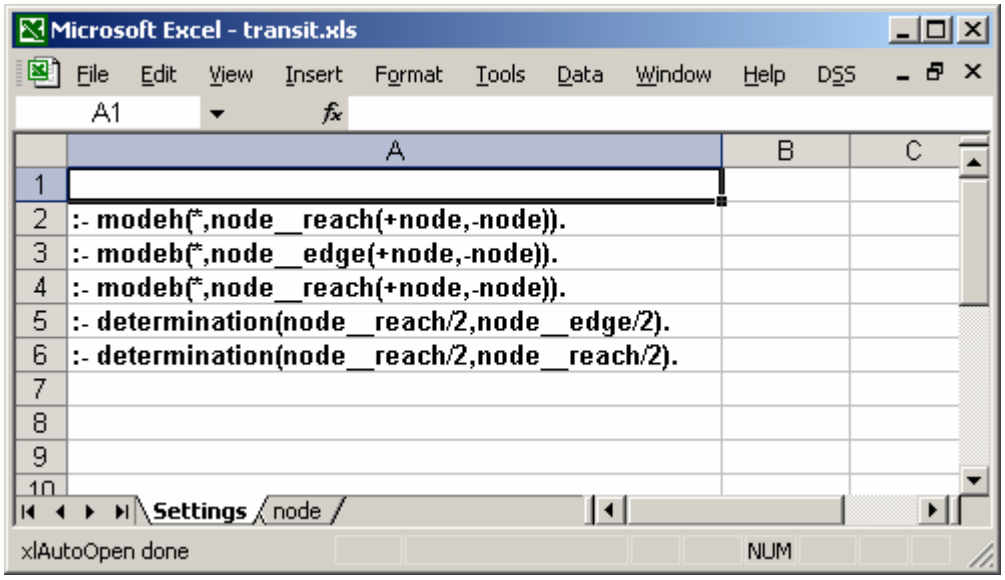
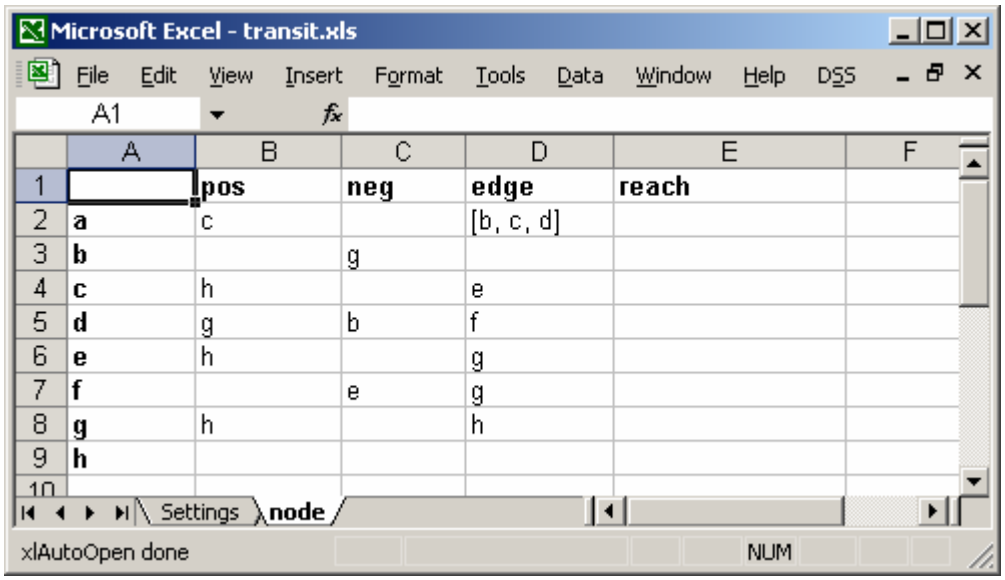


Figure 5.1.1 Transitive closure: Mode and determination settings used by Aleph

The ILP algorithm also requires positive and negative facts. In our example, they are given in the column 'pos' and 'neg', in the sheet 'node' (Figure 5.1.2). The rows in this sheet are the nodes in the graph. The 'pos' column gives a few examples of nodes that can be reached from the node written as row (e.g. the cell al B2 in Excel, specifies that node 'c' can be reached from node 'a'). The 'neg' column gives a few examples of nodes that can not be reached from the node on the corresponding row (e.g. in the Excel cell C3, we specify that node 'g' can not be reached from 'b').

We also provide background information that will appear as facts in the input to the ILP algorithm. Here we specified the nodes between which there is a direct link, in column 'edge'. For example, the Excel cell D2 specifies that there is a direct link between node pairs (a, b), (a, c), and (a, d). Column 'reach' is left empty, and will be automatically filled in during the learning process.



	A	B	C	D	E	F
1		**pos**	**neg**	**edge**	**reach**	
2	**a**	c		[b, c, d]		
3	**b**		g			
4	**c**	h		e		
5	**d**	g	b	f		
6	**e**	h		g		
7	**f**		e	g		
8	**g**	h		h		
9	**h**					
10						

Figure 5.1.2 Transitive closure: Positive, negative examples and background knowledge (columns pos, neg, edge)

Once the input information has been specified in the interface, we can initiate the learning process by clicking on the 'Learn DSS expression' menu command (Figure 5.1.3). The following steps are carried out in the background:

1. Reading the input information and constructing the files used by Aleph: a '.b' file with background knowledge, including the mode specifications, an '.f' file with positive examples and an '.n' file with negative examples.

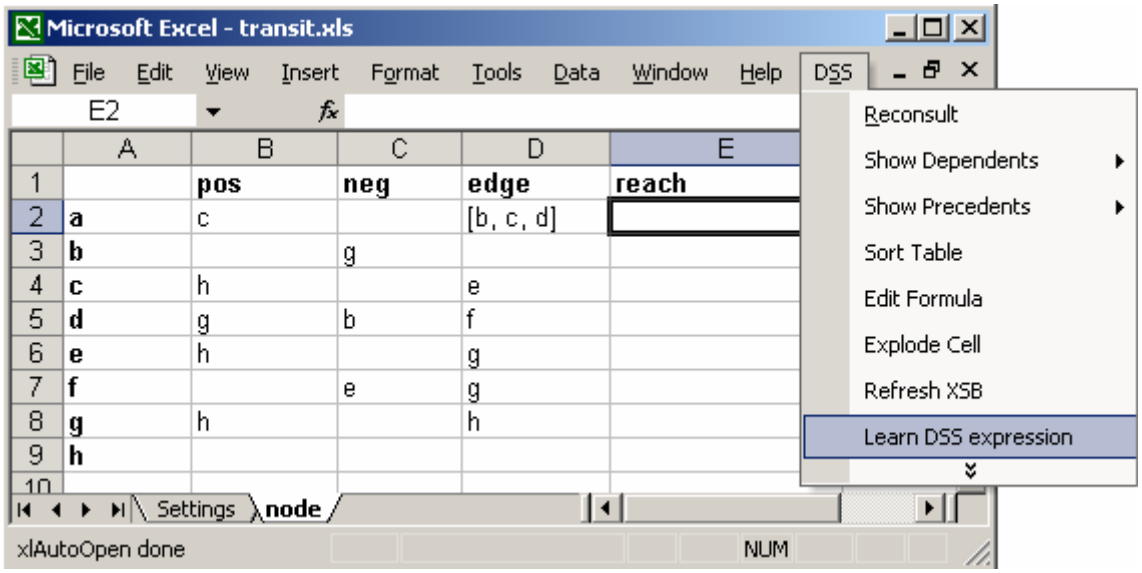
The content of the files is given below, for the 'reach' example:

Background knowledge file:

```
% Mode declarations
:- modeh(*,node__reach(+node,-node)).
:- modeb(*,node__edge(+node,-node)).
:- modeb(*,node__reach(+node,-node)).
:- determination(node__reach/2,node__edge/2).
:- determination(node__reach/2,node__reach/2).

% Background knowledge
node__edge(a,b).
node__edge(a,d).
node__edge(a,c).
```

```
node__edge(c,e).
node__edge(d,f).
node__edge(e,g).
node__edge(f,g).
node__edge(g,h).
```



	A	B	C	D	E
1		**pos**	**neg**	**edge**	**reach**
2	**a**	c		[b, c, d]	
3	**b**		g		
4	**c**	h		e	
5	**d**	g	b	f	
6	**e**	h		g	
7	**f**		e	g	
8	**g**	h		h	
9	**h**				
10					

Figure 5.1.3 Transitive closure: Initiating the learning process

Positive facts file:

```
node__reach(a,c).
node__reach(c,h).
node__reach(d,g).
node__reach(e,h).
node__reach(g,h).
```

Negative facts file:

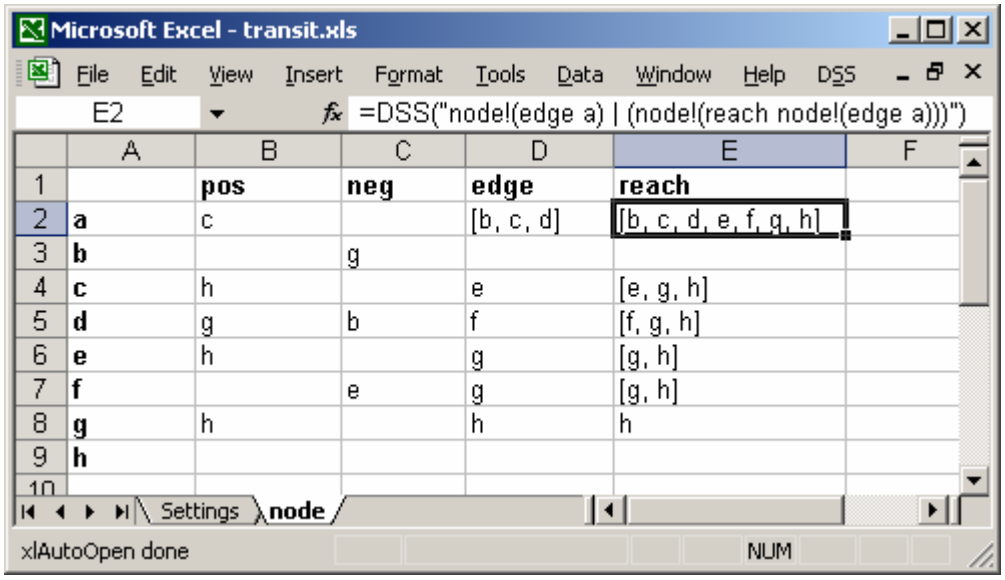
```
node__reach(b,g).
node__reach(d,b).
node__reach(f,e).
```

2. Calling and running Aleph with the three files described before as input. After this step, a text file is generating, with the Prolog rules that were learned by the ILP strategy.

Below is a sample script that can be used to call Aleph, read the three files with background knowledge, positive and negative examples, learn the Prolog rules and write the results in a text file.

```
cd C:\Demo
"C:\Program Files\pl\bin\plwin" -f none -g
"load_files(['input'],[silent(true)])" -t go -- $*
```

```
go:-
      [aleph],
      read_all(learn),
      induce,
      write_rules('pl_rules.txt').
```



	A	B	C	D	E	F
1		**pos**	**neg**	**edge**	**reach**	
2	**a**	c		[b, c, d]	[b, c, d, e, f, g, h]	
3	**b**		g			
4	**c**	h		e	[e, g, h]	
5	**d**	g	b	f	[f, g, h]	
6	**e**	h		g	[g, h]	
7	**f**		e	g	[g, h]	
8	**g**	h		h	h	
9	**h**					
10						

Figure 5.1.4 Transitive closure: The resulting DSS formula

For our example, Aleph generates the following rules:

```
node__reach(A, B) :-
   node__edge(A, B).

node__reach(A, B) :-
   node__edge(A, C), node__reach(C, B).
```

3. Next, the Translation Module is called to generate the DSS formula.
4. Finally, the DSS formula is written at each row in the column 'reach'. The results are shown in Figure 5.1.4.

In this example, the final DSS expression for node 'a' is:

node!(edge a) | (node!(reach node!(edge a)))

The resulting set of nodes is a union (specified by 'l') of the sets generated by each rule in the Prolog predicate.

5.2 Program Analysis

Broadly speaking, the objective of program analysis is to automatically study the behavior of computer programs. In this example, we focus on the reaching problem: for each variable defined in the structure of a program, determine the blocks of code where it can be accessed. We solve the problem in two parts, because we are able to learn only one predicate at a time, with our current DSS framework. The 'in' and 'out' predicates are mutually recursive, therefore we must first decide an order of learning. The predicate that is learned first will start from some initial background knowledge regarding the other predicate. In this example, we will learn 'in' first, based on some background facts that characterize 'out'. Next, we will learn 'out', using the definition of 'in' that we learned at the first step. Therefore, the two steps that we use to learn the in-out mutually recursive predicates are:

1. Determine in what blocks of code a variable can be accessed at the *beginning* of the block ('in_reach' predicate).
2. Determine in what blocks of code a variable can be accessed at the *end* of the block ('out_reach' predicate).

Due to the limitations of our current framework, we will have to learn each of the two predicates, separately, in distinct workbooks. Future extensions will allow learning of multiple predicates in the same book.

It might happen that a variable is accessed only at the beginning of a block and later killed in that block, or only at the end of a block if it is generated in that block, or both at the beginning and the end if it was generated in a previous block and is not killed in the current block.

Figure 5.2.1 shows the example that we use in our learning process. It illustrates the blocks and the precedence relations, as well as the variables that are generated in each block.

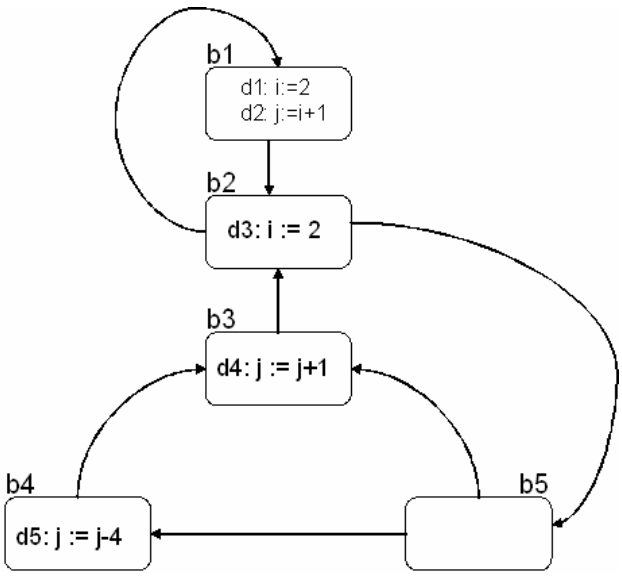


Figure 5.2.1 A program analysis example

The ‘predecesor’ relations between blocks are:
pred(b1, b2). pred(b2, b1). pred(b2, b3).
pred(b3, b4). pred(b3, b5). pred(b4, b5).

The ‘generate’ variable background knowledge is:
gen(b1, d1). gen(b1, d2). gen(b2, d3). gen(b3, d4). gen(b4, d5).

The ‘kill’ relations are:
kill(b1, d3). kill(b1, d4). kill(b1,d5). kill(b2, d1).
kill(b3, d2). kill(b3, d5). kill(b4, d2). kill(b4, d4).

The Prolog rules that would define ‘in’ and ‘out’ sets are:

```
out(X,B) :- in(X,B), not(kill(B,X)).
out(X,B) :- gen(B,X).

in(X,B) :- pred(P,B), out(X,P).
```

where X is a variable in the program, B is a block of code and P is a preceding block.

5.2.1 The ‘in’ reaching sets

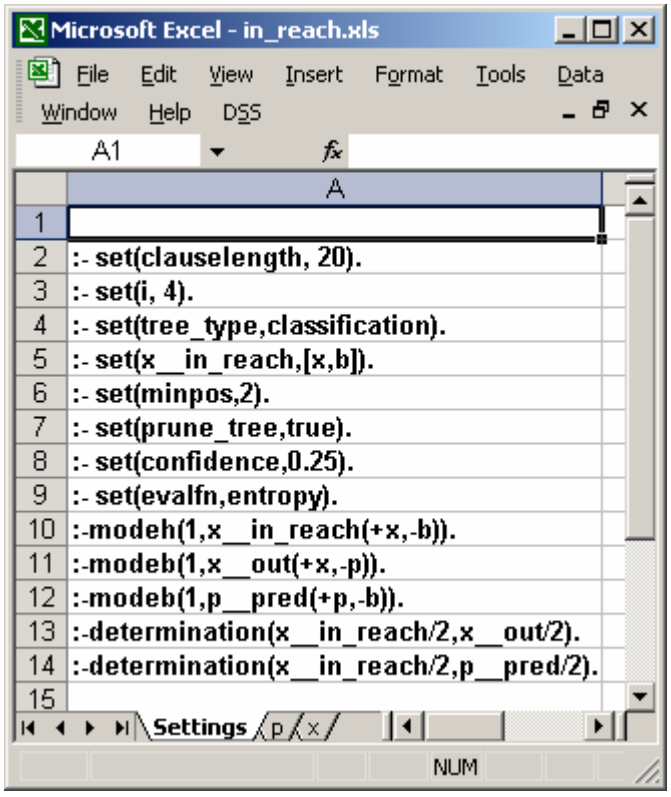
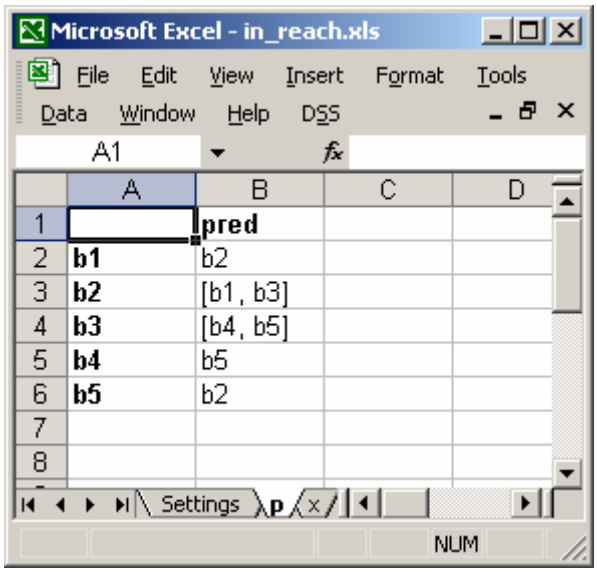
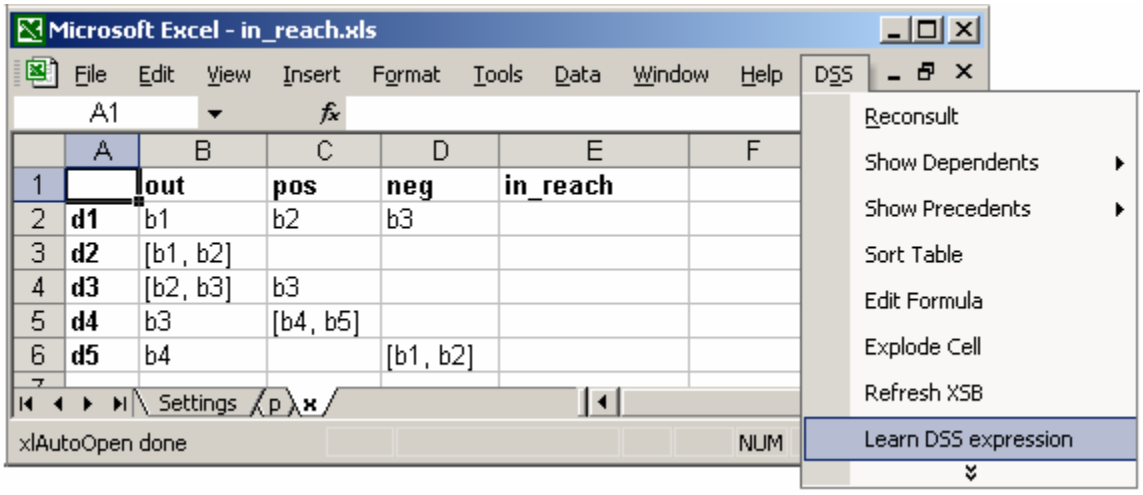


Figure 5.2.2 Program analysis: Mode and determination settings used by Aleph ('in_reach' predicate)



	A	B	C	D
1		**pred**		
2	**b1**	b2		
3	**b2**	[b1, b3]		
4	**b3**	[b4, b5]		
5	**b4**	b5		
6	**b5**	b2		
7				
8				

Figure 5.2.3 Program analysis: Some background knowledge used by Aleph ('in_reach' predicate)



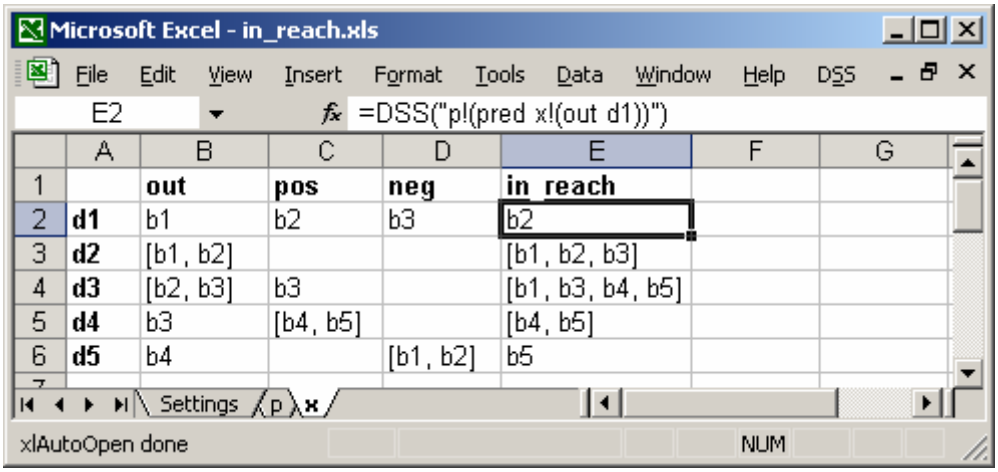
	A	B	C	D	E	F
1		**out**	**pos**	**neg**	**in_reach**	
2	**d1**	b1	b2	b3		
3	**d2**	[b1, b2]				
4	**d3**	[b2, b3]	b3			
5	**d4**	b3	[b4, b5]			
6	**d5**	b4		[b1, b2]		

Figure 5.2.4 Program analysis: Positive, negative examples and background knowledge (columns pos, neg, out) Initializing the learning process ('in_reach' predicate)

Figure 5.2.2 presents the 'Settings' sheet, which specifies the predicate to be learned, predicates that may appear in the body and variable types (input, output).

Figure 5.2.3 and Figure 5.2.4 present the information that will be given as input to the ILP system, in Aleph. We observe that the background facts appear in two sheets (p and x). In this example, there are 5 blocks of code (b1, b2... b5), and 5 variables (d1, d2 ... d5). The

'pred' predicate specifies the precedence relation between blocks. For example, the Excel cell B2 shows that block b1 precedes block b2. In the case of program loops, the relation is circular (e.g. for blocks b1 and b2). The 'out' predicate tells, for each variable, the set of block at the end of which it is accessible.



	A	B	C	D	E	F	G
1		**out**	**pos**	**neg**	**in_reach**		
2	**d1**	b1	b2	b3	b2		
3	**d2**	[b1, b2]			[b1, b2, b3]		
4	**d3**	[b2, b3]	b3		[b1, b3, b4, b5]		
5	**d4**	b3	[b4, b5]		[b4, b5]		
6	**d5**	b4		[b1, b2]	b5		

Figure 5.2.5 Program analysis: The resulting DSS formula ('in_reach' predicate)

The learning procedure follows the same steps as described in the transitive closure example. The resulting Prolog rules are:

```
x__in_reach(A, B) :-
   x__out(A, C), p__pred(C, B).
```

Next, the Prolog rules are translated to DSS formula. Figure 5.2.5 shows the result of translation. For example, for row d1, the DSS formula is: p!(pred x!(out d1)), which gives the set of blocks of code at the beginning of which variable d1 is accessible (i.e. block b2).

5.2.2 The ‘out’ reaching sets

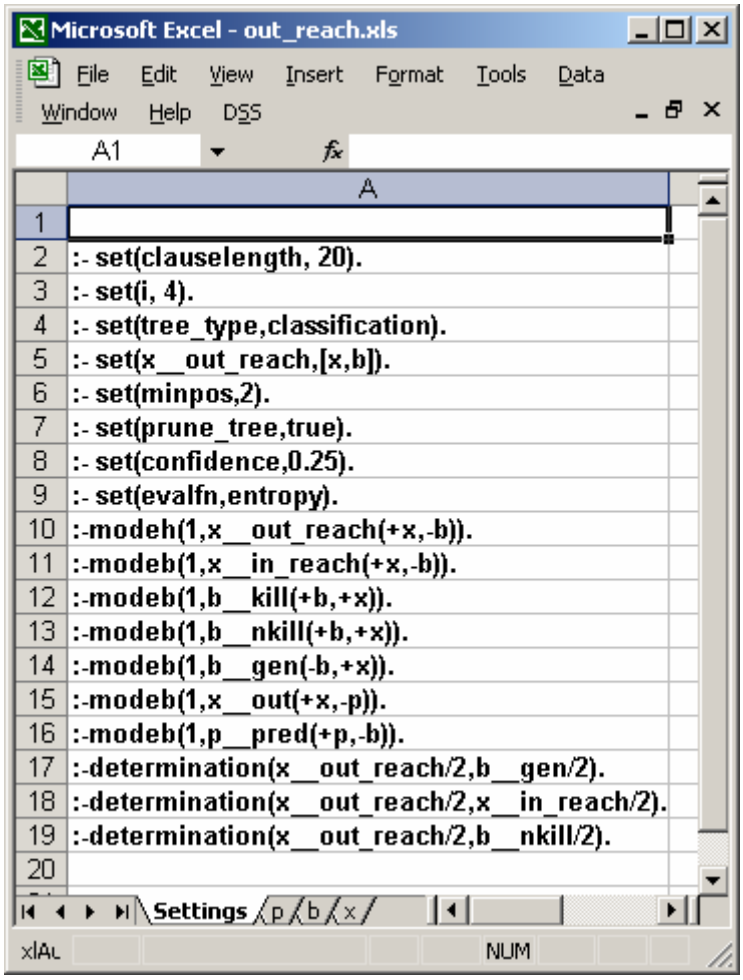
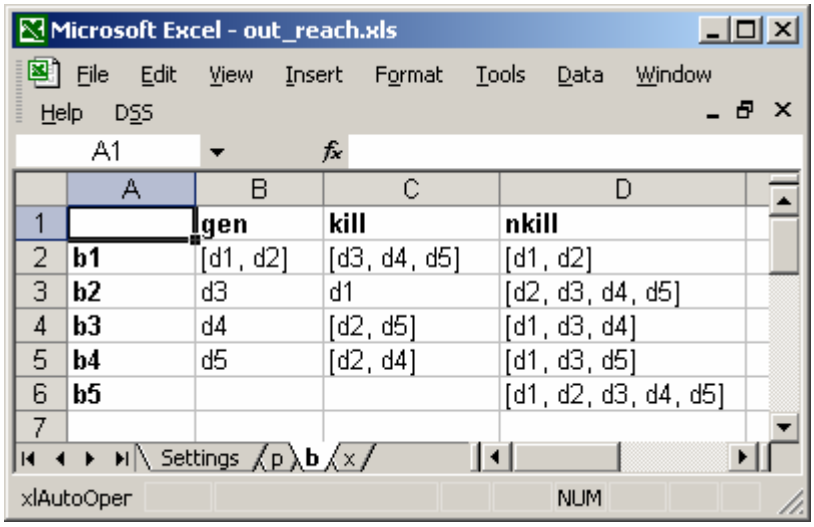


Figure 5.2.6 Program analysis: Mode and determination settings used by Aleph (‘out_reach’ predicate)

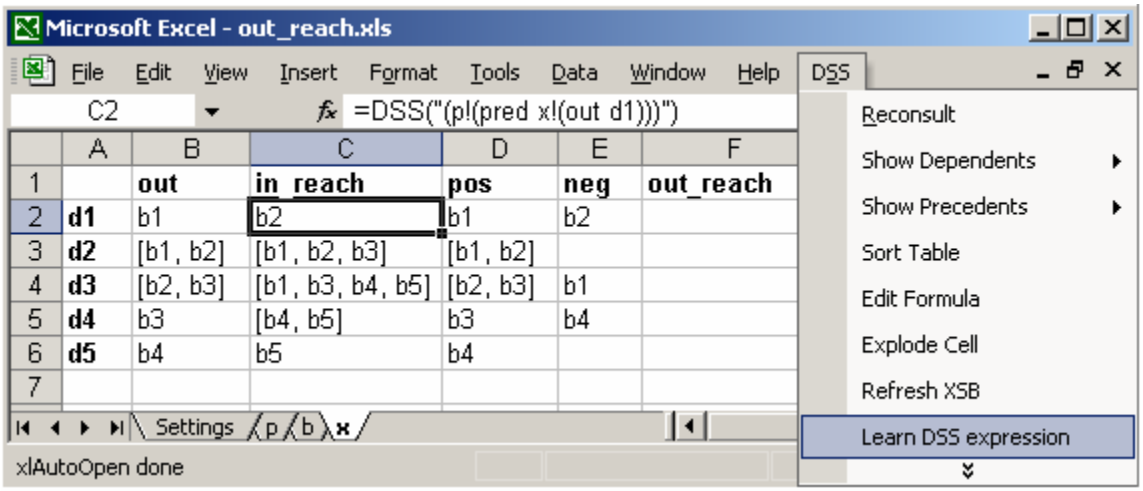
Once the ‘in_reach’ predicate is learned, we use the result to learn the second predicate, ‘out_reach’. The ‘out_reach’ predicate determines the set of code blocks at the end of which a variable is accessible.

Figure 5.2.6 illustrates the ‘Settings’ sheet for this predicate. We observe that the ‘out_reach’ predicate depends on the ‘in_reach’ predicate that we learned before. Furthermore, the ‘in_reach’ predicate depends on the ‘out’ predicate, which is actually a version of ‘out_reach’ (materialized by a few instances). Therefore, the reaching definition is recursive. In order to be able to learn both predicates, we need to start the process with some factual background knowledge for one of the predicates (i.e. ‘out’).



	A	B	C	D
1		**gen**	**kill**	**nkill**
2	**b1**	[d1, d2]	[d3, d4, d5]	[d1, d2]
3	**b2**	d3	d1	[d2, d3, d4, d5]
4	**b3**	d4	[d2, d5]	[d1, d3, d4]
5	**b4**	d5	[d2, d4]	[d1, d3, d5]
6	**b5**			[d1, d2, d3, d4, d5]
7				

Figure 5.2.7 Program analysis: Some background knowledge used by Aleph ('out_reach' predicate)



	A	B	C	D	E	F
1		**out**	**in_reach**	**pos**	**neg**	**out_reach**
2	**d1**	b1	b2	b1	b2	
3	**d2**	[b1, b2]	[b1, b2, b3]	[b1, b2]		
4	**d3**	[b2, b3]	[b1, b3, b4, b5]	[b2, b3]	b1	
5	**d4**	b3	[b4, b5]	b3	b4	
6	**d5**	b4	b5	b4		
7						

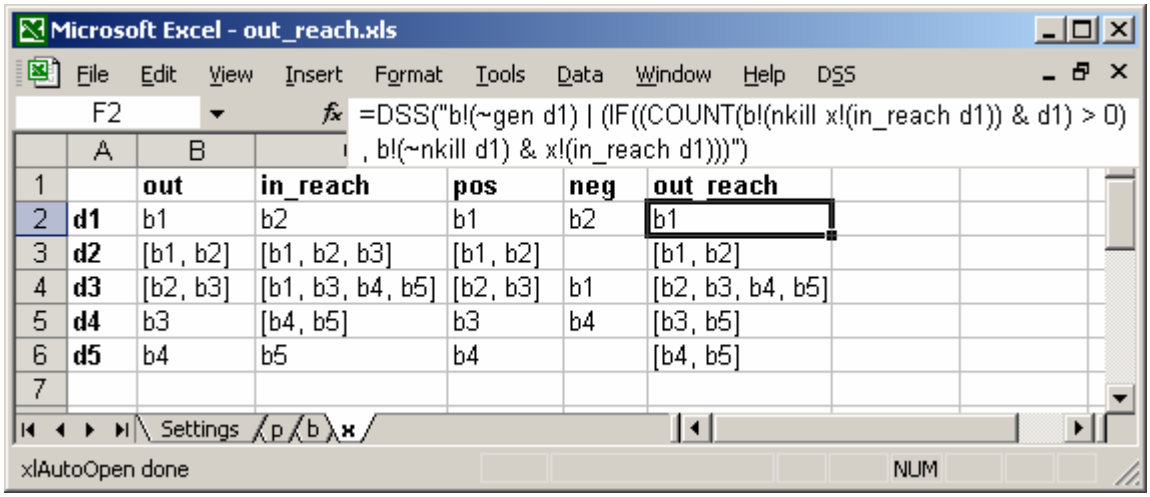
Figure 5.2.8 Program analysis: Positive, negative examples and background knowledge (columns pos, neg, out) Initializing the learning process ('out_reach' predicate)

Figure 5.2.7 and Figure 5.2.8 illustrate the background knowledge, positive and negative examples which were used in the learning process. The 'gen' column outputs the variables that were generated in each block of code, while the 'kill' column specifies the variables that were killed in each block of code. The 'nkill' column represents 'not kill'. From Figure 5.2.8, we observe that the definition learned for 'in_reach' is now used to provide background knowledge for the current predicate.

The Prolog rules that were learned by Aleph are shown below:

```
x__out_reach(A, B) :-
   b__gen(B, A).
```

```
x__out_reach(A, B) :-
   x__in_reach(A, B), b__nkill(B, A).
```



	A	B				
1		**out**	**in_reach**	**pos**	**neg**	**out_reach**
2	**d1**	b1	b2	b1	b2	b1
3	**d2**	[b1, b2]	[b1, b2, b3]	[b1, b2]		[b1, b2]
4	**d3**	[b2, b3]	[b1, b3, b4, b5]	[b2, b3]	b1	[b2, b3, b4, b5]
5	**d4**	b3	[b4, b5]	b3	b4	[b3, b5]
6	**d5**	b4	b5	b4		[b4, b5]
7						

Figure 5.2.9 Program analysis: The resulting DSS formula ('out_reach' predicate)

Figure 5.2.9 presents the result of the translation. The final DSS formula for the first row is:

```
b!(~gen d1)
|
(IF((COUNT(b!(nkill x!(in_reach d1)) & d1) > 0),
      b!(~nkill d1) & x!(in_reach d1)))
```

We see the two parts of the formula, corresponding to the two Prolog rules for this predicate. The final set is a union of the sets generated by each rule. The first set returns the blocks where a variable was generated.

The condition tested with COUNT ensures that the set that would correspond to the 'A' literal in the Prolog rule above is not empty. The condition is not necessary in this case, but it appears for generality purposes. The actual output set of the second rule is computed with the subexpression:

```
b!(~nkill d1) & x!(in_reach d1)
```

which retrieves the blocks for which the variable was in reach at the beginning of the block and was not killed in that block.

5.3 Network Flow Analysis

The network flow analysis example computes the web load in a network. The input information consists of packet descriptions, specified by the values of the following packet fields: packet number, source id, destination id, source port, destination port. This information is collected from the packet header, once a packet is sent over the network. The source and destination refer to the original source and final destination, regardless of the nodes through which the packet passes along the route. Therefore, the transmission method is point-to-point, not broadcast.

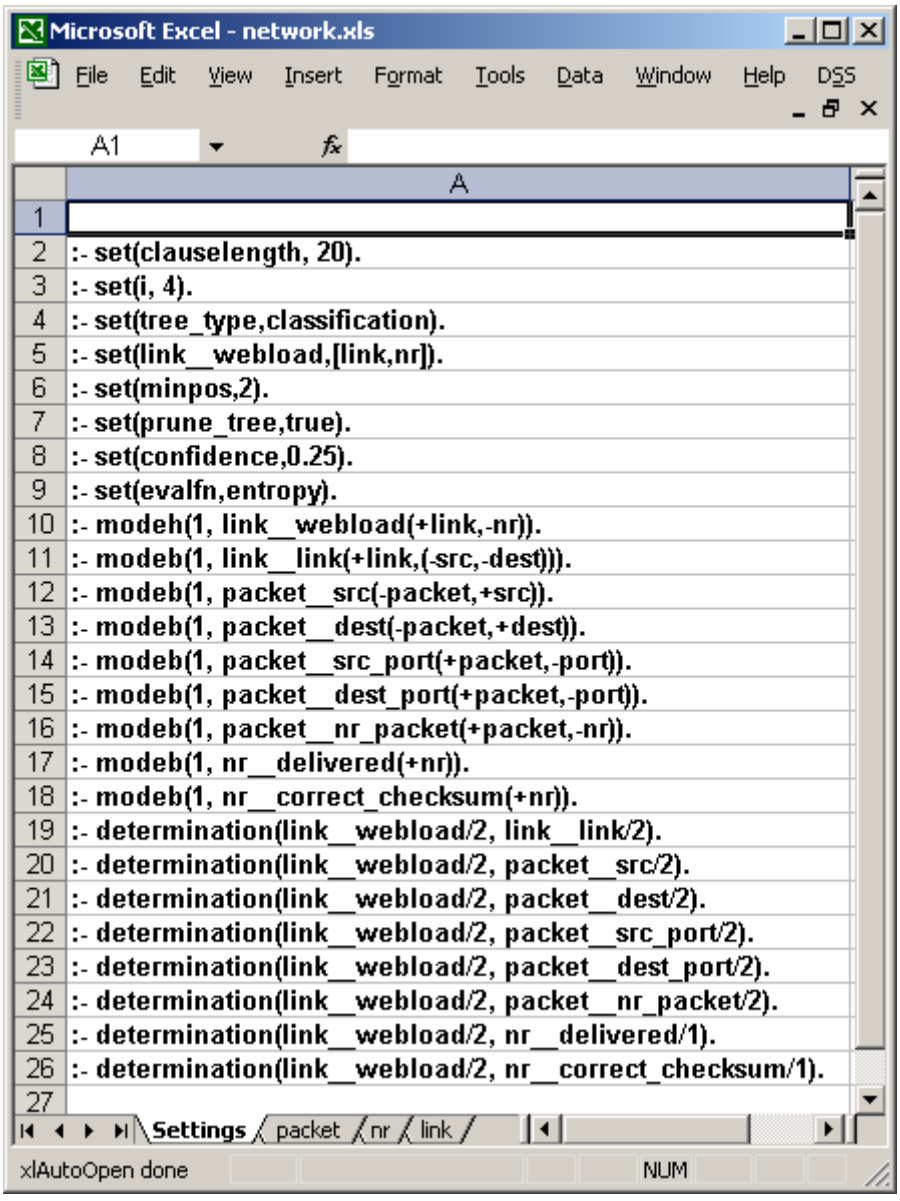
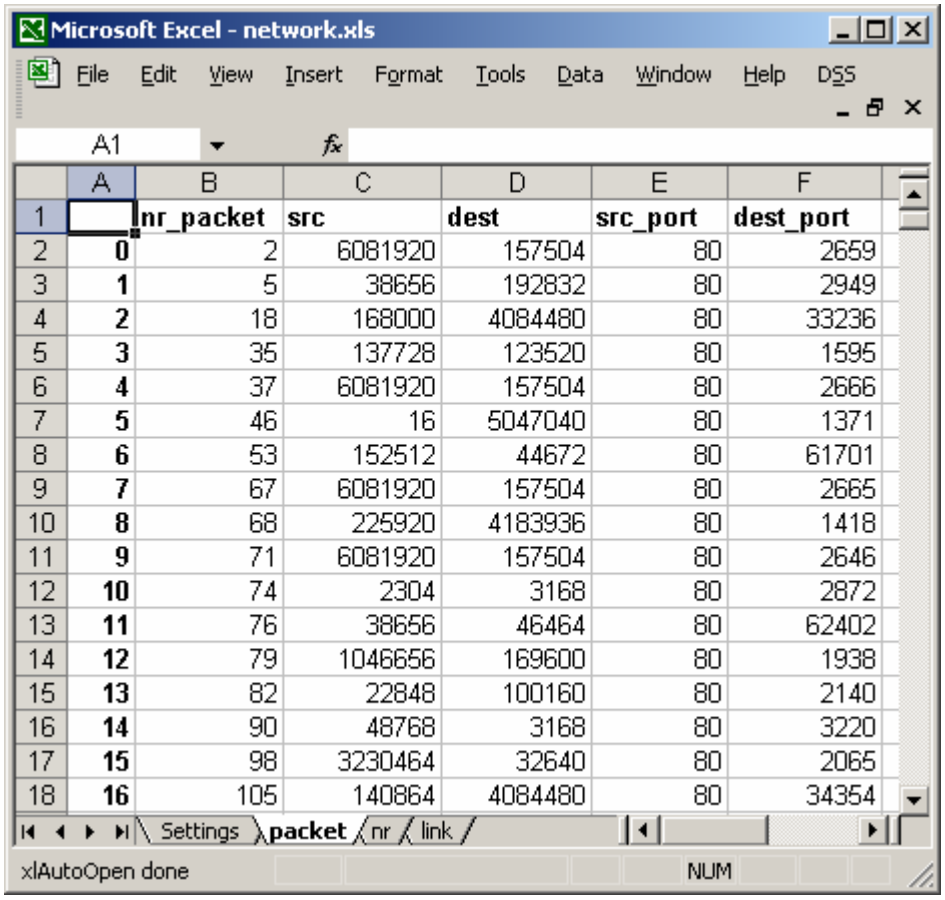


Figure 5.3.1 Network flow analysis: Mode and determination settings used by Aleph

While the previous information is recorded when packets are sent, we also collect information at the moment when packets are delivered at their destination. When a packet reaches its destination, we mark it as being delivered, and also mark if it has a correct checksum (otherwise, it has been altered along the transmission link).

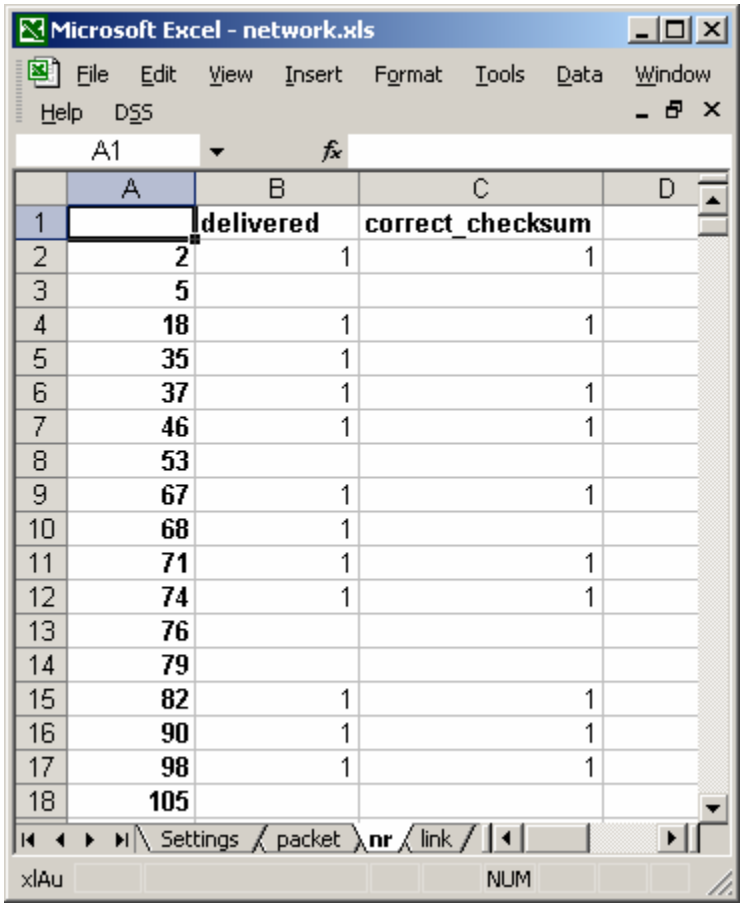
Based on the information described before (packet description, delivery status and checksum correctness), we determine the 'webload', which represents the set of packets correctly delivered along a link. Since we use a point-to-point connection, a link is specified by a start point (source id) and an end point (destination id).



	A	B	C	D	E	F
1		nr_packet	src	dest	src_port	dest_port
2	0	2	6081920	157504	80	2659
3	1	5	38656	192832	80	2949
4	2	18	168000	4084480	80	33236
5	3	35	137728	123520	80	1595
6	4	37	6081920	157504	80	2666
7	5	46	16	5047040	80	1371
8	6	53	152512	44672	80	61701
9	7	67	6081920	157504	80	2665
10	8	68	225920	4183936	80	1418
11	9	71	6081920	157504	80	2646
12	10	74	2304	3168	80	2872
13	11	76	38656	46464	80	62402
14	12	79	1046656	169600	80	1938
15	13	82	22848	100160	80	2140
16	14	90	48768	3168	80	3220
17	15	98	3230464	32640	80	2065
18	16	105	140864	4084480	80	34354

Figure 5.3.2 Network flow analysis: Some background knowledge used by Aleph

Figure 5.3.1 shows the 'Settings' used by Aleph to learn the 'webload' predicate. Figure 5.3.2 and Figure 5.3.3 illustrate some background knowledge that is also given as input to the ILP algorithm. Figure 5.3.2 illustrates the 'packet' sheet, which gives the packets descriptions. The columns specify the packet number, the source id, the destination id, the source port, and the destination port. The row value gives a temporal ordering in which the packets were sent. It is basically a counter that is incremented after each packet submission.



	A	B	C	D
1		delivered	correct_checksum	
2	2	1	1	
3	5			
4	18	1	1	
5	35	1		
6	37	1	1	
7	46	1	1	
8	53			
9	67	1	1	
10	68	1		
11	71	1	1	
12	74	1	1	
13	76			
14	79			
15	82	1	1	
16	90	1	1	
17	98	1	1	
18	105			

Figure 5.3.3 Network flow analysis: Some background knowledge used by Aleph

Figure 5.3.3 presents the 'nr' sheet, containing the information recorded when packets are delivered at destination. The columns represent the delivery status (1 if the packet was delivered, and empty cell if it was not), and checksum correctness (1 if the packet has a correct checksum, empty cell if not). The row names represent the packet number (no temporal order is employed).

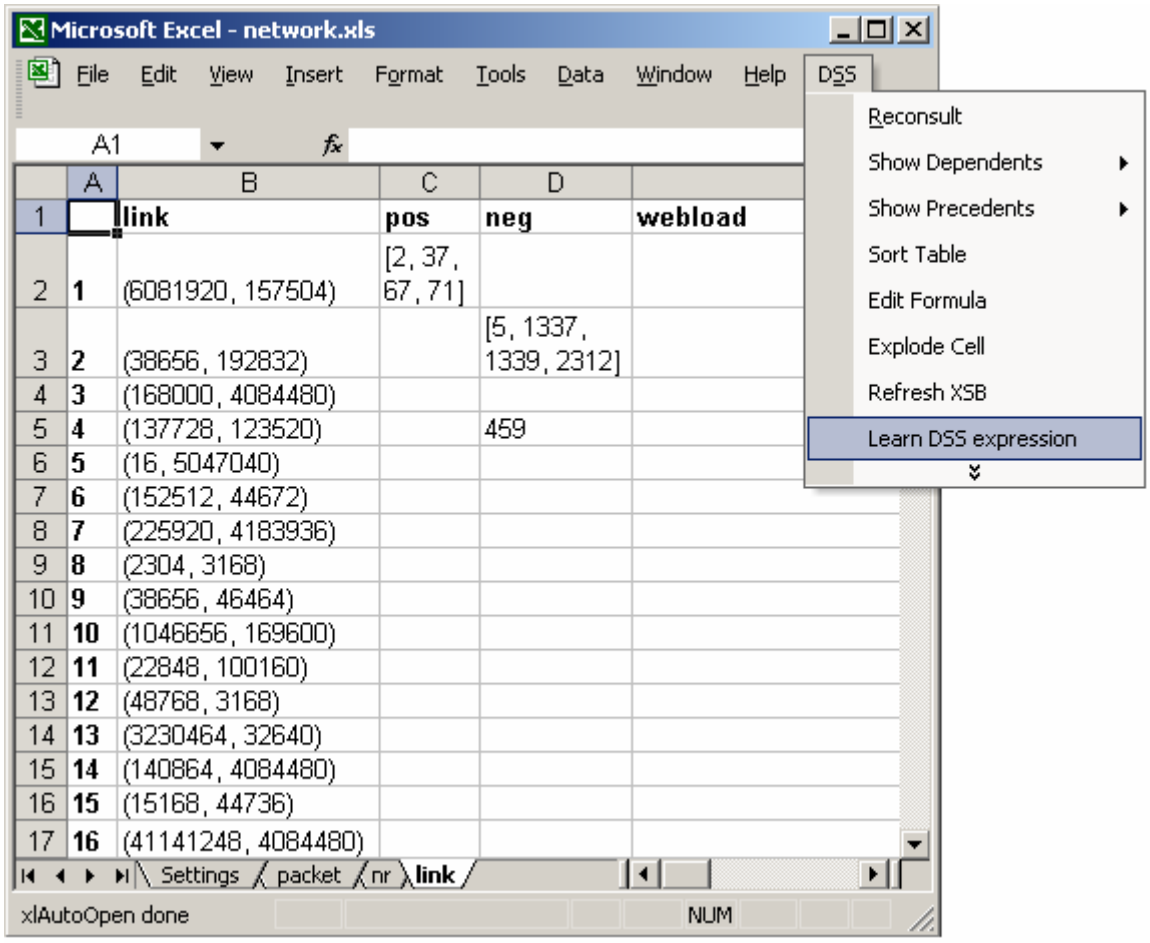


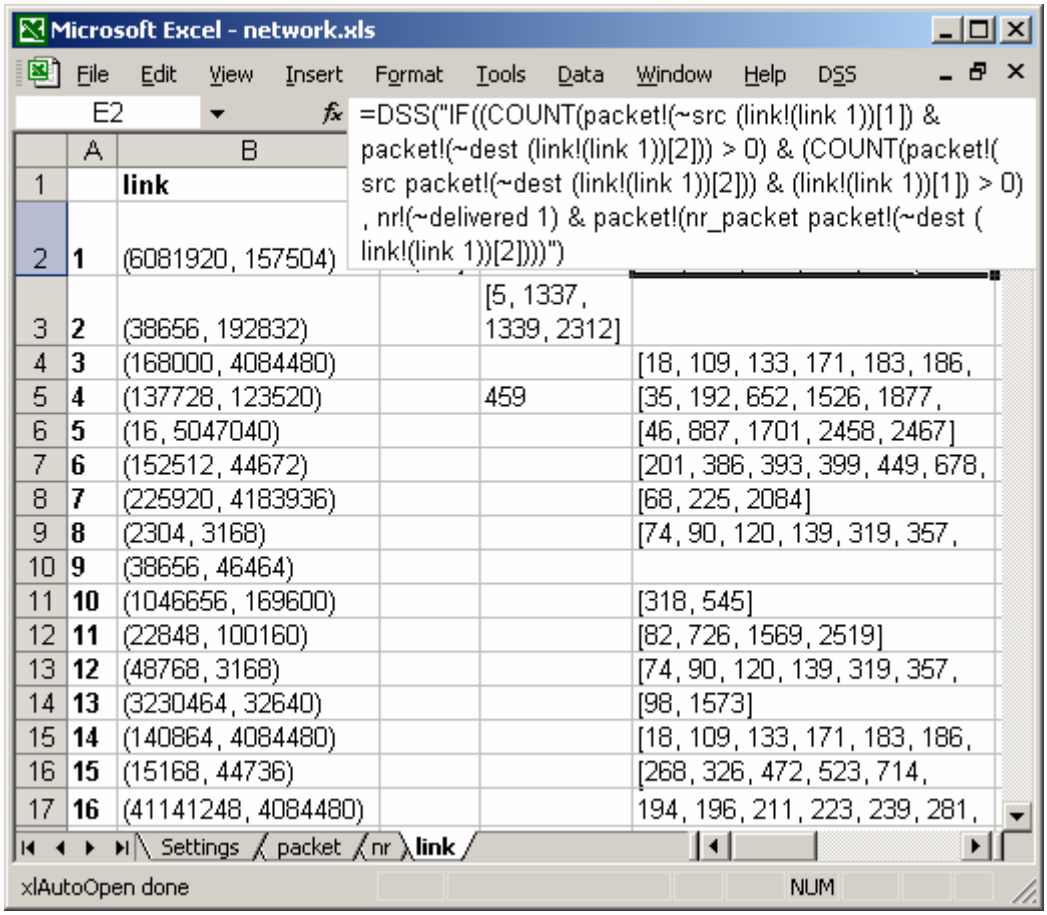
Figure 5.3.4 Network flow analysis: Positive, negative examples and background knowledge (columns pos, neg, link) Initializing the learning process

Figure 5.3.4 shows the 'link' sheet, where the row name is given by a link identification id. The column 'link' constitutes background knowledge, and relates the link id (row name) to the two extreme points of the link (source and destination). We specify a point-to-point link with a (source, destination) pair. This sheet also contains positive and negative examples of the 'webload' predicate.

Once the learning process is initiated, the ILP algorithm learns a Prolog predicate that complies with the specification, which is then translated to a DSS formula.

The Prolog rule learned with Aleph is:

```
link__webload(A, B) :-
  link__link(A,  (C,  D)),  packet__src(E,  C),  packet__nr_packet(E,  B),
  nr__delivered(B), packet__dest(E, D).
```



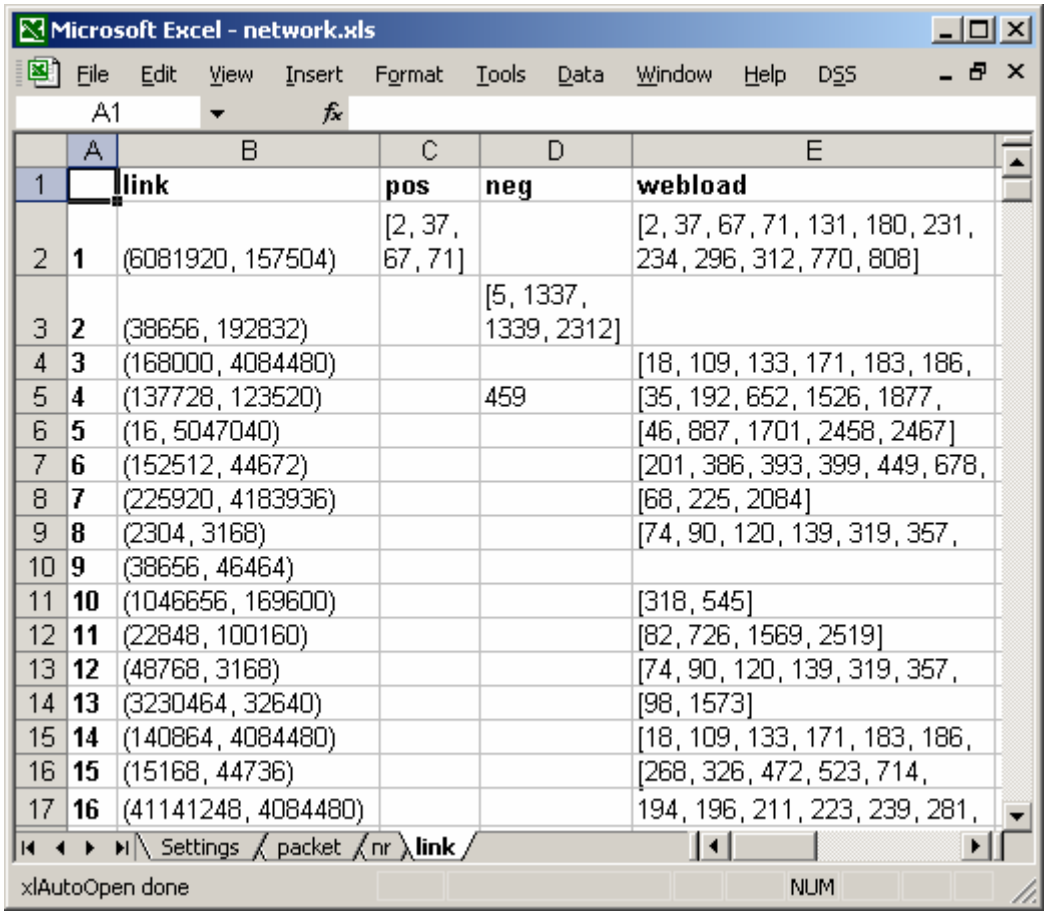
	A	B			
1		**link**			
2	**1**	(6081920, 157504)		[5, 1337, 1339, 2312]	
3	**2**	(38656, 192832)			
4	**3**	(168000, 4084480)		459	[18, 109, 133, 171, 183, 186,
5	**4**	(137728, 123520)			[35, 192, 652, 1526, 1877,
6	**5**	(16, 5047040)			[46, 887, 1701, 2458, 2467]
7	**6**	(152512, 44672)			[201, 386, 393, 399, 449, 678,
8	**7**	(225920, 4183936)			[68, 225, 2084]
9	**8**	(2304, 3168)			[74, 90, 120, 139, 319, 357,
10	**9**	(38656, 46464)			
11	**10**	(1046656, 169600)			[318, 545]
12	**11**	(22848, 100160)			[82, 726, 1569, 2519]
13	**12**	(48768, 3168)			[74, 90, 120, 139, 319, 357,
14	**13**	(3230464, 32640)			[98, 1573]
15	**14**	(140864, 4084480)			[18, 109, 133, 171, 183, 186,
16	**15**	(15168, 44736)			[268, 326, 472, 523, 714,
17	**16**	(41141248, 4084480)			194, 196, 211, 223, 239, 281,

Figure 5.3.5 Network flow analysis: The resulting DSS formula

Figure 5.3.5 shows the final DSS formula for 'webload', along with the set evaluation. For row 1, the DSS formula is:

```
IF((COUNT(packet!(~src (link!(link 1))[1]) & packet!(~dest (link!(link
1))[2])) > 0) & (COUNT(packet!(src packet!(~dest (link!(link 1))[2])) &
(link!(link 1))[1]) > 0),
nr!(~delivered 1) & packet!(nr_packet packet!(~dest (link!(link 1))[2])))
```

The complexity and length of the rule learned with ILP depends on the positive and negative examples given. As an observation, the set of packets *correctly delivered* is a subset of all the packets *delivered* at destination (with or without correct checksum). Therefore, the 'delivered' and 'correct_ckecksum' predicates will never appear together in the rule body, since Aleph prefers shorter rules, and 'correct_ckecksum' also incorporates the information from 'delivered'('correct_checksum' means delivered and having a correct checksum). In this case, we obtained the 'delivered' predicate in the body, based on the positive and negative examples given.



	A	B	C	D	E
1		**link**	**pos**	**neg**	**webload**
2	**1**	(6081920, 157504)	[2, 37, 67, 71]		[2, 37, 67, 71, 131, 180, 231, 234, 296, 312, 770, 808]
3	**2**	(38656, 192832)		[5, 1337, 1339, 2312]	
4	**3**	(168000, 4084480)			[18, 109, 133, 171, 183, 186,
5	**4**	(137728, 123520)		459	[35, 192, 652, 1526, 1877,
6	**5**	(16, 5047040)			[46, 887, 1701, 2458, 2467]
7	**6**	(152512, 44672)			[201, 386, 393, 399, 449, 678,
8	**7**	(225920, 4183936)			[68, 225, 2084]
9	**8**	(2304, 3168)			[74, 90, 120, 139, 319, 357,
10	**9**	(38656, 46464)			
11	**10**	(1046656, 169600)			[318, 545]
12	**11**	(22848, 100160)			[82, 726, 1569, 2519]
13	**12**	(48768, 3168)			[74, 90, 120, 139, 319, 357,
14	**13**	(3230464, 32640)			[98, 1573]
15	**14**	(140864, 4084480)			[18, 109, 133, 171, 183, 186,
16	**15**	(15168, 44736)			[268, 326, 472, 523, 714,
17	**16**	(41141248, 4084480)			194, 196, 211, 223, 239, 281,

Figure 5.3.6 Network flow analysis: The resulting DSS sets

Figure 5.3.6 shows the resulting sets, which are populated after the DSS formula is evaluated in each cell. The sets represent the packet numbers of the packets sent between the source and the destination given in the column ‘link’, and which have been delivered to destination.

The deductive spreadsheets can be an extremely useful tool for network flow analysis. Looking at the ‘webload’ column, we can trace the links where the network transmission has problems, like congestions, link failures, transmission errors, etc. For example, we observe that on rows 2 (between source with id 38656 and destination with id 192832) and 9 (between source with id 38656 and destination with id 46464), no packets have reached destination. For the network administrator, this can be an indicator of transmission problems along the two links.

Chapter 6. Conclusions and Future Work

In this thesis, we presented a tool for automatic learning of deductive spreadsheet formulas and demonstrated its use with three applications: transitive closure, program analysis and network flow analysis. Our method starts with some background knowledge and a few examples (instances) of the problem that is analyzed, given as input in the spreadsheet interface. Next, we use inductive logic programming to learn from the examples the Prolog rules that encode the problem and its solution. Finally, we translate the Prolog rules into equivalent deductive spreadsheet expressions, and output them in the spreadsheet.

The main attributes of our approach are its high usability and interactivity. The visual Excel interface is an easy-to-use method of storing and manipulating data, while the deductive engine underneath enables any modification that affects other relations to be instantly seen and propagated through the spreadsheet.

Possible extensions of the current work include a more complex interface design, where more than one predicate can be learned from the same spreadsheet. This way, we could, for example, design the program analysis application presented in Section 5.2 in a single spreadsheet. It is important to notice that for recursive definitions, such as the 'in_reach' and 'out_reach' predicates from the program analysis example we need to specify a temporal ordering for learning (i.e. in what order are the predicates learned, and the columns/sheets populated). Moreover, we would like to extend the translation procedure to correctly handle more complex definitions, such as any combination of tuples. Another future direction of interest is to use the tool to explore more complex real-world applications, such as vulnerability analysis of computer systems.

Bibliography

[1] K. Sagonas, T. Swift, D. S.Warren, J. Freirre, and P. Rao, XSB programmers manual, 2001, http://xsb.sourceforge.net/, Accessed July 2008.

[2] The Aleph Manual,
http://web2.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html, Accessed July 2008.

[3] C. R. Ramakrishnan, I. V. Ramakrishnan, and D. S. Warren, Deductive spreadsheets using tabled logic programming, In 22nd International Conference on Logic Programming (ICLP), volume 4079 of Lecture Notes in Computer Science, pages 391-405, Springer-Verlag, 2006.

[4] Anu Singh, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott D. Stoller, and David S. Warren, Security Policy Analysis using Deductive Spreadsheets, In Proceedings of the 5th ACM Workshop on Formal Methods for Security Engineering (FMSE), 2007.

[5] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs, In PADL, volume 3819 of Lecture Notes in Computer Science, pages 215-229, Springer, 2006.

[6] D. Maier and D. S. Warren, Computing with Logic: Logic Programming and Prolog, Benjamin/Cummings Publishers, Menlo Park, CA, 1988, ISBN 0-8053-6681-4, 535 pp.

[7] D. Merrit, J. Paine, and M. Kassof, Special Spreadsheet Issue of AI Expert Newsletter, May 2005, http://www.ainewsletter.com/newsletters/aix_0505.htm, Accessed July 2008.

[8] G. Gupta and S. F. Akhter, Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs, In Practical Aspects of Declarative Languages (PADL), volume 1753 of Lecture Notes in Computer Science, pages 308-323, Springer, 2000.

[9] M. Kassoff, L.-M. Zen, A. Garg, and M. Genesereth, PrediCalc: A logical spreadsheet management system, In 31st International Conference on Very Large Databases (VLDB), 2005.

[10] B. Jayaraman and K. Moon, Subset logic programs and their implementation, J. Log. Program., 42(2):71-110, 2000.

[11] S. P. Jones, A. Blackwell, and M. Burnett, A user-centered approach to function in Excel, In Intl. Conf. on Functional Programming, 2003.

[12] G. Gupta and S. F. Akhter, Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs, In Practical Aspects of Declarative

Languages (PADL), volume 1753 of Lecture Notes in Computer Science, pages 308-323, Springer, 2000.

[13] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky, Programming with sets; an introduction to SETL, Springer-Verlag, New York, NY, USA, 1986.

[14] S.H. Muggleton and L. De Raedt (1994), Inductive Logic Programming: Theory and Methods, Jnl. Logic Programming, 19, 20:629-679.

[15] S. Dzeroski, N. Lavrac, Inductive Learning in Deductive Databases, IEEE Transactions on Knowledge and Data Engineering, Volume 5 , Issue 6, Pages: 939 – 949, 1993.

[16] P. N. Bennett, S. T. Dumais and E. Horvitz, Inductive transfer using layered abstraction-based ensemble learning: Modeling text classifier reliability, In ICML Workshop on The Continuum from Labeled to Unlabeled Data in Machine Learning and Data Mining, 2003.

[17] S. T. Dumais, J. Platt, D. Heckerman and M. Sahami, Inductive learning algorithms and representations for text categorization, In Proceedings of ACM-CIKM98, Nov. 1998, pp. 148-155, 1998.

[18] Alexey Loginov and, Thomas W. Reps and Shmuel Sagiv, Abstraction Refinement via Inductive Learning, CAV 2005, 519-533.

[19] Tang L P R, Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining. Dissertation Thesis, University of Texas at Austin, 2003.

[20] N. Lavrac and S. Dzeroski, Inductive Logic Programming: Techniques and Applications. Ellis Horwood, New York, 1994, ISBN 0-13-457870-8.

[21] S. Muggleton, Inverse Entailment and Progol, New Generation Computing Journal, Vol. 13, pp. 245-286, 1995.

[22] S. Muggleton and C. Feng, Efficient induction of logic programs, In S. Muggleton, editor, *Inductive Logic Programming*, pages 281-298. Academic Press, London, 1992.

[23] S. Muggleton, R. King, and M. Sternberg, Protein secondary structure prediction using logic-based machine learning, *Protein Engineering*, 5(7):647-657, 1992.

[24] PharmaDM – Dmax™, http://www.pharmadm.com/dmax.asp, Accessed July 2008.

[25] Eric McCreath – Lime, http://cs.anu.edu.au/people/Eric.McCreath/lime.html, Accessed July 2008.

[26] ACE Datamining System, http://www.cs.kuleuven.ac.be/~dtai/ACE/, Accessed July 2008.

[27] Tilde and Warmr, http://www.cs.kuleuven.ac.be/~ml/Doc/TW_User/, Accessed July 2008.

[28] RSD, Relational Subgroup Discovery through First-Order Feature Construction, http://labe.felk.cvut.cz/~zelezny/rsd/, Accessed July 2008.

[29] DL-Learner, http://dl-learner.org/Projects/DLLearner, Accessed July 2008.

[30] J. R. Quinlan, Learning Logical Definitions from Relations, Machine Learning Journal, Volume 5, Issue 3, Pages: 239 – 266, 1990.

[31] J.W. Lloyd, Logic for Learning: Learning Comprehensible Theories from Structured Data, Springer, Cognitive Technologies Series, 2003.

[32] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, Using Datalog with binary decision diagrams for program analysis, In Third Asian Symposium on Programming Languages and Systems (APLAS), volume 3780 of Lecture Notes in Computer Science, pages 97-118, Springer-Verlag, 2005.