Leibniz System

Development Tool
for Logic-based
Intelligent Systems

K. Truemper

Cover Art:
The author gratefully acknowledges permission to use photos 3,65:28 / 3,65:35 / 3,65:59 of the book *Das letzte Original, Die Leibniz-Rechenmaschine der Gottfried Wilhelm Leibniz Bibliothek* (The Last Original, the Leibniz Calculator of the Gottfried Wilhelm Leibniz Library) by Ariane Walsdorf, Klaus Badur, Erwin Stein, and Franz Otto Kopp, with preface by Georg Ruppelt.
Photographs: Maike Kandziora.
© 2014 Gottfried Wilhelm Leibniz Bibliothek - Niedersächsische Landesbibliothek, Hannover, Germany.

The book covers the history of the Leibniz calculator, technical construction, and mathematical aspects of the invention. The book includes about 150 photos, diagrams, and sketches of the calculator and related material.

Cover Design: Ingrid Truemper

Library of Congress Cataloging-in-Publication Data

Truemper, Klaus, 1942–
Leibniz System: Development Tool for Logic-based Intelligent Systems / Klaus Truemper.
Includes bibliographical references and subject index.
ISBN 0-9663554-7-4
1. Software. 2. Intelligent Systems
I. Title.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Table of Contents

Version 16.0 1 July 2015

Leibniz System

Development Tool for Logic-based Intelligent Systems

Preface

In 1989, we embarked on the creation of a toolbox for the design and implementation of intelligent systems.

The Name 'Leibniz System'

The toolbox is named the *Leibniz System* in honor of the mathematician, engineer, inventor, and philosopher Gottfried Wilhelm Leibniz (1646 - 1716). He invented calculus, as did Isaac Newton independently; constructed an effective mechanical calculator; and is the father of computational logic. The name *Leibniz System* was chosen to honor his contributions to logic and computing.

The Leibniz calculator performs the four basic arithmetic operations: addition, subtraction, multiplication, and division. It was the first computational device to do so. Two prototypes of the calculator were built. Both were lost, but the second one was found in an attic in 1876. The calculator is now part of the collection of the Gottfried Wilhelm Leibniz Library, Hannover, Germany.

The title page of the manual shows details of the calculator, using photos 3,65:28 / 3,65:35 / 3,65:59 by Maike Kandziora that are part of the book *Das letzte Original: Die Leibniz-Rechenmaschine der Gottfried*

Version 16.0 1 July 2015

Wilhelm Leibniz Bibliothek (The Last Original, the Leibniz Calculator of the Gottfried Wilhelm Leibniz Library), by Ariane Walsdorf, Klaus Badur, Erwin Stein, and Franz Otto Kopp, with preface by Georg Ruppelt; published by Gottfried Wilhelm Leibniz Bibliothek, 2014. The author gratefully acknowledges permission to use the photos.

The book covers the history of the Leibniz calculator, technical construction, and mathematical aspects of the invention. The book includes about 150 photos, diagrams, and sketches of the calculator and related material.

The following quote is often cited in connection with Leibniz's calculator:

"...it is beneath the dignity of excellent men to waste their time in calculation when any peasant could do the work just as accurately with the aid of a machine." - Gottfried Wilhelm von Leibniz

Development of Leibniz System

In the first step of the development of the *Leibniz System*, a module for logic computation was created.

Subsequently, machine learning of logic relationships was tackled, culminating in a subgroup discovery module where the characterization of subgroups is accomplished with humanly understandable formulas.

Lastly, optimization and related issues were covered where no assumption is made about the mathematical characteristics of the underlying functions.

After more than 25 years of system development, we are now stopping our effort and releasing the *Leibniz System* under the Lesser General Public License (LGPL), which allows free-of-charge use in almost any commercial and noncommercial environment. This includes that the programs can be inserted into other programs or systems that are proprietary or public-domain and that are distributed commercially or free-of-charge under a user-defined license. All such choices are permitted under the LGPL. We would like to see the widest possible distribution, and so encourage users to recommend the code to others.

Of course, the toolbox is not complete. But the tools created so far have proved to be helpful in various settings. We would be pleased if others took over and worked on expansion of the system, as long as the generous conditions of the LGPL remain and are not replaced by restrictive licenses prohibiting use in proprietary systems.

Page Numbering and Printing

The pages of this manual have been numbered so that the page numbers in the table of contents and the subject index agree with the page numbers listed for the .pdf file. For example, if the table of contents lists page 118 for a given section, then page 118 of the pdf file displays the desired page. Of course, the page itself shows that same number.

The manual is formatted for two-sided printing.

Acknowledgments

The creation of the *Leibniz System* has involved a great many people.

R. Karpelowitz programmed much of the lbcc compiler over a three-year period with exemplary dedication and precision. C. Ratliff assisted with testing of early code.

As soon as the first operational version was completed, it was used in classes at my home university, the University of Texas at Dallas. Over the years, students built a large variety of interesting intelligent systems.

Several researchers, a number of Ph.D. students working under my guidance, and other students have been part of a major effort to develop

additional ideas, programs, and modules. In particular, the following persons and programs should be mentioned.

The FasTrac system for dynamic traffic control, developed with G. Rinaldi and G. Felici, IASI-CNR, Rome, Italy.

Y. Zhao's learning system for spell and syntax checking. The resulting Laempel System has been widely used since it learns user behavior and carries out a clever analysis of sentence structure.

H. Al-Mubaid's expansion of the Laempel System to the discovery of homophonic spelling errors based on analysis of prior texts.

J. Straach's OCHEM learning system for optimal management of hazardous materials. The system becomes smart just by being used.

Joint work with G. Felici on learning logic formulas from data, and subsequent implementation of the results by F.-S. Sun in the Lsquare program.

A. Remshagen's work on learning clauses for fast solution of logic problems. Subsequent collaboration with her at the University of West Georgia resulted in the program qallsatcc, which solves the problem Q-ALL SAT. Reliable solution of that problem is essential for efficient interactive query systems.

Work with S. Bartnikowski, M. Granberry, and J. Mugan on discretization of rational data, resulting in the cutcc program.

K. Riehl's work on data mining of logic explanations from numerical data, including key concepts for subgroup discovery.

K. Moreland's work on detection of important subgroups and rare classes in numerical data.

Joint research with G. Liuzzi, IASI-CNR, Rome, Italy, resulting in the parallelized optimization programs dencon and denpar, which are part of the multicc module.

Help by S. Le Digabel and C. Tribes, University of Montreal, with installation and use of the NOMAD software, which is part of the multicc module. In particular, they added the block evaluation feature to NOMAD, which is essential for full parallelization of multicc.

J. Janiga supplied problem instances for testing of the program redcc for dimension reduction.

The development of the theory underlying the *Leibniz System* was funded in Germany by the German Science Foundation (Deutsche Forschungsgemeinschaft) and the Alexander von Humboldt-Foundation, and it was funded in the United States by the National Science Foundation, the Office of Naval Research, and the author's home institution, the University of Texas at Dallas.

In contrast, the entire implementation effort was privately funded since other support could not be obtained. In turned out that private funding was an excellent choice, since complete flexibility was retained for the eventual distribution. Indeed, free-of-charge distribution of the system under the very flexible LGPL is now possible, a pleasing result.

We received extensive assistance during preparation of the software and manual for distribution. G. Rinaldi reviewed the manual and proposed a number of improvements. Indeed, the original layout of the manual was done by him twenty years ago. M. Walter and M. Jünger tested the software installation on Linux installations different from the one at our home university. I. Truemper designed the cover page using the earlier mentioned photos supplied by the Gottfried Wilhelm Leibniz Library. She also reviewed the manual.

Whatever errors or oversights remain, they are mine.

Chapter 1

Overview

The *Leibniz System* is a software package for the development and implementation of logic-based intelligent systems. The package covers various aspect of the construction of such systems with modules for logic computation, learning logic formulas from data, discretization of data, subgroup discovery from data, data estimation by a lazy learner, dimension reduction of models, decomposition of graphs and matrices, and solution of constrained optimization problems involving single or multiple objective functions. All programs are written in C using the ANSI standard. Installation is intended under Linux/Unix. Installation using other operating systems should not be difficult but is not covered in the manual.

License

The *Leibniz System* originally was companion software for the book *Design of Logic-based Intelligent Systems* by K. Truemper, published by Wiley, 2004. Anybody purchasing the book was licensed to download the original version and use it for commercial or noncommmercial applications, without any additional charge or license fee.

Since then, the *Leibniz System* has been significantly expanded and the requirement of book purchase has been removed. Instead, the system is now made available without charge under the GNU Lesser General Public License (LGPL), which is much less restrictive than the General Public License (GPL). In fact, the adjective "Lesser" is a misnomer. Instead, it should say "More General Public License" since so much more can be done under the LGPL than under the GPL.

The file lgpl.txt supplied with the installation files has the precise definition of LGPL. Informally speaking, the LGPL allows the modules of the *Leibniz System* to be "linked with ... a non-(L)GPLed program, regardless of whether it is free software or proprietary software. The non-(L)GPLed program can then be distributed under any terms if it is not a derivative work" (Wikipedia). In simple language, the Leibniz System can be freely and without charge used by or incorporated into noncommercial, or commercial, or public, or proprietary software, as long as that software is not derivative work and the use of the *Leibniz System* under the LGPL is acknowledged; see suitable citation below. Here, "not derivative" means that the software using the *Leibniz System* must differ in a non-trivial way from the *Leibniz System* software and doesn't constitute just a trivial reuse of the software under another name.

Citation

Please cite the *Leibniz System* as follows:

K. Truemper, “Leibniz System: Development Tool for Logic-based Intelligent Systems, Version 16.0,” Leibniz Company, Plano, Texas, USA, 2015.

BibTeX reference:

```
@Book{KTruemper,
 title = {Leibniz System: Development Tool for Logic-based
          Intelligent Systems, Version 16.0},
 publisher = {Leibniz Company},
 year = {2015},
 author = {K. Truemper},
 address = {Plano, Texas, USA},
}
```

Disclaimer of Warranty of Any Kind

Summary of System

The *Leibniz System* consists of

- lbcc, which compiles solution algorithms and interface codes from logic formulations, to be linked with user C programs;
- program qallsatcc for solving the logic problem Q-ALL SAT;
- programs lsqcc, optcc, pyrcc, and tstcc for learning logic formulas from logic data;
- program cutcc for transformation of rational and set data to logic data;
- module subcc for subgroup discovery from data;
- program matchcc for data estimation using a lazy learner;
- module redcc for reducing the dimension of function domains;
- program decc for decomposition of graphs and matrices;
- module prpcc containing a large number of programs for data transformation and manipulation;
- module multicc for solution of constrained optimization problems with single or multiple objective functions.

A portion of the *Leibniz System* carries out most but not all algorithms of the book *Design of Logic-based Intelligent Systems.*

Specifically, compiler lbcc and the output produced by that compiler carry out the algorithms of book Chapters 2 (Introduction to Logic and Problems SAT and MINSAT), 3 (Variations of SAT and MINSAT), 5 (Basic Formulation Techniques), and 6 (Uncertainty).

Program qallsatcc solves the problem Q-ALL SAT of Chapter 4 (Quantified SAT and MINSAT).

Programs lsqcc and cutcc cover the algorithms of book Chapters 7 (Learning Formulas) and 8 (Accuracy of Learned Formulas), except for the presently not treated algorithms of Section 8.7 (Multipopulation Classification).

With the programs of the *Leibniz System*, one can readily assemble software for the algorithms of book Chapters 4 (Quantified SAT and MINSAT), 9 (Nonmonotonic and Incomplete Reasoning), and 10 (Question-and-Answer Processes).

Since completion of the book in 2004, programs have been developed for data analysis, data estimation, reduction of function domains, decomposition, and optimization.

In the remainder of this chapter, the modules are summarized.

lbcc Compiler

The lbcc compiler produces a solution program for a logic problem formulated in the *Leibniz System* syntax. The compiler also generates C code for a certain transfer process. Execution procedures of the *Leibniz System* provide access to the solution program from any user program written in C. Via procedure calls, the user may verify theorems, get answers for queries, modify the problem, or extract information about the problem. The C code produced by the lbcc compiler for the transfer process defines the logic variables of the logic formulation within the user program. The transfer process moves the values of logic variables from the user program to the execution procedures and vice versa. These moves are automatically done and do not require any user programming or specification. The user can even compile logic programs at run time and dynamically load and execute them.

Schematically,

$$\begin{tikzcd}
\text{Problem Formulation} \arrow[r, Rightarrow] & \text{lbcc Compiler} \arrow[r, Rightarrow] & \text{Solution Program, Transfer Code}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Solution Program Execution Procedures} \arrow[r, Leftrightarrow, "\text{Transfer Code}"] & \text{User Program}
\end{tikzcd}$$

The manual includes several chapters with a complete description of the lbcc compiler, the execution procedures, and the transfer process.

The mathematics underlying the lbcc compiler is described in the book

Effective Logic Computation – Revised Edition by K. Truemper, published by the Leibniz Company, 2010, and available free of charge at `http://www.utdallas.edu/~klaus/lbook.html`.

Spell checking that learns user behavior is described in

"Effective Spell Checking by Learning User Behavior" by Y. Zhao and K. Truemper), *Applied Artificial Intelligence* 13 (1999) 725–742.

An example expert system that involves learning during a query process is described in

"Learning to Ask Relevant Questions" by J. Straach and K. Truemper, *Artificial Intelligence* 111 (1999) 301–327.

A traffic control system using logic modules is covered in

"Traffic Control: A Logic Programming Approach and a Real Application" by G. Felici, G. Rinaldi, A. Sforza, and K. Truemper, *Ricerca Operativa* 30 (2001) 39–60.

and in

"A logic programming based approach for on-line traffic control" by G. Felici, G. Rinaldi, A. Sforza, and K. Truemper, *Transportation Research Part C* 14 (2006) 175–189.

qallsatcc Program for solving Q-ALL SAT

The qallsatcc program solves the logic problem Q-ALL SAT, which often arises in logic-based intelligent systems.

For details of the problem, see Chapters 4 of the book *Design of Logic-based Intelligent Systems.* Suffice it to say here the following. Suppose an intelligent systems asks for information and, based on the answers, concludes certain results or establishes that more information is needed. The latter decision requires solution of an instance of Q-ALL SAT.

The algorithm carried out by program qallsatcc is described in

“A Solver for Quantified Formula Problem Q-ALL SAT” by A. Remshagen and K. Truemper, available at
`http://www.utdallas.edu/~klaus/Wpapers/qallsatcc.pdf`

The algorithm is based on a prior method covered in

“An Effective Algorithm for the Futile Questioning Problem” by the same authors, *Journal of Automated Reasoning* 34 (2005) 31-47.

lsqcc Program for Learning Logic Formulas

The lsqcc program extracts logic formulas from logic data. The extraction process is fully automated and does not require any user preparation such as selection of parameters, tuning for data structure, or manual correction.

Besides the logic formulas, program lsqcc also outputs probability distributions for certain votes that are produced when the logic formulas are applied to new data. For that situation, program lsqcc also supplies probability distributions that may be used for error control.

The mathematics underlying the lsqcc program is described in Chapters 7 and 8 of the book *Design of Logic-based Intelligent Systems*; in the book chapter

"Learning Logic Formulas and Related Error Distributions" by G. Felici, F. Sun, and K. Truemper, in *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques* (E. Triantaphyllou and G. Felici, eds.), Springer-Verlag, Berlin, 2006, pp. 193–226; and in

"A Method for Controlling Errors in Two-class Classification" by G. Felici, F. Sun, and K. Truemper, *Proceedings of 23rd Annual International Computer Software and Applications Conference* (COMPSAC '99), 186–191.

The construction of consistent and stable explanations is treated in "Construction of Deterministic, Consistent, and Stable Explanations from Numerical Data and Prior Domain Knowledge" by K. Riehl and K. Truemper, available at
`http://www.utdallas.edu/~klaus/Wpapers/explanation.pdf`.

An application where context-based spelling errors are detected and corrected is described in "Learning to Find Context-based Spelling Errors" by H. Almubaid and K. Truemper, in "Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques" (E. Triantaphyllou and G. Felici, eds.), Springer-Verlag, Berlin, 2006, pp. 597–627.

An application to cancer diagnosis is covered in

"Application of a New Logic Domain Method for the Diagnosis of Hepatocellular Carcinoma" by G. Di Giacomo, G. Felici, R. Maceratini, and K. Truemper, *Proceedings of 10th Triennial Congress of the International Medical Informatics Association* (MEDINFO2001), 434–438.

A special case of data mining where needles are to be found in a haystack when just one needle is initially given, is covered in

"The Needles-in-Haystack Problem" by K. Moreland and K. Truemper, *Proceedings of International Conference on Machine Learning and Data Mining* (MLDM 2009) (2009) 516–524.

There are three auxiliary subroutines for the lsqcc program, called optcc, pyrcc, and tstcc.

The optcc program computes optimal records from training records. This step is needed when data for training records can only be obtained via test at some costs.

The pyrcc program analyzes logic formulas produced by the lsqcc program and attempts to produce additional variables and related formulas that allow significant simplification of the logic formulas.

The tstcc program carries out the evaluation of testing records using logic formulas created by the lsqcc program.

cutcc Program for Transformation of Data

The cutcc program processes records with rational data and converts them to equivalent records with logic data. Such conversion is needed when one wants to extract logic relationships from records with rational data. Indeed, in that situation, one first transforms the records with rational data to ones with logic data using the program cutcc, and then extracts the underlying logic relationships from the logic data using program lsqcc.

The cutcc program may also be used to convert records with set data to equivalent records with logic data. In that case, one uses the formulas of Chapter 7 of the book *Design of Logic-based Intelligent Systems* to convert the set data to rational data, and then applies program cutcc to transform the rational data to logic data.

The mathematics underlying the cutcc program is described in the book chapter

"Transformation of Rational and Set Data to Logic Data" by S. Bartnikowski, M. Granberry, J. Mugan, and K. Truemper, in *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques* (E. Triantaphyllou and G. Felici, eds.), Kluwer Academic Publishers, New York, 2006, pp. 253–278.

subcc Module for Subgroup Discovery

The subcc module has programs subcc, sub2cc, and suballcc. The programs analyze rational and/or integer data and identify important subgroups and corresponding explanations for specified targets.

The underlying method is called SUBARP. It is summarized in

"Subgroup Discovery Method SUBARP" by K. Truemper, available at `http://www.utdallas.edu/~klaus/Wpapers/subarp.pdf`

An application in Speech Therapy is described in

"Data-driven subclassification of speech sound disorders in preschool children," by J. Vick, T. Campbell, L. Shriberg, J. Green, K. Truemper, H. Rusiewicz, and C. Moore, *Journal of Speech, Language, and Hearing Research*, 2015.

matchcc Program for Data Estimation

Program matchcc accepts a training file and a testing file, each of which contains a collection of vectors of same dimension, indeed defined for the same set of attributes. A proper subset of the attributes is marked. Program matchcc estimates the values of the marked attributes in the testing file using all vectors of the training file and the values of the unmarked attributes of the testing file. The underlying algorithm is a lazy learner that is a particular locally weighted learning algorithm.

The program is particularly effective when the marked attributes when viewed as functions of the other attributes, have irregular or disconnected contours that defy approximation by regression approaches.

Details of the lazy learner are described in

"Dimension Reduction of Chemical Process Simulation Data" by G. Janiga and K. Truemper, available at
`http://www.utdallas.edu/~klaus/Wpapers/reduce.pdf`

The lazy learner is a particular locally weighted learning algorithm; see

C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," *Artificial Intelligence Review* 11 (1997) 11–73.

redcc Module for Reduction of Dimension

The redcc module uses programs redcc, sub2cc, and matchcc to reduce the dimension of function domains. Due to the use of matchcc, cases can be handled where the contours of the function are irregular or disconnected, and thus defy approximation by regression.

Details are covered in

"Dimension Reduction of Chemical Process Simulation Data" by G. Janiga and K. Truemper, available at
`http://www.utdallas.edu/~klaus/Wpapers/reduce.pdf`

decc Program for Decomposition of Graphs and Matrices

The decc program decomposes graphs and matrices either into components that are linked in tree fashion or in linear fashion. The programs allow analysis of complex systems and their decomposition into much simpler components.

The decomposition method is based on Chapter 12 of the book *Effective Logic Computation – Revised Edition*, available at `http://www.utdallas.edu/~klaus/lbook.html`.

prpcc Module for Data Preparation

A large number of programs are supplied for preparation of data. For example, data given by excel tables are readily transformed to input files for programs cutcc , subcc, sub2cc, suballcc, matchcc, and redcc.

multicc Module for Optimization

The multicc module has programs multicc, dencon, denpar, nomad, seqpen, gencc, and nsga that, jointly used, solve constrained optimization problems where one or several objective functions are to be minimized. Several of the programs rely on prior work. For the complete license statement, see the file multicc.c in Leibniz/Multicc/Code.

The function and constraint values may be provided by black boxes that upon input of a solution vector return the function and constraint values. This feature allows optimization of problems where an explicit description of the functions and constraints is not available, for example, when the values are obtained via simulation.

In the case of multiple objective functions, the set of efficient solutions, also called the Pareto front, is computed that comprehensively depicts the trade-off among the objective functions.

When the computing effort of black boxes becomes significant and thus requires considerable time, parallelized versions of the programs allow distribution of the black box evaluations over multiple processors and thus rapid solution of very difficult optimization problems. This feature is available regardless of the number of objective functions to be minimized.

The mathematics used by the programs is summarized in

"Simple Seeding of Evolutionary Algorithms for Hard Multiobjective Minimization Problems" by K. Truemper, available at
`http://www.utdallas.edu/~klaus/Wpapers/seed.pdf`

and

"Parallelized hybrid optimization methods for nonsmooth problems using NOMAD and linesearch" by G. Liuzzi and K. Truemper, available at `http://www.utdallas.edu/~klaus/Wpapers/hybrid.pdf`.

Chapter 2

Getting Started

Installation

This section explains the system installation. If the system has already been installed, the user should ignore the discussion of this section and instead should proceed directly to the section The First Time.

Go to the website

`www.utdallas.edu/~klaus/Leibnizprogram/leibnizmain.html`

and download the files leibniz.complete.zip and installation.pdf. Carry out the installation instructions of the file installation.pdf.

When installation has been completed, the Leibniz directory contains in alphabetical order the following directories: Cutcc, Decc, Expertsys, GlobalVersion, Lbcc, Lsqcc, Manual, Matchcc, Multicc, Nomad, Prpcc, Qallsatcc, Redcc, Subcc, Sub2cc, Suballcc, and Utility.

The Nomad directory contains the NOMAD optimization software supplied by `https://www.gerad.ca/nomad/Project/Home.html`.

Directory names terminating with “cc” contain C programs. Indeed, the name of the main program in such a directory also ends with “cc.”

The convention may seem strange. When work on the *Leibniz System* began many years ago, in 1989, all programming was done in FORTRAN. But gradually the programs were translated to C, and the tag "cc," standing for "C code," was added to the program names to indicate translated code. By the mid-1990s all FORTRAN code had been translated to C. For consistency's sake, subsequently written C programs used the same naming convention.

Each directory with name ending in "cc" has three subdirectories: Code, Doc, and Makedata. The Code directory has the C source code in a file xxx.code.zip, the Doc directory has source code documentation in a file xxx.doc.zip, and the Makedata directory has a file xxx.makedata.zip that typically contains files for example execution.

The directories discussed next in alphabetical order.

- Cutcc has the cutcc program for conversion of rational data to logic data.
- Decc has the decc program for decomposition of graphs and matrices.
- Expertsys has example expert systems constructed with the lbcc compiler.
- GlobalVersion has files defining the version of the system.
- Lbcc has the lbcc compiler and related execution routines.
- Lsqcc has the programs lsqcc, optcc, pyrcc, and tstcc for learning logic from data.
- Manual has the manual in file leibnizmanual.pdf. The manual is formatted for two-sided printing.
- Matchcc has the matchcc program for data estimation.
- Multicc has the multicc program for the solution of optimization problems.
- Qallsatcc has the qallsatcc program for the solution of problem Q-ALL SAT.
- Prpcc has a number of programs for data preparation and manipulation.
- Redcc has program redcc for the dimension reduction of data sets.

- Subcc, Sub2cc, and Suballcc have programs subcc, sub2cc, and suballcc for subgroup discovery from data sets.
- Utility has the program file2pdf, which converts an ASCII file to a Plain TEX file and then compiles the latter file to .pdf format. The directory also contains variations of file2pdf that produce .dvi and .ps files.
  For page control, the ASCII files may contain the line
  `/*eject*/`
  It triggers the start of a new page during the conversion process. The files of the C code of the various programs employ `/*eject*/` records to start each routine on a new page and generally organize the code into blocks of statements.
  Some programs produce voluminous ASCII output files, in particular the programs of the subcc module. These output files may also contain `/*eject*/` records, and conversion to .pdf format results in files that are easy to read.

All C source code follows the ANSI standard and thus can be compiled by virtually any C/C++ compiler.

Linux/Unix Installation

It is assumed that the gcc compiler is available. The compilation and linking steps are accomplished by executing in directory Leibniz

```
make -f makefile.install
```

This step first installs NOMAD software and then the *Leibniz System* software. The user may want to define aliases in .cshrc or .bashrc for all program names. Since all program names terminate in "cc", a convenient alias is obtained by just deleting the last "c". For example, program lbcc has the alias lbc. Alternately, the user may specify the paths to the various Code directories and then use the program names directly. In the remainder of the manual, we assume the latter case and thus use all program names without modification.

CAUTION: Programs redcc, subcc, sub2cc, and suballcc create so-called detail directories in the local Linux /tmp directory. If the latter directory does not exist, the user should create a local directory with that name.

CAUTION: The code has been tested on CentOS, Debian, and Ubuntu Linux. Compilation under Debian as well Ubuntu produces a number of warning messages that arise from certain programming practices and should be ignored. We will expand this list as tests on additional Linux systems are completed. If a run problem is encountered, it may very well be caused by local limits on stack memory. So if an unexpected termination occurs claiming "segmentation fault," first try the following. First remove the stack limits by executing the bash command

```
ulimit -S -s unlimited
```

and then run the program again.

All Other Installations

For all non-Linux/Unix installations, the following steps are required.

First, install NOMAD according to the steps defined in makefile.install.

Second, go into each one of the lowest subdirectories of Leibniz and unzip the .zip file found there.

Third, in Code subdirectories of Prpcc, Lbcc, Decc, Cutcc, Lsqcc, Subcc, Sub2cc, Suballcc, Matchcc, Qallsatcc, Redcc, Multicc, and Utility, use a suitable C/C++ compiler and carry out compilation and linking steps described in the makefile.install file of that subdirectory.

CAUTION: Compilation must be done in the Code subdirectories in the indicated order.

The First Time

Here is an easy-to-follow recipe to get going with each program or module.

lbcc Compiler

First, copy into a new directory the file lbcc.makedata.zip of Leibniz/Lbcc/Makedata, and unzip that file. All steps below are carried out in the new directory.

Second, access the file makefile.machine. Change the line

```
LEIBNIZ = /home/Leibniz/Lbcc/Code/
```

so that the complete path ending in Leibniz/Lbcc/Code/ is specified.

Third, execute

```
make -f makefile.machine
```

The makefile invokes the Leibniz compiler lbcc with the file leibnizparams.machine for compilation of the logic formulation of file machine.log. The output consists of a program file machine.prg, a transfer process file leibniztrans.c, and three files machine.trs, leibnizdefs.h, and leibnizexts.h. While lbcc is being executed, some information about the compile process is displayed. Ignore that information for the time being. Next, the makefile invokes the gcc compiler to compile the C code of files machine.c, leibniztrans.c, and leibnizerroraction.c and to link the resulting files with certain object files of Leibniz/Lbcc/Code.

Fourth, execute

```
./machine
```

to enter the expert system coded in machine.c and machine.log. Follow the instructions and provide answers to questions posed by the expert system. When done, scan the C code of machine.c and the logic formulation of machine.log to get an idea how the expert system analyzes various scenarios.

The rules for the construction of logic formulations are described in Chapter 3. Compilation of logic formulations is covered in Chapter 4, together with the use of compiled solution programs and execution procedures in user C programs.

qallsatcc Program

Program qallsatcc solves the problem Q-All Sat, which is as follows. Given are two formulas R and S in conjunctive normal form (CNF). There are three types of variables: q-variables, x-variables, and y-variables. The formula R uses the q-variables and x-variables, while the formula S uses the q-variables and y-variables.

Q-ALL SAT asks that the following question be decided: Are there values for the q-variables such that formula R can be satisfied with suitably selected values for the x-variables, and formula S cannot be satisfied no matter how the values for the y-variables are chosen?

If the answer to the question is "yes," then program qallsatcc outputs on the screen "Have a solution." If the answer is "no," then the output is "Have NO solution.'

For a detailed discussion of Q-ALL SAT and applications for intelligent systems, see Chapter 4 of the book *Design of Logic-based Intelligent Systems*.

For a first try of program qallsatcc for solving problem Q-ALL SAT, proceed as follows.

First, copy into a new directory the file qallsatcc.makedata.zip of Leibniz/Qallsatcc/Makedata. All subsequent steps are carried out in the new directory.

Second, open the file book.log. It is an encoding of a small instance of Q-ALL SAT.

There are three q-variables q_1, q_2, and q_3. Formula R uses just those variables, so there are no x-variables. Formula S uses the q-variables plus three y-variables y_r, y_s, and y_t.

As an aside, the Q-ALL SAT instance is discussed in Chapter 4, Section 4.2, of the book *Design of Logic-based Intelligent Systems*, except that variables r, s, and t of the book correspond here to y_r, y_s, and y_t due to input requirements of program qallsatcc.

Execute

```
Leibniz/Qallsatcc/Code/qallsatcc book.log
```

The program produces the screen output

```
INSTANCE book
Time: 0
no. R-conflict clauses: 0
no. S-conflict clauses: 0
no. nodes in Qallsat tree: 1
total no. nodes in Sat trees: 2
No. times easy part is used: 0
Have NO solution
```

Ignore the statistics of the algorithm and focus on the last line, `Have NO solution`.

It means that, whenever values for q_1, q_2, and q_3 are chosen so that they satisfy formula R, then there are always values for y_r, y_s, and y_t such that formula S is satisfied.

Now change in book.log the line

```
not q_1. name r_2
```

to

```
q_1. name r_2
```

Once more execute

```
Leibniz/Qallsatcc/Code/qallsatcc book.log
```

This time, the screen output terminates with

```
Have a solution
```

which means that values for q_1, q_2, and q_3 exist such that R is satisfied and, no matter how values are chosen for y_r, y_s, and y_t, formula S is not satisfied.

For yet another try, the file qprob.log of Leibniz/Qallsatcc/Makedata has a formula R with both q- and x-variables, and a formula S with both q- and y-variables.

Execute

```
Leibniz/Qallsatcc/Code/qallsatcc qprob.log
```

The conclusion is `Have NO solution`. Thus, whenever values for the q-variables are chosen so that formula R can be satisfied using suitable values for the x-variables, then there are always values for the y-variables so that S is satisfied as well.

Details of program qallsatcc are described in Chapter 6.

lsqcc Program

For a first try of program lsqcc for learning logic, the user should proceed as follows.

First, copy into a new directory the file lsqcc.makedata.zip of Leibniz/Lsqcc/Makedata. All subsequent steps are done in the new directory.

Second, execute

```
Leibniz/Lsqcc/Code/lsqcc lsqccparams.data1
```

Program lsqcc analyzes the data of the training file data1.trn, extracts logic formulas, and computes certain distributions and probabilities. The logic formulas are outputted in the file data1.sep, while the distributions and probabilities are provided in the file data1.dis. The user may employ any text editor to look at these files.

Program lsqcc also applies the formulas of data1.sep to data in the testing file data1.tst and outputs the results in the file data1.vot. The user may want to look at these two files. Details about program lsqcc and the above files—data1.trn, data1.sep, data1.dis, data1.tst, and data1.vot—are provided in Chapter 7.

cutcc Program

For a first try of program cutcc for transformation of data, the user should proceed as follows.

First, copy into a new directory the file cutcc.makedata.zip of Leibniz/Cutcc/Makedata, and unzip that file. All subsequent steps are done in the new directory.

Second, execute

```
Leibniz/Cutcc/Code/cutcc lsqccparams.heart
```

The program analyzes the rational data and set data of the training file heart.rtr, determines certain cutpoints, and transforms the data into logic data which are placed into a file heart.trn. The latter file plus a second output file heart.ptl may be used as input files for program lsqcc. For interpretation and analysis of the transformation process, program cutcc produces additional output files heart.cut and heart.vis that show the cutpoints.

Program cutcc also transforms the testing file heart.rts with rational data and set data to a file with logic data, which is called heart.tst. The latter file is another input file for program lsqcc.

Details about program cutcc and the above files—heart.rtr, heart.ptl, heart.cut, heart.vis, heart.rts, and heart.tst—are provided in Chapter 8.

subcc Program

For a first try of program subcc for the discovery of subgroups from data, the following steps should be done.

First, copy into a new directory the file subcc.makedata.zip of the subdirectory Leibniz/Subcc/Makedata, and unzip that file. All steps below are done in the new directory.

Second, open the file lsqccparams.heart.dat with any editor and replace on the line

```
Leibniz directory = ~/Leibniz/
```

the right-hand-side path by the path to the Leibniz directory.

Third, execute

```
Leibniz/Subcc/Code/subcc lsqccparams.heart.dat
```

The program analyzes the data of master file heart.mst to discover important subgroups. To this end, the file heart.mst defines attributes in the section titled ATTRIBUTES. Some of the attributes are labeled as targets; for example, the line

```
chol        TARGETCASES_1_10_10
```

says that the attribute chol is a target of a certain type. In the subsequent DATA section, each line is a record that has numerical values for all attributes.

Program subcc derives from the file heart.mst a target file heart.tgt that contains possible discretization intervals for the target attributes. For these target intervals, the program attempts to find in the data of heart.mst relationships that link targets with other attributes and that are unlikely to be due to random variations. The relationships are outputted in the file heart.mst.sub.

Program subcc evaluates the importance of the relationships using the alternate testing file heart.ats. The relationships and their significance are then outputted in the file heart.ats.sub.

The reader may want to look at the files heart.mst.sub and heart.ats.sub to get an idea how the relationships and their importance are exhibited.

Details about program subcc are included in Chapter 9.

sub2cc Program

Program sub2cc executes program subcc, then sorts the output files so that the most significant cases are listed firsts.

For a first try of program sub2cc, go into the new directory defined previously for the trial execution of subcc, and execute

```
Leibniz/Sub2cc/Code/sub2cc lsqccparams.heart.dat
```

The program calls subcc and once more produces the files heart.mst.sub and heart.ats.sub. In a second step, the program sorts the data of the file heart.ats.sub and outputs the records in the file heart.sub2.

Details about program sub2cc are included in Chapter 9.

suballcc Program

Program suballcc repeatedly executes program sub2cc while iteratively deleting attributes. As a result, suballcc narrows the focus on attributes, and by repeated evaluations deduces crucial relationships.

For a first try of program suballcc, go into the new subdirectory defined previously for the trial execution of subcc and sub2cc, and execute

```
Leibniz/Suballcc/Code/suballcc lsqccparams.heart.dat
```

The final file heart.suball contains key results extracted from the output of the numerous sub2cc calls.

Details about program suballcc are included in Chapter 9.

matchcc Program

Program matchcc accepts a training file and a testing file, each of which contains a collection of vectors of same dimension. Indeed, each vector is defined for the same set of attributes. A proper subset of the attributes is marked with the label DELETE. Program matchcc estimates the values of the marked attributes in the testing file using all vectors of the training file and the values of the unmarked attributes of the testing file. The files are structured like the data files for the programs of module subcc. That is, the leading ATTRIBUTES section contains attributes names, and the subsequent DATA section has records, each of which has values for the attributes.

For a first try, copy into a new directory the file matchcc.makedata.zip of Leibniz/Matchcc/Makedata, and unzip the file. This step produces
- a README file that can be ignored for the moment,
- a training file small.train.mst,
- a testing file small.test.mst,
- an additional data file small.new.mst, and
- an executable file matchcc.small that executes the command

```
Leibniz/Matchcc/Code/matchcc \
     small.train.mst small.test.mst \
     match.result match.error match.learn \
     match.solution average.error
```

All subsequent steps are carried out in the new directory.

Open up the training file small.train.mst. It has an ATTRIBUTES section and a DATA section. The first two attributes obj_1 and obj_2 are marked DELETE. That option means that these attributes are to be estimated from the remaining attributes, which are xvalue_0 and xvalue_1. Each record of the DATA section has values for all attributes.

CAUTION: The DATA records contain only positive integers. Indeed, program matchcc demands that all numbers are integers ranging from 0 to at most 10^6. If other data are to be used, say rational data, then they must first be converted by scaling and rounding to integers falling within the demanded bounds. For example, if numbers for an attribute range from 0.0 to 1.0, then multiplication by 10^6 and rounding gives the desired integer values. The required range for the integer values implies that the precision of the input data is six digits.

The testing file small.test.mst has the same structure as the training file small.train.mst.

When the above command is executed, program matchcc uses all data of the training file to estimate the data for obj_1 and obj_2 in the testing file from the values for xvalue_0 and xvalue_1.

Output files match.result, match.error, match.learn, match.solution, and average.error provide estimated values for obj_1 and obj_2, maximum error and total error per record, and the overall average error. The file match.result has all details for each record of the testing file small.test.mst. The remaining files essentially have subsets of that information. For interpretation details, go to Chapter 10.

Go into the file small.new.mst. It has 0s for the values of obj_1 and obj_2, and various values for xvalue_0 and xvalue_1. Each record can be viewed as an instance where values for xvalue_0 and xvalue_1 are known, and values for obj_1 and obj_2 are unknown and are to be estimated.

Execute the same command as before, except that small.new.mst takes on the role of small.test.mst.

```
Leibniz/Matchcc/Code/matchcc \
    small.train.mst small.new.mst \
    match.result match.error match.learn \
    match.solution average.error
```

This time ignore the error results of the output files and focus on the estimated values for obj_1 and obj_2. They match the estimated values obtain in the earlier execution. This is not surprising since the values for xvalue_0 and xvalue_1 in the files small.new.mst and small.test.mst are identical.

For the moment, view the attributes marked DELETE to be functions of the other attributes. Consider the DATA records to be a sampling of these functions. Program matchcc is particularly effective when the functions have irregular or disconnected contours that make approximation by regression approaches difficult.

Details about program matchcc are included in Chapter 10.

redcc Program

The redcc module uses programs redcc, sub2cc, and matchcc to reduce the dimension of function domains. Due to the use of matchcc, cases can be handled where the contours of the function are irregular or disconnected, and thus make approximation by regression difficult.

For a first try, copy into a new directory the file redcc.makedata.zip of Leibniz/Redcc/Makedata, and unzip that file. All steps below are carried out in the new directory.

Second, open the file lsqccparams.methane.dat with any editor and replace on the line

```
Leibniz directory = ~/Leibniz/
```

the right-hand-side path by the path to the Leibniz directory.

Third, execute

```
Leibniz/Redcc/Code/redcc lsqccparams.methane.dat
```

The program applies matchcc and sub2cc in several rounds to the training file methane.mst and the testing file methane.ats to estimate the attribute Enthalpy, which is marked with TARGETCASES_1_20_20. The process examines all other attributes except for the two attributes X and Y, which have label DELETE * DISCARD. During the rounds, the unmarked attributes are eliminated until the remaining attributes just suffice to estimate Enthalpy with error less than the max error fraction specified in the file lsqccparams.methane.dat.

CAUTION: Due to the use of program matchcc, the numbers in the data records of the input files must be integers ranging from 0 to 10^6. If other data are to be used, say rational data, then they must first be converted by scaling and rounding to integers falling within the demanded bounds. For example, if numbers for an attribute range from 0.0 to 1.0, then multiplication by 10^6 and rounding gives the desired integer values. This process implies that the precision of the input data is six digits.

The output is in file methane.opt. Almost all attributes have been deleted, and the conclusion is that attributes H2 and H20, which are the only unmarked attributes in the output file, suffice to estimate the attribute Enthalpy with error less than the demanded max error fraction.

The example run involves estimation of just one function, Enthalpy. But domain reduction is also possible when a common reduced domain is to be found for several functions. In such a case, each attribute representing a function is marked with TARGETCASES_1_20_20.

Execution of redcc is computationally expensive due to numerous runs of sub2cc. If the run time of redcc becomes unacceptably large, the user may want to execute the alternate, faster program redgreedycc. The method does not rely on sub2cc, and instead is a straightforward greedy method that directly evaluates reduction choices with program matchcc. As a result, redgreedycc may achieve less of a domain reduction than redcc.

For a trial run of redgreedycc, execute

```
Leibniz/Redcc/Code/redgreedycc lsqccparams.methane.dat
```

The solution is not quite as desirable. To see this, look at the output file methane.opt. In the earlier run, the attributes H2 and H20 sufficed to estimate Enthalpy with good accuracy. In contrast, the output file methane.opt produced by redgreedycc says that the three attributes OH, H2, and O2 suffice to estimate Enthalpy with good accuracy.

For another run with redcc, open the file lsqccparams.methane.plus.dat with any editor and replace on the line

```
Leibniz directory = ~/Leibniz/
```

the right-hand-side path by the path to the Leibniz directory.

Then execute

```
Leibniz/Redcc/Code/redcc lsqccparams.methane.plus.dat
```

The input files are methane.plus.mst and methane.plus.ats. The files have the same data records, but declare that the three attributes C2, N2, and Enthalpy are the functions to be estimated. The conclusion of the output file methane.plus.opt is that the three functions can be estimated with good accuracy by the same attributes H2 and H20 selected earlier just for estimation of Enthalpy.

With the same input files, the faster program redgreedycc determines that OH, H2, and O2 suffice for estimation of Enthalpy with good accuracy. So here, too, the faster program redgreedycc delivers an inferior solution.

Details about program redcc are included in Chapter 11.

decc Program

Program decc accepts a graph or matrix as input and carries out a decomposition into components that are linked in tree fashion or in linear fashion.

The input is similar to that for the lbcc compiler, except that the records represent rows of a matrix or edges of a graph instead of logic facts.

To start, copy into a new directory the file decc.makedata.zip of Leibniz/Decc/Makedata and unzip that file. All steps are done in the new directory.

Let's look at the file edgegraph.mat, which represents a small graph. The VARIABLES section lists z1, z2, ..., z13. They are the nodes of the graph. In the FACTS section, each record corresponds to an edge. For example, the line `z1 | z2.` says that an edge connects nodes z1 and z2. Lines starting with an asterisk `*` are comments and are disregarded.

For the matrix case, the file matrix.mat has variables x1, x2, ..., x392, each of which indexes one column of the matrix. Each fact encodes one row of the matrix. For example, the third fact line

```
x2 | x100 | x102 | x329 .  c2
```

encodes that the third row of the matrix has four nonzeros in columns x2, x100, x102, and x329.

The "c2" after the period of the record is the label of the row. Indeed, every row of the matrix is listed with a label. If a row does not have a label, then decc assign one of the form @<number> where number is a 6-digit integer with leading 0s. The same applies to the edges of a graph. For example, in the file edgegraph.mat no edge labels are specified, so for each edge decc generates a label of the form @<number>.

Suppose two graphs G_1 and G_2 are composed to a single graph G by identifying k nodes in one graph with k nodes of the other one. Decomposition is a reversal of this process, so G is decomposed into G_1 and G_2 where k nodes of G_1 have the same label as k nodes in G_2. The k nodes are the *connecting nodes* of the decomposition.

In graph algorithms, subsequent decomposition of G_i into G'_{i1} and G'_{i2} must always be so done that at most one of the k connecting nodes of the G_1-G_2 decomposition is part of the k' connecting nodes of the G'_{i1}-G'_{i2} decomposition. In graph decomposition algorithms, this is achieved by attaching, for $i = 1, 2$, to the k connecting nodes of G_i edges for all possible node pairs; that is, the additional edges define a clique.

Here, the computations are done using matrices, and instead of adding the clique edges, which would inflate edge sets, just one row is added to each representation of G_1 and G_2 that has 1s in the k columns and 0s elsewhere. We call that additional row a *clique row.* If there is just one connecting node, then we still create the clique row. Formally, the single 1 in that row represents a loop of the graph.

Execute

```
Leibniz/Decc/Code/decc deccparams.edgegraph.dat
```

For the moment we ignore the features of the parameter file deccparams.edgegraph.dat. The screen output displays progress as the graph is recursively decomposed.

The last line of the screen output is

```
*** Blocks are in file ./edgegraph.blk
```

The output in edgegraph.blk is organized by blocks. We ignore block 1, which lists

```
Row = *:  *
```

That specification represents an artifact of the decomposition process. Block 2 is one component of the decomposition. The block is defined in the sections `Matrix Rows` and `Matrix Columns`, which are discussed next.

In `Matrix Rows`, each row of block 2 corresponds to an edge of the original graph or to the clique row defining the attachment of block 2 to the rest of the graph. For example, the line

```
Row = @0000001:  z1 z2
```

says that the edge connecting z1 and z2 is part of block 2. The label `@0000001` is the edge label generated by decc as discussed above. Note the line

```
Row = clique.3:  z13
```

It is the clique row mentioned earlier. Since there is just one node mentioned, z13, the clique edge actually is a loop.

In `Matrix Columns`, each record for block 2 lists the edges incident at the nodes of the block. For example, the row

```
Col = z1:  @0000001 @0000002 @0000003
```

says that the edges @0000001, @0000002, and @0000003, which were defined in the `Matrix Rows` section, are incident at node z1. An edge label such as `clique.3` tells that the edge is part of clique.3.

The matrix decomposition case is similar, indeed easier to interpret, since row and columns in the input and output directly correspond to rows and columns of the given matrix.

Consider the example file matrix.mat. Each row has been explicitly given a label, so decc will not construct labels of the form @<number>.

Execute

```
Leibniz/Decc/Code/decc deccparams.matrix.dat
```

As before, the screen output shows progress and terminates with the line

```
*** Blocks are in file ./matrix.blk
```

Here, block 1 has not only the earlier-seen artifact row `Row = *:  *`, but also all occurrences where a matrix entry is the single entry in a row and a column.

The output records for blocks 2–28 directly represented the entries in columns and rows. The clique rows tell how the blocks are connected.

Details about program decc are included in Chapter 12.

prpcc Module

The module has a number of small C programs for data manipulation. The user may want to scan file prepare.pdf in Leibniz/Prpcc/Doc for details about these programs.

Data analyzed by the various Leibniz programs are often supplied in MS Excel files. These files must satisfy certain requirements that are described in the file excel_format.pdf of Leibniz/Prpcc/Makedata. An informal description is given in informal.explanation.pdf of the same directory. The directory also contains ASCII versions of these files.

Details of module prpcc are described in Chapter 13.

multicc Module

The module contains several programs for solving constrained optimization problems with one or more objective functions. In the case of several objective functions, the Pareto front of efficient solutions is computed.

For a first try, execute in a new directory

```
Leibniz/Multicc/Code/multicc zdt6 \
     -input NOINPUTFILE -objrange 1 1
```

Several programs of the module are invoked to solve the problem zdt6, which is one of many optimization example problems supplied with the installation.

The problem zdt6 has 10 variables with certain lower and upper bounds, and 2 objective functions. The execution of multicc produces extensive screen output that should be ignored for the time being. When execution stops, a graph displays the Pareto front, which consists of all efficient objective function pairs. Informally speaking, a point (f_1, f_2) is efficient if there is no other point (f_1', f_2') where $f_1' \leq f_1$ and $f_2' \leq f_2$ and strict inequality holds in at least one of the two cases.

Details of module multicc are described in Chapter 14.

At this point, the user has enough insight into the *Leibniz System* to read the remainder of the manual and run the various programs. For the construction of intelligent systems with the programs, the user may want to read the companion book *Design of Logic-based Intelligent Systems*.

Chapter 3

lbcc Compiler

Introduction

The lbcc compiler produces a solution program for a logic problem formulated in the *Leibniz System* syntax. The logic formulation is specified in a file with .log extension. The compiler outputs several files: a file with extension .prg, which contains the solution program; a file with extension .trs and the file leibniztrans.c, which together contain the C code of the transfer process; and files leibnizdefs.h and leibnizexts.h, which define variables for the transfer process. Except for the file with .prg extension, these files are later compiled and linked with the object code of the execution procedures and the user program to an executable program.

This chapter covers the syntax of the logic formulations and the use of the lbcc compiler. The transfer process, compilation of the generated files, and linking with user code are covered in Chapter 4.

The syntax of the logic formulations is not complicated. It adheres to conventions of first order logic, with suitable extensions to simplify the formulation.

We first present the input for an example problem that involves several rules of the syntax.

Example of Logic Formulation

We want to diagnose malfunctions of a machine for automated assembly. The file machine.log contains the logic formulation together with explanations. Here is the file. After reading the file below, the reader may want to carry out the steps described in Chapter 2 to see how that information is put to use, unless the user has already done so.

```
SATISFY
* The first line specifies that this is a
* logic satisfiability problem (in contrast to
* a logic minimization problem, which would be
* declared by 'MINIMIZE'). Next follows the name
* of the logic problem.
Automated_Assembly

* Description

* ****Checking Crash Scenarios****
* An automated assembly machine has two blocks among
* its components.
* Block_a moves between position_1 and position_2.
* Block_b moves between position_3 and position_4.
* If both blocks move simultaneously, then potentially
* they will collide and produce a crash of the machine.
* Objective is to find how the system may crash due to
* failure of switches at the four positions.

SETS
* In this section, sets for quantification are
* declared, to be used in conjunction with the
* predicates of the subsequent section.

* universe (not needed, but useful for looping in user code.
* Controls the sequence in which elements are recorded and
* given integer values.  Without this set, the elements are
* recorded in the order in which they are encountered in the
* definitions of sets.)
define universe
  position_1
  position_2
  position_3
  position_4
  z1
  z2
  z3
  z4

* end positions of blocks
define positions
  position_1
  position_2
  position_3
  position_4

* end positions of block_a
define positions_block_a
  position_1
  position_2

* end positions of block_b
define positions_block_b
position_3
position_4

* scenarios of crashes
define scenarios
  z1 * block_a moves from position_1 and
     * block_b moves from position_3
  z2 * block_a moves from position_1 and
     * block_b moves from position_4
  z3 * block_a moves from position_2 and
     * block_b moves from position_3
  z4 * block_a moves from position_2 and
     * block_b moves from position_4

PREDICATES
* In this section, predicates are specified.  Each predicate
* is defined over one or two of the earlier defined sets.

* lever_switch predicate:
* true means lever of switch is in on position
* the lever of the switch at position_1 or position_2
* is actuated by block_a
* the lever of the switch at position_3 or position_4
* is actuated by block_b
define lever_switch on positions

* switch_okay predicate:
* true means that switch is normally open and becomes closed
* when contacted by block
define switch_okay on positions

* move predicate:
* true means that block_a or block_b moves away from the position.
* example:
*               ____
*    o---------|____|->>>--o        move(position_1) = True
* position_1   block_a  position_2   move(position_2) = False

define move on positions

* output_signal predicate:
* true means that switch outputs signal.  this happens
* when the lever is pushed by the block or when the
* switch is stuck closed.
```

```
define output_signal on positions

* stuck_open/closed predicates:
* true means switch is stuck open/closed
define stuck_open on positions
define stuck_closed on positions

* conflict predicate:
* true means that conflict of the scenario is occurring
define conflict on scenarios

VARIABLES
* In this section, propositional variables
* are declared.

* crash variable
crash

FACTS
* In this section, the facts about the machine and
* its operation, as well as diagnostic statements,
* are described.  Each fact is terminated by a period.

* I. Mechanical operation

* if move, then levers at both ends are in off position
if move(position_1) or move(position_2)
    then not lever_switch(position_1) and
         not lever_switch(position_2).
if move(position_3) or move(position_4)
    then not lever_switch(position_3) and
         not lever_switch(position_4).

* block can be only at one end
not lever_switch(position_1) or not lever_switch(position_2).
not lever_switch(position_3) or not lever_switch(position_4).

* two moves of same block are not possible
not move(position_1) or not move(position_2).
not move(position_3) or not move(position_4).

* II. Failures of switches
* switch is okay if and only if it is not stuck open/closed
* switch cannot be both stuck open and stuck closed
for all pi in positions
    switch_okay(pi) or stuck_open(pi) or stuck_closed(pi).

for all pi in positions
    not switch_okay(pi) or not stuck_open(pi).

for all pi in positions
    not switch_okay(pi) or not stuck_closed(pi).

for all pi in positions
    not stuck_open(pi) or not stuck_closed(pi).

* switch signal depends on lever and on whether switch is okay
for all pi in positions
    if lever_switch(pi) and switch_okay(pi)
        then output_signal(pi).

for all pi in positions
    if not lever_switch(pi) and switch_okay(pi)
        then not output_signal(pi).

for all pi in positions
    if stuck_open(pi)
        then not output_signal(pi).

for all pi in positions
```

```
    if stuck_closed(pi)
        then output_signal(pi).

* III. Conditions in controller of machine

* the conditions are so chosen that machine should not crash
* regardless of the instruction sequence.  It says 'should'
* since the goal is to find out if these conditions prevent
* crashes under all possible circumstances.
for all pi in positions_block_a
    if move(pi) then output_signal(position_3) or
                     output_signal(position_4).

for all pi in positions_block_a
    if move(pi) then not output_signal(position_3) or
                     not output_signal(position_4).

for all pi in positions_block_b
    if move(pi) then output_signal(position_1) or
                     output_signal(position_2).

for all pi in positions_block_b
    if move(pi) then not output_signal(position_1) or
                     not output_signal(position_2).

* IV. conflict scenarios
* possible scenarios
if conflict(z1)
    then move(position_1) and
         move(position_3).

if conflict(z2)
    then move(position_1) and
         move(position_4).
```

```
if conflict(z3)
    then move(position_2) and
         move(position_3).

if conflict(z4)
    then move(position_2) and
         move(position_4).

* anyone of the conflict scenarios produces a crash
if crash then
    conflict(z1) or conflict(z2) or
    conflict(z3) or conflict(z4).

* last statement
ENDATA
```

CAUTION: The OR is inclusive, as is customary in logic.

leibnizparams.dat

Before compilation begins, lbcc reads the file leibnizparams.dat. That file specifies the name of the file containing the logic formulation and a number of options.

The parameter file consists of keywords, followed by the = sign and user-defined parameters, in free format. Below, the parameter file for the machine problem is listed as a typical example.

```
*** Parameter File for Leibniz Compiler lbcc
***
*** To effectively remove any statement below, place an
*** asterisk in column 1.  See manual for details.
*********************************************************
*** show steps on screen
show steps on screen
*********************************************************
*** Input file with logic file extension (see below)
input file = machine.log
*********************************************************
*** Input directory - specify complete path
input directory = ./
*********************************************************
*** File extensions:
logic file extension           (default: log) = log
transfer file extension        (default: trs) = trs
program file extension         (default: prg) = prg
error file extension           (default: err) = err
learned clauses file extension (default: lrn) = lrn
*********************************************************
*** Max size of formulations
colmax (max number of variables) = 500
rowmax (max number of clauses)   = 500
anzmax (max number of literals)  = 10000
blkmax (max number of blocks)    = 10
```

```
*********************************************************
*** Compile process options
compile process (normal, fast, very fast) = normal
use approximate minimization if time bound (sec) >= 5
*** to force approx.  min., specify 0 as time bound
*** to force exact min., comment out statement
keep all variables even if not used in any fact
compile for transfer process
make transfer files
transfer defs.h file = leibnizdefs.h
transfer exts.h file = leibnizexts.h
transfer code file   = leibniztrans.c
list warnings for nonuse of sets, predicates, variables
output compiled program to program file
*********************************************************
*** Learn option
* learn using file (filename or 'random') = random
max learning cycles = 100
output learned clauses
*********************************************************
*** do satisfiability test
do satisfiability test
*********************************************************
*** Machine speed for bound calculations
machine speed (mips) = 150
*********************************************************
*** Output of selected input portion in condensed format
*write condensed input into file = condensedfile
*begin condensed input keyword = FACTS
*end condensed input keyword = ENDATA
*********************************************************
ENDATA
```

If a line starts with an asterisk *, the compiler considers the line to be a comment and ignores it.

The line `show steps on screen` directs the compiler to output information about each step of the compile process. If this line is commented out, the compiler runs silently unless an error termination is encountered.

The line `input file = machine.log` indicates that the logic formulation in the file machine.log is to be processed.

The line `input directory = ./` declares that the input .log file is in the current directory. The compiler places all output files into the same directory.

The file extensions .log, .trs., .prg., .err, and .lrn may be modified by a suitable change of the right hand side of the lines

```
logic file extension           (default: log) = log
transfer file extension        (default: trs) = trs
program file extension         (default: prg) = prg
error file extension           (default: err) = err
learned clauses file extension (default: lrn) = lrn
```

The new extension names may contain up to thirty-two (32) characters.

The max size of formulations accepted by the compiler is specified by the lines

```
colmax (max number of variables) = 500
rowmax (max number of clauses)   = 500
anzmax (max number of literals)  = 10000
blkmax (max number of blocks)    = 10
```

Here, colmax is the max number of variables of a logic formulation, rowmax is the max number of clauses, anzmax is the max number occurrences of variables in the clauses, and blkmax is a decomposition parameter. Each one of the parameters must be at least 2.

CAUTION: The compiler always adds to each formulation at least one, and sometimes more than one, variable and clause to achieve a certain standard form. The user need not be concerned with that standard form, but should always select colmax, rowmax, and anzmax so that some additional variables and clauses can be accommodated. There is no simple prediction formula for the required increase, but an increase by 10 almost always suffices. A good range for blkmax is 10-20. If the selected value for blkmax too small, lbcc puts out a warning message during compilation saying that blkmax should be increased to improve the solution program.

CAUTION: Parameters colmax and rowmax cannot exceed 10000 each. If parameter rowmax is increased beyond 6001, parameter dermax in file exparms.h must be increased to the rowmax value.

The compiler allows for a number of processing options.

The line `compile process (normal, fast, very fast) = normal` allows for selection of one of three levels of quality for the solution program. If a problem is compiled and solved just once, the option `very fast` or `fast` may be a good choice. Otherwise, the option `normal` is appropriate.

In general, a logic formulations may be a satisfiability problem or a logic minimization problem. The machine problem presented earlier in this chapter is an example of a satisfiability problem. In the case of a minimization problem, the user may choose between two types of solution programs. In the first case, the user wants to have the problem solved to optimality, so that an exact solution is produced. In the second case, the user will accept a very good solution that need not be a truly optimal solution. Such a solution is called approximate. It seems appropriate that the choice between exact and approximate solutions be based on the solution time. That is, if the solution time for producing an exact solution is small, then the user obviously would desire such a solution. On the other hand, if computation time for an exact solution is excessive, than the user should be prepared to accept an approximate solution. Accordingly, the user should specify on the line

```
use approximate minimization if time bound (sec) >= 5
```

when a switch from exact to approximate solution is to be made. That is, if an integer n is specified on that line, then the compiler produces a solution program for an exact solution if the run time for that program does not exceed n seconds. On the other hand, the compiler produces a solution program for an approximate solution if an exact solution requires more than n seconds. The compiler makes the correct choice as follows. First, the compiler produces a solution program for finding an exact solution. Next, the compiler computes an upper bound on run time for that program, without actually executing the program. Finally, if that bound exceeds n, the compiler produces a second solution program that can find an approximate solution.

The compiler computes an upper bound on run time not only for exact minimization problems, but also for satisfiability problems. In the case of approximate minimization problem, the compiler computes a rough estimate for the run time of the generated program, as well as an exact bound on the time required to find a satisfiable solution. The upper bounds on run time and the rough estimate, if applicable, are displayed when the compiler is done. They are also included in the preamble of each generated program.

If the user wishes to enforce approximate minimization, then the time bound should be specified as 0, that is, the corresponding line in params.dat is

```
use approximate minimization if time bound (sec) >= 0
```

Such a choice is appropriate in cases where by experience it is already known that exact minimization requires unacceptably large run times. Specifying 0 as time bound in the params.dat file speeds up the generation process since then the compiler does not attempt generation of an exact solution program.

If the user wants to rule out approximate minimization, the line

```
use approximate minimization if time bound (sec) >= 5
```

should be commented out by an asterisk `*` as first character of that line.

The line `keep all variables even if not used in any fact` concerns the treatment of logic variables that do not occur in facts. The compiler keeps all such variables in the solution program unless the line is commented out. Reasons for keeping or not keeping these variables are discussed in Chapter 4 when the transfer process is described.

The line `compile for transfer process` instructs the compiler to produce the files for the transfer process. If the transfer process is not to be used, the line must be commented out.

The line `make transfer files` is also linked to the transfer process. If commented out, the making of transfer files is skipped even though the transfer process is used. This choice is useful when the user has already produced correct transfer files in previous compilations for a given logic formulation, and when another compilation is guaranteed to produce the same transfer files. The choice is also useful when the user has augmented the transfer files to accommodate a special situation, and when these transfer files are not to be changed by the compiler.

The names of the files for the transfer process are given by

```
transfer defs.h file = leibnizdefs.h
transfer exts.h file = leibnizexts.h
transfer code file   = leibniztrans.c
```

These names can be changed to handle special situations.

If a set, predicate, or variable is not used, the compiler issues a warning unless the line `list warnings for nonuse of sets, predicates, variables` is commented out.

The main output of lbcc is a .prg file that contains the solution program unless the line `output compiled program to program file` is commented out. There are cases when storage of the solution program in a .prg file is not needed, for example, when lbcc is executed from a user program using the procedure execute_lbcc and the solution program is run just once.

The compiler has a learning capability where it solves a number of cases using the solution program and learns how performance of the solution program can be improved. The line `learn using file (filename or 'random') = random` causes such learning to take place. The cases to be solved during the learning process are given in the specified file or may be randomly selected. The line accommodates both cases. Chapter 5 contains the details of the learning process. If learning is not desired, the line should be commented out.

The lines

```
max learning cycles = 100
output learned clauses
```

concern the learning process. Details about those lines are given in Chapter 5.

When the compiler has produced the solution program, the line `do satisfiability test` instructs it to run the solution program and check whether the given logic formulation is satisfiable. To suppress that test, the line must be commented out.

The line `machine speed (mips) = 150` is important for the computation of time bounds for solution programs. If the mips (million instructions per second) rating of the computer that is to run the solution program is not known, a reasonable alternate number is the MHz rating of the processor.

The line `write condensed input into file = condensedfile` tells that a portion of the input file is to be written out in condensed format into the specified file, here 'condensedfile'. All blank lines, all lines containing only comments, and all lines with INCLUDE statements are omitted. To skip writing to the condensed file, comment out the statement. The next two lines tell options for the writing to the condensed file.

The line `begin condensed input keyword = FACTS` tells that the record having the specified word as first word, here 'FACTS', triggers the beginning of the writing into the condensed file. If the line is commented out, writing starts with the first nonblank and noncomment record of the input file.

The line `end condensed input keyword = ENDATA` tells that the record having the specified word as first word, here 'ENDATA', stops the writing into the condensed file. Note that the record with the specified word is not written into the condensed file. If the line is commented out, writing to the condensed file continues until the entire input file has been processed.

Writing into the condensed file is useful when a number of INCLUDE statements have been used in the input file and if one desires just one file containing all or a portion of the input data in one file, without any blank or comment lines.

CAUTION: Do not use the first word of a comment record to trigger the beginning or end of writing into the condensed file, since such records are ignored by the writing process.

CAUTION: If the option telling the beginning or end record involves a special word such as FACTS or ENDATA, always use the upper case version even if the input file actually specifies the lower case version.

The last record of leibnizparams.dat must be `ENDATA`.

Logic Formulation – Overview

The .log file contains the logic formulation. The formulation contains the following sections.

SATISFY (or MINIMIZE)	Identify problem as a satisfiability or minimization problem.
⟨problem name⟩	State problem name.
SETS	Define sets, to be used for existential or universal quantification of predicates.
PREDICATES	Define predicates whose domains are some sets.
VARIABLES	Define propositional variables.
FACTS	State logic expressions that describe the system to be modeled.
ENDATA	Identify end of data.

Reserved Words and Special Symbols

With one exception, reserved words can be stated in all upper case or all lower case. The exception is the word GOAL, which cannot be used in lower case. In the example file machine.log, a mixture of the two types is used to improve readability. In the compiler, all lower case reserved words are first converted to upper case. Thus, all error messages involving records of the .log file display reserved words in upper case, as do the examples given in the remainder of this chapter. That decision has the beneficial effect that, in this manual, reserved words stand out without use of different type style, quotes, or other methods of emphasis.

Reserved words may only be used as described below. Of course, the lower case versions must observe the same rules.

Reserved Word or Symbol	Meaning
( )	Used only to enclose arguments of predicate.
[ ]	Used only for logical grouping.
, (comma)	Used to separate arguments in predicate.
. (period)	End of fact statement
=	Assignment
*	Indicates that the remaining part of the line contains comments and is to be ignored by the compiler. If an * is placed into column 1, then the entire line is ignored.
- (hyphen)	Negation
&	AND of logic
|	OR of logic
==	"equal" in WHERE statement
!=	"not equal" in WHERE statement
$<$	"less than" in WHERE statement
$\leq$	"less than or equal" in WHERE statement
$>$	"greater than" in WHERE statement
$\geq$	"greater than or equal" in WHERE statement
AND	AND of logic
ASSIGNFALSE	Indicates that the default value for the specified predicate variable or propositional variable is False. The default value can be replaced by True at execution time.
ASSIGNTRUE	Indicates that the default value for the specified predicate variable or propositional variable is True. The default value can be replaced by False at execution time.

AT LEVEL n	Statement holds with a likelihood of about $n\%$ ($1 \leq n \leq 100$).
DEFINE	Definition of set or predicate
DELETABLE	Indicates that the specified predicate variable or propositional variable is deletable at execution time.
ENDATA	Last statement (end of input)
FACTS	Denotes beginning of fact declarations.
FALSECOST	Used to assign cost of False to a predicate variable or to a propositional variable.
FC, FCOST	Abbreviated forms of FALSECOST
FOR ALL	Universal quantification
FREQUENTLY	Statement holds with a likelihood of 65%.
GOAL	Used to define goals. This is the only reserved word that cannot be used in lower case.
HALF THE TIME	Statement holds with a likelihood of 50%.
IF .. THEN	IF .. THEN statement
IN	Used in quantification definition.
INCLUDE	For inclusion of a file. Must be followed by the name of the file.
ON	Used in predicate definition.
OR	OR of logic
MINIMIZE	Problem is to be solved for satisfying solution with minimum total cost.
NAME	Indicates that a clause name follows next. The named fact is deletable at execution time.
NOT	Negation
PREDICATES	Denotes beginning of predicate declarations.

RARELY	Statement holds with a likelihood of 10%.
SATISFY	Problem is to be solved for satisfiability.
SETS	Denotes beginning of set declarations.
SOMETIMES	Statement holds with a likelihood of 25%.
THERE EXISTS	Existential quantification
TRUECOST	Used to assign cost of True to a predicate variable or to a propositional variable.
TC, TCOST	Abbreviated forms of TRUECOST
UNIVERSE	Special set name
VARIABLES	Denotes beginning of variable declarations.
VERY FREQUENTLY	Statement holds with a likelihood of 85%.
WHERE	Specifies a restriction on quantification for the associated fact.

CAUTION: The OR is inclusive, as is customary in logic. For example, the statement `x OR y` means that at least one of x and y must be True.

Tab characters and most special characters used by word processors are considered to be blanks by lbcc. Specifically, any ASCII characters whose decimal equivalent falls into the range 1-31 or 176-255 is considered to be a blank. Thus, the .log file can be prepared with almost any word processor or text editor.

User-defined Names

User-defined names must obey the following rules:

Only alphanumeric characters (upper or lower case) and underscore ("_") are permitted. The first character must be a letter.

At most fifty-two (52) characters per name. This limit also applies to predicate variables, and thus includes predicate name, parentheses, comma, and arguments.

Embedded blanks are not allowed since blanks are treated as delimiters.

INCLUDE Statement

The statement `INCLUDE` ⟨filename⟩ in a .log file causes the compiler to insert the file with the specified filename. An included file may itself include other files, which in turn may include files, and so on. Such nesting can be done thirty-two (32) files deep.

The INCLUDE command is particularly useful when .log files use some common sets, predicates, variables, or facts. Common portions should then be stored in separate files and included as needed. That way, a change in an included file produces a correct update of all affected .log files.

Comments and Blank Lines

Comments are indicated by the `*` character. The `*` may occur on any line, in any position. All entries to the right of the leftmost `*` of a line are ignored by the compiler. Comments may contain any symbol or character since they are not evaluated by the compiler.

Any blank line is ignored.

Declaration Sections

The declaration sections must adhere to the following rules:

The declaration sections must appear in the input file in the same order in which they are described below.

Either both the SETS and PREDICATES sections must appear, or both must not.

The VARIABLES section is optional.

The FACTS section is mandatory.

If a declaration section is included at all, then it must be non-empty.

Indentation shown in the examples below is for the purpose of readability. Its use is strongly recommended, but not required. Spaces are treated as delimiters, so at least one or more blanks must separate keywords and names from surrounding text.

Changing Formulations During Execution

If one anticipates that the formulation will be expanded during execution, one should insert into the appropriate sections of the .log file INCLUDE statements that initially specify empty files. For example, if one expects to add logic statements to the formulation, one inserts into the FACTS section the statement `INCLUDE` ⟨fact_filename⟩, where fact_filename is an empty file. When during execution facts are to be added to the logic formulation, the appropriate statements are inserted in that file, and the .log file is recompiled with the procedure execute_lbcc(). For details of that procedure, see Chapter 4.

CAUTION: If the above approach adds a set, predicate, or variable to the .log file, and if the transfer process is used during execution, the C code produced by the lbcc compiler from the .log file must be compiled and linked with the user C program. Thus, the user must interrupt execution to carry out these steps.

SETS

Sets specified here are used as domains for predicates.

CAUTION: If the SETS section is included, then it must be followed by the PREDICATES section, described below.

The reserved word SETS marks the beginning of the section.

A set is defined by the reserved word DEFINE followed by the set name. Each element name of the set begins on a new line. The last member of a set is followed by either another DEFINE (for another set) or by the PREDICATES declaration section. Both set and element names may be any legal name.

Generally:

```
SETS
  DEFINE ⟨set 1⟩
    ⟨element 1⟩
      .
      .
    ⟨element n⟩
  DEFINE ⟨set 2⟩
      .
      .
```

Example:

```
SETS
  DEFINE positions
    position_1
    position_2
    position_3
    position_4

  DEFINE scenarios
    z1
    z2
```

```
z3
z4
```

There is a set called UNIVERSE that has a special feature. In contrast to any other set, the set UNIVERSE need not be used in the definition of any predicate, and yet the compiler will not produce any warning message. The reason for this special set is as follows. The compiler stores all element names consecutively, in the order in which they are encountered when processing the definitions of the various sets. That order typically is unimportant, but may be important when the transfer process is employed and when the user program loops through predicate variables. To control the sequencing of elements, and thus to control the loop parameters in a predictable way, one can specify the set UNIVERSE as the first set in the section SETS and define in that set all element names in the desired order. That order is then maintained when the user program loops through the arguments of any predicate.

Subsets

A given element may occur in any number of sets. A given set may intersect other sets or may be a subset of other sets.

File z1to1000

The subdirectory Makedata of the *Leibniz System* installation contains a file z1to1000 with 1000 records that list z1, z2, and so on, up to z1000. The user can extract portions of that file as needed to define sets. When such extracted portions are large, they should be included in the .log file using the INCLUDE statement to unclutter the .log file.

Restrictions

There may be at most five hundred (500) sets defined in any given problem.

There may be at most five thousand (5,000) elements in any given set.

There may be at most five thousand (5,000) element names in the union of all declared sets.

Empty sets are not allowed, so at least one element must be specified for each set.

PREDICATES

A predicate is a True/False function defined on one or two sets, which are the domain of the predicate. These sets must have been defined in the SETS section.

CAUTION: If the PREDICATES section is present, then it must be preceded by the SETS section.

The reserved word PREDICATES marks the beginning of the predicate section.

A predicate is defined by the reserved word DEFINE followed by the predicate name. The predicate name is followed by the keyword ON, which in turn is followed by the name(s) of the one or two sets defining the domain(s). If two sets are given, then the symbol X (capital letter X, denoting cross product) is used to separate the two set names.

Options (described below) may be included with the definition of each predicate.

Generally:

```
PREDICATES
  DEFINE ⟨predicate⟩ ON ⟨set⟩ {options}
  or
  DEFINE ⟨predicate⟩ ON ⟨set 1⟩ X ⟨set 2⟩ {options}
```

Example:

Assume the SETS declaration example given earlier, which defined the two sets “positions” and “scenarios”.

```
PREDICATES
  DEFINE lever_switch on positions
  DEFINE conflict on scenarios
```

The statements define the predicates lever_switch and conflict. The predicate lever_switch is defined on the set positions, and the predicate conflict is defined on the set scenarios.

With each predicate, several options can be specified.

Options – Overview

A predicate variable or predicate instance is the propositional variable derived from a predicate by fixing its arguments to elements of the domain. The general form of a predicate variable is

predicate(element) {options}
or
predicate(element 1,element 2) {options}

Any of five variable options may be specified for an individual predicate variable, or for all predicate variables arising from a predicate. The options are:

DELETABLE	A variable with this option may be deleted during execution time.
TRUECOST = ⟨value⟩	The specified cost is associated with the assignment of True to the variable.
FALSECOST = ⟨value⟩	The specified cost is associated with the assignment of False to the variable.
ASSIGNTRUE	A default value of True is assigned to the variable. This value can be replaced by False at execution time.
ASSIGNFALSE	A default value of False is assigned to the variable. This value can be replaced by True at execution time.

The options may be specified in any order.

CAUTION: ASSIGNTRUE and ASSIGNFALSE may not appear together in the option specification for a given predicate or predicate variable.

The delete option (DELETABLE) indicates that the given variable may be deleted from the problem at execution time. Deletion amounts to effectively removing the variable from the system, as if it were not there. Indicating that a variable may be deleted does not in and of itself cause the variable to be deleted. The actual deletion must be specified at execution time. The use of the delete option may degrade execution time performance, so it is recommended that it only be used when needed.

A cost value is an integer in the range −16383 to 16383. The cost of True is specified by TRUECOST = ⟨integer⟩; the cost of False by FALSECOST=⟨integer⟩. The default value for the cost of True or False is 0. Cost assignments are effective only when a satisfying solution of minimal total cost is desired (keyword MINIMIZE), and are ignored when only a satisfying solution is desired (keyword SATISFY).

The assign variable option (ASSIGNTRUE, ASSIGNFALSE) sets the associated variable to the specified default value.

Options – Rules

In the PREDICATES section, a particular set of variable options may be defined for all predicate variables in the range of the associated predicate, or for individual predicate variables.

The general form for specifying variable options for all predicate variables in the range of a predicate is

`DEFINE` ⟨predicate⟩ `ON` ⟨set⟩ {options}
or
`DEFINE` ⟨predicate⟩ `ON` ⟨set 1⟩ `X` ⟨set 2⟩ {options}

Typical use:

```
PREDICATES
  DEFINE prd1 ON set1 DELETABLE
  DEFINE prd2 ON set1 X set2 ASSIGNTRUE DELETABLE
```

All predicate variables of a predicate inherit the specified options. For an individual predicate variable, these options may be overridden in the following manner. Immediately following the line with the predicate definition (DEFINE...), the predicate variable is listed on the next line, together with its options. Any number of such predicate variables may be so listed, each one with its options on a separate line. For each such predicate variable, at least one option must be specified. If no option at all is desired for a predicate variable, one should specify TRUECOST=0, which imposes the default cost 0 for the assignment of True.

CAUTION: The options specified with a predicate variable override all options listed with the predicate in the DEFINE statement.

For example, given set1 = {a,b,c,d},

```
DEFINE prd1 ON set1 DELETABLE ASSIGNTRUE TRUECOST=5
  prd1(a) FALSECOST=2
  prd1(d) ASSIGNFALSE FALSECOST=12
```

specifies that prd1(b) and prd1(c) have the options DELETABLE, ASSIGNTRUE, TRUECOST=5, while prd1(a) has only the option FALSECOST=2, and prd1(d) has only the options ASSIGNFALSE, FALSECOST=12.

A predicate variable may be specified only once. Additional specifications for the same predicate variable trigger a warning message by the compiler and are then ignored.

The general form for specifying variable options to an individual predicate variable is:

predicate(element) {options}
or
predicate(element 1,element 2) {options}

Typical use:

```
PREDICATES
  DEFINE prd1 ON set1
    predicate1(elt1) DELETABLE TRUECOST = 100
    prd1(elt2) ASSIGNFALSE
    prd1(elt3) DELETABLE
```

Restrictions

There may be at most five thousand (5,000) predicates defined.

VARIABLES

The term “variables” is used here to denote propositional variables.

The reserved word VARIABLES marks the beginning of the section for propositional variables.

A propositional variable is defined by simply giving its name. Any legal name be used. Each propositional variable must appear on a separate line.

Generally:

```
VARIABLES
  ⟨variable 1⟩ {options}
       .
       .
  ⟨variable n⟩ {options}
```

Typical use:

```
VARIABLES
  crash
```

Options

All options for predicate variables may be invoked for propositional variables. See the PREDICATES section for a complete description of these options.

Generally:

```
VARIABLES
  ⟨variable name⟩ {options}
```

Typical use:

```
VARIABLES
  x
  y DELETABLE TRUECOST = 304 FALSECOST = 32
```

```
z TRUECOST = 10 ASSIGNFALSE
w DELETABLE
v FALSECOST = 23 DELETABLE
```

Restrictions

There may be at most eight thousand (8,000) propositional variables defined.

FACTS

A fact is a logic expression. It consists of a single logic statement which may be quantified and may involve likelihoods. A fact consists of quantification (optional), quantification conditions (optional), and the fact body. The fact body may involve likelihood terms. A fact can be named and thus becomes deletable.

The reserved word FACTS marks the beginning of the facts section.

In general, a fact statement may consist of up to three parts, as follows:

{quantification part, optional}
{quantification conditions, optional}
{the fact, possibly with likelihoods or fact name}

A fact statement may span any number of lines and is terminated by a period.

CAUTION: The record length for any one line may be at most one hundred twenty-five (125) characters, including blanks. For readability, it is strongly recommended that each line be limited to 80 characters or less, including blanks.

Fact Forms

Several forms of facts are allowed. Below, they are labeled (A) through (F). In the statement of these cases below, each x_i, y_i, z_i is a literal, i. e., it is the occurrence of a possibly negated predicate variable or propositional variable. Each exp_i is also such an occurrence, or is an expression of the form [x_1 AND .. AND x_k] or NOT [x_1 AND .. AND x_k]. Below, $m \geq 1, n \geq 1$.

(A) x_1 OR .. OR x_m.
(B) IF x_1 AND .. AND x_m THEN y_1 OR .. OR y_n.
(C) IF x_1 AND ... AND x_m THEN y_1 AND .. AND y_n.
(D) IF x_1 OR ... OR x_m THEN y_1 OR .. OR y_n.

(E) `IF` x_1 `OR ... OR` x_m `THEN` y_1 `AND .. AND` y_n.
(F) exp_1 `OR` exp_2 `.. OR` exp_n.

Remark: (A) is a conjunctive normal form (CNF) clause, (B) through (E) are various IF .. THEN forms, and (F) is a disjunctive normal form (DNF) formula.

CAUTION: The rules imply the following. The brackets [and] may not be used in conjunction with any IF .. THEN statement. The connectives between IF and THEN must be either all AND or all OR, as must be the connectives after THEN. The connectives occurring between the brackets [and] must be all AND, and the connectives outside [..] must be all OR. Nested brackets are not allowed.

Here are some examples.

```
(A) r OR s OR NOT v OR w.
(B) IF NOT r AND s THEN v OR NOT w.
(C) IF NOT r AND s THEN v AND NOT w.
(D) IF NOT r OR s THEN v OR NOT w.
(E) IF NOT r OR s THEN v AND NOT w.
(F) [NOT r AND s] OR [NOT v AND w].
    NOT [NOT r AND s] OR [NOT v AND w].
    [NOT r AND s].
    NOT [NOT r AND s].
    NOT [r] OR [NOT s].
    NOT [r].
    [r].
```

The last three examples for (F) are legal but unnecessarily cumbersome since `NOT r OR NOT s`, `NOT r`, and `r`, respectively, are equivalent simpler forms of type (A).

Quantification

Quantification is allowed under the following restrictions.

At most three (3) quantified variables may be used.

The quantification of each variable must be of the same type, i.e., all universal, or all existential.

Quantification is permitted only if the fact body is of the form (A) or (B) given above.

Universal quantification is specified as follows:

`FOR ALL i IN` ⟨set⟩

Note that the variable i may be any legal name. This also applies to existential quantification, which is specified as follows:

`THERE EXISTS i IN` ⟨set⟩

For a given fact, each quantification variable must occur in exactly one quantification statement (FOR ALL or THERE EXISTS) of that fact.

<u>CAUTION</u>: The quantification variables are not separately declared, since each such variable is a local variable that is only valid for the single fact following the quantification statement.

The set specified for a quantification variable in the FOR ALL or THERE EXISTS statement must be a subset of the domain sets of the predicates that in the fact body are listed with that variable.

Here are some examples from machine.log.

```
for all pi in positions
    if not lever_switch(pi) and switch_okay(pi)
        then not output_signal(pi).

for all pi in positions_block_b
    if move(pi) then output_signal(position_1) or
                     output_signal(position_2).
```

A note on indentation: It is suggested that, when using quantification, the associated body of the fact be written indented and be followed by at least one blank line, with `*` in column 1 if so desired. This convention greatly aids readability.

Conditional Quantification

Constraints may be placed on quantification through the use of the WHERE keyword. A WHERE condition is of the form

`WHERE` ⟨variable 1⟩ ⟨relation⟩ ⟨variable 2⟩

⟨variable 1⟩ and ⟨variable 2⟩ are two of the quantification variables, and ⟨relation⟩ is one of the following relations:

==	equal
!=	not equal
$<$	less than
$\leq$	less than or equal
$>$	greater than
$\geq$	greater than or equal

The constraint is satisfied if the relation holds when variable 1 is lexicographically compared with variable 2 as done in the C procedure strcmp(). For example, suppose the relation is $<$. That relation holds if, starting from the left, the first difference between variable 1 and variable 2 is negative, using the ASCII values for comparison. This means, for example, that “A” is less than “a”.

The WHERE conditions immediately follow the quantification part of a fact statement and precede the fact body. Each WHERE condition is on a separate line.

Typical use:

```
FOR ALL i in set1
FOR ALL j in set2
  WHERE i < j
    IF prd1(i,j) THEN prd2(j).
```

The above fact statement limits the quantification to values of i and j where i is less than j according to the ASCII decimal equivalent values for i and j. Suppose set1 = {a,b}, and set2 = {b,d}. Then the above fact is equivalent to the following three statements:

```
prd1(a,b) OR prd2(b).
prd1(a,d) OR prd2(d).
prd1(b,d) OR prd2(d).
```

Note that prd1(b,b) OR prd2(b) is not listed since b is not less than b.

CAUTION: At most one WHERE condition may be specified for a given pair of quantification variables. This implies that a fact with two quantification variables may have at most one WHERE condition, and that a fact with three quantification variables may have at most three (3) WHERE conditions.

Likelihood Terms and Levels

Likelihood terms, or, more generally, likelihood levels, are used to indicate varying degrees of uncertainty for logic statements. They permit the inclusion of vague or imprecise statements. Here are some examples.

```
IF x THEN y OR z FREQUENTLY.
x OR NOT y OR z AT LEVEL 90.
```

According to the first statement, it frequently holds that, if x, then y or z. The second statement says that x or not y or z, with a likelihood of 90%.

Likelihoods terms or likelihood levels may be specified for any fact immediately before the terminating period.

Here are the available likelihoods terms, along with a percentage that approximately describes the frequency of occurrence for that likelihood.

VERY FREQUENTLY	likelihood is 85%
FREQUENTLY	likelihood is 65%
HALF THE TIME	likelihood is 50%
SOMETIMES	likelihood is 25%
RARELY	likelihood is 10%
AT LEVEL n	likelihood is n% $(1 \leq n \leq 100)$

For the manipulation of likelihood levels at execution time, see procedure modify_clause of Chapter 4.

If no likelihood is assigned to a fact, then the fact is considered to be always valid, as one would expect. Formally, the compiler assigns to such a fact a likelihood level of 0. This convention is of no consequence during compilation, but is important at execution time.

CAUTION: A likelihood level of 0 does not imply that the associated fact is unlikely to be valid. It simply indicates that the user has not explicitly assigned a likelihood level.

Clause Name and Clause Deletion

In the input, a logic statement is termed a fact. The compiler converts the input to CNF (conjunctive normal form) format, where a logic statement is termed a clause. In general, a single fact may produce several clauses.

One may name facts as follows. After the period of the fact statement, one either adds the keyword NAME followed by the name of the fact, or adds just the name of the fact. The word(s) may immediately follow the period or be separated from it by any number of blanks. However, the period and the word(s) must be on the same line.

The name given to a fact is assigned to all clauses arising from the fact. Named facts, and thus named clauses, may be deleted at execution time, by referring to the name. Such a deletion effectively removes the clauses from the formulation, as if they were not present.

One may specify the same name for several facts. When at execution time the deletion of clauses is specified using that name, all clauses with that name are simultaneously deleted. This feature simplifies the deletion of sets of clauses. The deletion of named clauses is further simplified by use of the wildcard character “?” at execution time. For details, see procedure modify_clause of Chapter 4.

CAUTION: Whenever a fact is named, the clauses of that fact are deletable at execution time. Making clauses deletable may affect execution time performance. Hence, one should not name facts unless clause deletion is intended at execution time. If one only wants to name a fact for easy reference in the .log file, one may add * ⟨fact name⟩ after the period of the fact statement. The added * ⟨fact name⟩ is considered to be a comment by the compiler and does not cause the fact to be named and become deletable.

Here are some examples.

```
IF x THEN y.  NAME rule1
IF x THEN y.  rule1
FOR ALL i IN set1
  IF prd1(i) THEN prd2(i).  rule3
```

CAUTION: In subsequent examples, the keyword NAME is always omitted.

Likelihood Terms and Clause Deletion

When a fact contains a likelihood term, then the clauses of that fact are considered deletable even if the fact is not named. In that case, deletion is possible only by specification of the given likelihood value.

CAUTION: A clause becomes deletable by one of two ways: By being named, or by assignment of a likelihood term.

See procedure modify_clause of Chapter 4 about the various ways in which deletions may be accomplished, using the fact name or likelihood level.

Goals and GOAL Variable

In certain logic formulations involving minimization, one may want to include goals. For example, one may desire that a given resource be used up, or that usage of a resource observe a specified upper limit. Such goals can be included in the logic formulation if each logic variable occurs in at most one such goal, and if the logic formulation is solved by approximate minimization. The earlier section discussing the leibnizparams.dat file explains how approximate minimization is specified.

An example best explains how goals are handled. Suppose three logic variables x, y, and z represent the purchase of three items X, Y, and Z. For example, variable x is True if item X is purchased. Suppose the cost for X is $p(X) = \$10$, for Y it is $p(Y) = \$13$, and for Z it is $p(Z) = \$11$. Suppose we have a budget limit of \$21.

Of course, we could figure out which combinations of the three items may be purchased under the budget limit. We then could include the logic statements describing these combinations in the FACTS section. This approach would work here, but might be very cumbersome if we had, say, ten items instead of just three.

Mathematically, the budget restriction is

$$\sum p(I) \leq 21$$

where the summation is over the items I that are selected for purchase. The budget constraint is formulated in two steps. In the .log file, one includes in the VARIABLE section a new logic variable GOAL, with the option ASSIGNTRUE, and in the FACTS section one adds a fact `GOAL. budget.`

<u>CAUTION</u>: The variable name GOAL cannot be replace by its lower-case version. This is the only case where a keyword does not a lower-case equivalent version.

Note that the variable must have the name GOAL, while the name budget of the fact `GOAL. budget` may be replaced by any chosen name observing the naming rules for facts. The fact statement `GOAL. budget` causes the compiler to reserve space for the budget restriction

$$\sum p(I) \leq 21$$

The data for the budget restriction are not included in the .log file, but are inputted at execution time via procedures modify_goal and modify_variable. Specifically, with procedure modify_goal one specifies the budget limit of $21 as well as a penalty cost for exceeding that limit. Procedure modify_variable is used to assign the costs of the items, that is, $p(X)$ =$10, $p(Y)$ =$13, and $p(Z)$ =$11, to the logic variables x, y, and z, and to declare that these costs are applicable to the goal named budget. For details about procedures modify_goal and modify_variable, see Chapter 4.

One may include more than one goal in the FACTS section as long as each logic variable of the logic formulation is involved in at most one of the goals. For example, if one has three goals named budget, space, and time, then one adds the GOAL variable just one time to the VARIABLE section as before, but adds to the FACTS section the three statements `GOAL. budget`, `GOAL. space`, and `GOAL. time`. During execution of the solution program, one then adds the appropriate goal data using procedures modify_goal and modify_variable analogously to the situation described above.

In agreement with the general rules for facts, each goal fact included in the logic formulation is assigned a likelihood level of 0, that is, the goal is considered to be always valid. One may delete goal facts at execution time using the same procedure as for the deletion of any other fact. For details about such deletions, see procedure modify_goal of Chapter 4.

CAUTION: The variable GOAL may only be used in facts representing goals, and that variable must be declared with the ASSIGNTRUE option and no other option. However, the fact names of goals may also be used for facts of logic statements. The latter feature is useful when one wants to simultaneously delete or activate goals as well logic statements.

Error Messages

The compiler displays warning and error messages with the line number of the input .log file or of the included file causing the message, plus an error code prefixed by “LBCC”. This section contains detailed explanations for each such code. There are three types of warning/error messages:

Warning	Indicates that an unexpected entry was encountered in the .log file. Compilation is continued after the display of the message, and output files are generated.
Nonfatal Error	Indicates incorrect syntax in the .log file. Processing proceeds with the next statement following the error, but no program is generated.
Fatal Error	Indicates incorrect syntax in the .log file. The error is such that processing cannot continue.

Whenever a warning or error is encountered, it is displayed on the screen if that has been specified in the parameter file leibnizparams.dat. The warning or error is also recorded in a file with same name as the input .log file except that the extension .err is used. For example, when the file machine.log is processed, warnings and errors are recorded in the file machine.err.

If the compiler produces not warnings or errors, the .err file contains at the end of compilation the records

```
**********************************
Total Warnings = 0
Total Errors = 0
Processing of .log file completed
**********************************
```

Warnings

LBCC0010: Variable with cost not used: ⟨name⟩

A predicate or propositional variable was specified with cost but was not used in the FACTS section. If the variable should have been used, check the FACTS section for correctness. If the FACTS section is correct, then the predicate variable or propositional variable causing the warning may be commented out or removed. The variable may not have appeared either because it was never used in a FACT statement, or because all clauses containing that variable were removed from the system as always satisfied.

LBCC0020: Variable with delete option not used: ⟨name⟩

A predicate or propositional variable was specified with the delete option but was not used in the FACTS section. If the variable should have been used, check the FACTS section for correctness. If the FACTS section is correct, then the predicate variable or propositional variable causing the warning may be commented out or removed. The variable may not have appeared either because it was never used in a FACT statement, or because all clauses containing that variable were removed from the system as always satisfied.

LBCC0025: Variable with assigned value not used: ⟨name⟩

A predicate or propositional variable was specified with ASSIGNTRUE or ASSIGNFALSE but was not used in the FACTS section. If the variable should have been used, check the FACTS section for correctness. If the FACTS section is correct, then the predicate variable or propositional variable causing the warning may be commented out or removed. The variable may not have appeared either because it was never used in a FACT statement, or because all clauses containing that variable were removed from the system as always satisfied.

LBCC0030: Set never used: ⟨name⟩

A set with name ⟨name⟩ was specified in the SETS section but was not referenced in any predicate definition.

Check the PREDICATES definitions for correctness. If the PREDICATES section is correct, the set declaration causing the warning may be commented out or removed.

LBCC0040: Predicate never used: ⟨name⟩

A predicate with name ⟨name⟩ was specified in the PREDICATES section but was not referenced in the FACTS section. Check the FACTS section for correctness.

LBCC0050: Variable never used: ⟨name⟩

A propositional variable with name ⟨name⟩ was specified in the VARIABLES section but was not referenced in the FACTS section. Check the FACTS section for correctness. If the FACTS section is correct, the variable causing the warning may be commented out or removed.

LBCC0060: Redefinition of variable options for ⟨name⟩ ignored

The named predicate variable was specified more than once with variable options. A given predicate variable may appear only once with options. Any subsequent specifications are ignored. Check which specification is correct, and remove the others to eliminate this warning.

LBCC0070: Fact statement is always satisfiable

A clause with a variable appearing both negated and non-negated is always satisfied and hence is not included in the FACTS section. If for some fact statement, all resulting clauses are removed, then this message is displayed. Remove or comment out the fact statement to eliminate this warning.

LBCC0080: Extra characters on this line are ignored

Additional characters were encountered on a line and were not processed by the compiler. As typical example, two element names appearing on the same line cause this message. In that case, the compiler expects only one name per line, and thus declares the second name as additional, unexpected characters. The message is also produced when a period "." occurs that is not the terminating period of a fact or goal.

Nonfatal Errors

LBCC1010: ⟨name⟩ has already been defined as a(n) ⟨type⟩

A set, element, predicate, variable, quantifier name, or clause name has already been used. The message indicates how the name was used ⟨type⟩, where ⟨type⟩ is one of the following:
– element
– set
– predicate
– variable
– quantifier
To correct the input, select a new name for ⟨name⟩.

LBCC1020: Empty set not allowed

A set name was defined with no elements.

LBCC1030: Too many elements in set; at most ⟨N⟩ allowed

A maximum of ⟨N⟩ elements are allowed as members of a set. This maximum was violated. See section above on SETS for parameters of set definitions.

LBCC1040: Element already a member of this set

The element name on the given line number is a duplicate for the set being processed.

LBCC1050: Unrecognized set name: ⟨name⟩

Either the set name given as a domain for a predicate was not recognized, or the set name given as the domain of quantification was not recognized.

LBCC1060: Too many sets defined for predicate

At most two arguments are allowed for any predicate, and this restriction was violated.

LBCC1080: Predicate name mismatch in variable option specification

The predicate name given in the variable option part of a predicate definition did not match that of the predicate name in the definition.

LBCC1090: “(” expected; got: ⟨string⟩

The “(” character is required between the predicate name and its first argument. ⟨string⟩ represents the syntactic component which was encountered.

LBCC1100: Incorrect number of arguments in variable option specification

The number of arguments for a predicate variable given in the variable option part of a predicate definition did not match that of the predicate definition itself.

LBCC1110: Element ⟨name⟩ is not in set ⟨name⟩

The given predicate variable in the variable options part of a predicate definition contained an element name as an argument that is not in the domain of the original definition. The element name causing the error is given, as well as the set name defined as the domain for that argument in the original predicate definition.

LBCC1120: Unrecognized element name: ⟨name⟩

The element name ⟨name⟩ given in the variable options part of a predicate definition was not recognized.

LBCC1130: “)” or “,” expected; got: ⟨string⟩

Either the “)” or “,” characters were expected, but ⟨string⟩ was encountered.

LBCC1140: VARIABLES section but no variables defined

The VARIABLES keyword was encountered, but no variables were defined.

LBCC1150: Variable options intended; none given

A predicate variable was given in the variable option part of a predicate definition, but no options accompanied it.

If a predicate variable without any options is desired, and if some options have been specified for the predicate, then one should use the option TRUECOST=0 for the predicate variable. That option sets the TRUECOST to the default value of 0, and thus achieves the desired result.

LBCC1160: Duplicate TRUECOST (FALSECOST) definition

The TRUECOST or FALSECOST keyword was encountered twice in a variable options specification.

LBCC1170: Variable option keyword expected; got: ⟨string⟩

A variable option keyword (e.g., TRUECOST, FALSECOST, DELETABLE) was expected, but ⟨string⟩ was encountered instead.

LBCC1180: "=" symbol expected

The assignment operator was expected, but was not encountered. Check the variable options portion of a declaration for correctness.

LBCC1190: ⟨name⟩ is an illegal identifier: ⟨indication⟩

The given name ⟨name⟩ is not a legal identifier. ⟨indication⟩ gives the reason, where ⟨indication⟩ is one of the following:
– reserved word
– too long
– illegal characters
Check the rules for name construction.

LBCC1200: Cost value expected

An integer cost value was expected, but was not given.

LBCC1210: Five digits maximum for cost value

A cost value may be at most five digits long. This limit was exceeded.

LBCC1220: Cost must be integer: -16383 .. 16383

Correct the cost value so that it falls within the specified range.

LBCC1230: Invalid number: non-numeric characters

A cost value must consist of an optional "-" sign, and digits.

LBCC1240: Cannot mix quantifiers

The quantification part of a fact statement may consist only of quantification of the same type; either all FOR ALL, or all THERE EXISTS.

LBCC1250: Keyword ALL (EXISTS) expected.

The indicated keyword was expected but was not encountered.

LBCC1260: Quantifier already used: ⟨name⟩

Quantified variables may be the same for different fact statements, but must be unique within the same statement. The same name was used for two or more quantifiers in the same statement.

LBCC1270: Too many quantified variables; at most: ⟨N⟩

⟨N⟩ represents the largest number of quantifiers that may be used in a given fact statement. This number was exceeded.

LBCC1280: Keyword IN expected

The keyword "IN" was expected, but not encountered.

LBCC1290: Unrecognized predicate or variable: ⟨name⟩

⟨name⟩ was not recognized as a previously defined predicate or variable. Check for typos and syntax correctness. This message is also displayed when a likelihood keyword appears anywhere in a fact statement except immediately after the keyword THEN.

LBCC1300: Element ⟨name 1⟩ is not in set ⟨name 2⟩ of predicate: ⟨name 3⟩

The element ⟨name 1⟩ given as an argument in the predicate ⟨name 3⟩ in a fact statement is not a member of the set domain ⟨name 2⟩ defined for that argument of the predicate.

LBCC1310: Quantifier set and predicate argument set mismatch: ⟨name⟩

The set specified for a quantifier in the quantification part of a fact statement is not a subset of the set domain defined for the appropriate argument of the given predicate ⟨name⟩.

LBCC1320: Element or quantifier expected with ⟨name⟩; got: ⟨string⟩

Predicate ⟨name⟩ requires at least one argument, which is either an element or a quantifier. ⟨string⟩ was encountered instead.

LBCC1330: Incorrect number of arguments with predicate ⟨name⟩

A predicate given in the fact statement did not have the same number of arguments as its definition.

LBCC1340: Incorrect use of AND operator

The AND operator is allowed only within brackets [ ] and after IF or THEN.

LBCC1350: OR operator not allowed within [ ]

The OR operator was encountered within a bracketed term [ ]. This is not allowed.

LBCC1360: Too many "]" encountered.

Correct the mistake.

LBCC1370: Keyword IF missing.

The keyword THEN was encountered without an associated IF.

LBCC1380: THEN already occurred

The keyword THEN may only appear once.

LBCC1390: Quantifier not used: ⟨name⟩

A quantifier was specified in the quantification part of a fact statement but was never referenced in the associated fact statement.

LBCC1400: THEN is missing

The IF part of an IF .. THEN was given, but the THEN part was not encountered.

LBCC1410: Syntax error: unmatched brackets

Left and right brackets do not match.

LBCC1420: Clause name required after NAME

NAME was specified for a clause, but the name was not given.

LBCC1440: Operator expected; got: ⟨string⟩

A logic operator was expected in a fact statement, but ⟨string⟩ was encountered instead.

LBCC1450: Unrecognized likelihood

An unknown likelihood was given. Refer to the documentation for a complete list of available likelihoods.

LBCC1460: Statement must be a legal form

Refer to the FACTS section for a list of legal statement forms.

LBCC1470: Illegal statement for quantification

One of the legal forms was used for a fact statement, but it was not a form that is compatible with quantification.

LBCC1480: WHERE condition variable not recognized

A condition variable on either side of the binary WHERE relation was not recognized. A condition variable must be one of the quantified variables in the associated quantification statements.

LBCC1490: Unrecognized WHERE relation

Refer to the FACTS section regarding WHERE conditions for a list of recognized relations.

LBCC1500: Too many WHERE conditions

At most one WHERE condition for two quantification variables, and at most three WHERE conditions for three quantification variables, are allowed.

LBCC1510: Too many WHERE conditions for given quantifier pair

At most one WHERE condition may be specified for any given pair of quantification variables.

LBCC1520: Duplicate WHERE condition variable in WHERE condition

The two condition variables associated with any WHERE condition must be distinct.

LBCC1530: Delete option already specified

The variable delete option may be specified at most one time for any given variable.

LBCC1540: Variable already assigned a value

One of ASSIGNTRUE or ASSIGNFALSE may be assigned to a variable, but not both.

LBCC1550: FOR ALL and WHERE conditions are contradictory.

All instances defined by the FOR ALL statement(s) violate at least one of the WHERE conditions. Hence the fact does not produce any clauses, and should be removed.

LBCC1560: THERE EXISTS and WHERE conditions are contradictory.

All instances defined by the THERE EXISTS statement(s) violate at least one of the WHERE conditions. Hence the fact does not produce any clauses, and should be removed.

LBCC1570: Level or % value expected.

Keywords AT LEVEL must be followed by, or % must be immediately preceded by, an integer: 1 ..100.

LBCC1580: Three digits maximum for level or % value.

A level or % value may have at most three digits.

LBCC1590: Invalid level value: non-numeric characters.

A level or % value must consist of integer digits.

LBCC1600: Level value must be an integer: 1 .. 100.

Correct the level or % value so that it falls in the specified range.

LBCC1610: Improper level or % term.

An improper level is given in connection with AT LEVEL keywords, or an improper % term is given.

LBCC1620: Likelihood must be followed by period terminating fact.

A likelihood value or term may only be specified at the end of a fact and thus must be followed by a period.

LBCC1630: Brackets [and] are not allowed in IF THEN statement

The fact has IF THEN form and contains the bracket [or]. This is not allowed.

Fatal Errors

LBCC2010: MINIMIZE or SATISFY expected

The first non-comment in the input file must be one of the two indicated keywords.

LBCC2020: Keyword missing: SETS, PREDICATES, VARIABLES, FACTS

One of the indicated keywords must be given to indicate the beginning of a particular declaration section.

LBCC2030: Unexpected end of source

The last record of the input file was read before an ENDATA was encountered.

LBCC2040: Unexpected ENDATA encountered

ENDATA was encountered before all sections of the input were properly completed.

LBCC2050: Too many sets; at most ⟨N⟩ allowed

⟨N⟩ represents the maximum number of set definitions which are allowed in a given input file. This number was exceeded.

LBCC2060: Keyword DEFINE expected

DEFINE is required prior to a set name when defining sets, or prior to a predicate name when defining predicates.

DEFINE is also expected on the line following immediately the keyword SETS or PREDICATES.

LBCC2070: SETS section already encountered.

A second SETS declaration was encountered. There may be at most one such section.

LBCC2080: Cannot exceed ⟨N⟩ elements.

⟨N⟩ represents the total number of unique elements that are allowed. This number was exceeded.

LBCC2090: Too many predicates; at most ⟨N⟩ allowed

⟨N⟩ represents the total number of predicates which may be defined. This number was exceeded.

LBCC2100: Keyword ON expected

The keyword ON was expected in the predicate definition but was not encountered.

LBCC2110: Cannot exceed ⟨N⟩ total variables

⟨N⟩ represents the total number of predicate variables, propositional variables, and new variables (introduced during conversion to internal format) that are allowed. This number was exceeded on the given line number.

LBCC2120: New variable addition exceeds max total variables: ⟨N⟩

⟨N⟩ represents the total number of variables allowed. The addition of a new variable during the conversion process caused this number to be exceeded.

LBCC2130: PREDICATES section already encountered

A second PREDICATES section was encountered. There may be at most one such section.

LBCC2140: SETS section must come before PREDICATES

The SETS keyword was encountered after the PREDICATES section.

LBCC2150: VARIABLES section already encountered.

A second VARIABLES section was encountered; there may be at most one such section.

LBCC2160: SETS (PREDICATES) section must come before VARIABLES

The indicated section was encountered after the VARIABLES section.

LBCC2170: Too many Boolean variables; at most ⟨N⟩ allowed

⟨N⟩ indicates the maximum number of propositional variables allowed. This number was exceeded.

LBCC2180: SETS (PREDICATES) (VARIABLES) section must come before FACTS

The indicated section was encountered after the FACTS section.

LBCC2190: PREDICATES section expected

The SETS section was declared, but the VARIABLES or FACTS section was the next section in the input. If the SETS section is specified, then the PREDICATES section must immediately follow.

LBCC2200: Too many variables in the clause; at most ⟨N⟩ allowed

There may be at most ⟨N⟩ literals in a clause. Reformulate the statement.

LBCC2210: INCLUDE statement has file name missing

INCLUDE must be followed by input file name.

LBCC2220: Cannot open file ⟨file name⟩

INCLUDE statement specifies ⟨file name⟩, but no such file exists.

LBCC2230: Input record too long

Input records may have at most 125 characters, including blanks. The record of the given line number exceeds that maximum.

LBCC2240: Number of input files exceeds maximum allowed (= ⟨N⟩)

The number of input files that may be simultaneously opened by INCLUDE statements is limited to the given number. Reduce the use of INCLUDE statements so that this maximum is not exceeded.

LBCC2250: Conflicting INCLUDE statements for ⟨file name⟩

The file name is listed in an INCLUDE statement, but currently is already open. Change INCLUDE statement(s) to eliminate such conflicting use of the file.

LBCC2260: GOAL variable specified but not used

The GOAL variable has been defined in the VARIABLES section but has not been used in the FACTS section. This is not allowed.

System Errors

The compiler performs various compatibility checks and outputs an error message when an inconsistency is detected. An error messages is of the form

```
*************LEIBNIZ SYSTEM ERROR***************
ERROR IN ⟨name⟩ LINE ⟨line number⟩
************************************************
```

If the error is detected while the file leibnizparams.dat is read, then the message is recorded in a file named leibnizparams.err. If the error is detected after the reading of leibnizparams.dat, the error is recorded in the .err file that also records warnings and errors concerning the .log file. In some cases, the system tries to recover from the error. When this is possible, processing continues, and the error message can be ignored. Thus, an error message is significant only if processing is halted due to the error.

Please save the .err file and the input .log file, and report the error to the author. These files are needed for the analysis of the difficulty.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Chapter 4

Logic Reasoning

Introduction

This chapter covers all steps of compiling .log files and using the compiled solution programs, related files, and the execution procedures in a user C program to carry out logic reasoning.

The steps are not difficult. They are best explained by going through an example that entails all aspects. The example consists of the logic formulation of machine.log and the user program machine.c.

We first look at makefile.machine for Linux/Unix installations, which specifies all steps. For other installations, the steps are described in that makefile.

Version 16.0 1 July 2015

makefile.machine

The following lines of makefile.machine pertain to the processing of machine.log and machine.c.

```
LEIBNIZ = /home/Leibniz/Lbcc/Code/
include $(LEIBNIZ)makefile.lbcc.object.list
# uncomment just one of the two statements below to get
# compile options for optimized code or debugging
COMPOPTIONS = -Wall -O -c
#COMPOPTIONS = -Wall -g -c
all:  machine
machine:    machine.log machine.c leibnizerroraction.c
    cp leibnizparams.machine leibnizparams.dat
    $(LEIBNIZ)lbcc
    gcc $(COMPOPTIONS) machine.c
    gcc $(COMPOPTIONS) leibnizerroraction.c
    gcc $(COMPOPTIONS) leibniztrans.c
    gcc -o machine machine.o \
                  leibnizerroraction.o \
                  $(LBCCOBJS)
```

We discuss the makefile line by line.

`include $(LEIBNIZ)makefile.lbcc.object.list`
Includes a file that defines `LBCCOBJS` used later.

`machine: machine.log machine.c leibnizerroraction.c`
Specifies that the executable program machine depends on three files: machine.log, machine.c, and leibnizerroraction.c. The file machine.log was introduced in Chapter 3. The remaining two files are treated later in this chapter. Suffice it to say the following here. The program in machine.c is written by the user. It is an expert system for machine malfunction. The file leibnizerroraction.c is available as template from Leibniz/Makedata. It defines the actions taken when the number of warnings or errors during execution exceed specified limits. The user can customize the file to achieve any response.

`cp leibnizparams.machine leibnizparams.dat`
Copies leibnizparams.machine into leibnizparams.dat, in preparation for the compile step using lbcc.

`$(LEIBNIZ)lbcc`
Applies the lbcc compiler to machine.log. That step produces the files machine.err, machine.prg, machine.trs, leibnizdefs.h, leibnizexts.h, and leibniztrans.c.

`gcc $(COMPOPTIONS) machine.c`
Compiles the user program of the expert system for machine malfunction. The file machine.c contains 'include' statements for the files leibnizexts.h and leibnizmacro.h. The file leibnizexts.h inserts into machine.c definitions of the sets and logic variables of machine.log; the memory allocation for those variables is not done here, but is part of the definition of the transfer process. That process is described later. The file leibnizmacro.h defines communication arrays and *Leibniz System* procedure names, which are also discussed later.

`gcc $(COMPOPTIONS) leibnizerroraction.c`
Compiles leibnizerroraction.c. This can be the original file from Leibniz/Makedata or a file customized by the user.

`gcc $(COMPOPTIONS) leibniztrans.c`
Compiles the code for the transfer process. The file leibniztrans.c has 'include' statements that insert the files leibnizdefs.h and machine.trs. The file leibnizdefs.h defines the sets and logic variables of machine.log in agreement with the definitions done by leibnizexts.h, and it also allocates memory for those variables.

`gcc -o machine machine.o ...`
Links the listed object code to create the executable program machine. The program machine can be used as an expert system. The system, which is purposely small to simplify the discussion to follow, is intended as a template for the implementation of real expert systems.

The next sections fill in details about the files and steps.

.prg File

Each .prg file contains a solution program coded in ASCII format. The first 14 lines of a .prg file are a preamble that contains summary statistics of the solution program. We explain the format of the preamble using the first part of machine.prg. The rest of the .prg file is of no concern for the user.

```
**********************************************************
Program for Automated_Assembly                           *
SATISFY problem                                          *
Program compiled with Leibniz System Version 16.0        *
Execution time bound (sec) = 0.0319                      *
if machine speed (mips) = 150                            *
Minimum parameter values for execution                   *
Parameter Min Value                                      *
COLMAX       30                                          *
ROWMAX       62                                          *
ANZMAX      146                                          *
BLKMAX        2                                          *
STOSIZ      139                                          *
End of parameter data                                    *
```

The preamble of machine.prg begins with the statement `Program for Automated_Assembly` and terminates with `End of parameter data`. The interpretation of the lines is as follows:

`Program for Automated_Assembly`: At execution time, the problem will be referred to as Automated_Assembly.

`SATISFY problem`: The logic formulation involves a satisfiability problem, as opposed to a minimization problem. In the latter case, the line would be `MINIMIZE problem (solved EXACTLY)` or `MINIMIZE problem (solved APPROXIMATELY)`, depending on whether the solution program produces exact or approximate solutions.

`Program compiled with Leibniz System Version 16.0`: Gives the version number of the compiler.

`Execution time bound (sec) = 0.0319 if machine speed (mips) = 150`: Gives the time bound in seconds for execution of the solution program. Here, the speed is 150 mips (million instructions per second), and the time bound is 0.0319 sec.

The line `Execution time bound (sec) =` is also included in .prg files for exact minimization. However, in the case of approximate minimization, that line is replaced by two lines `Estimated execution time (sec) =` and `Satisfiability check time bound (sec) =` . The value given on the first line is a rough estimate of the run time (in seconds) of the solution program, while the second one is an upper bound, also in seconds, on the solution time of the satisfiability problem underlying the minimization problem.

`Minimum parameter values for execution`: Below that line, minimum values of the parameters COLMAX, ROWMAX, ANZMAX, BLKMAX, and STOSIZ for execution of the solution program are listed. These parameter values are used shortly.

machine.c

We discuss the file machine.c next. In contrast to earlier coverage of files, we do not list the file in its entirety, but display key lines and explain the actions they cause.

```
/* first record of machine.c***** */
/*
* -----------------------------------------------------
* machine example expert system
* copyright 1990-2001 by Leibniz Company, Plano, Texas
*
* -----------------------------------------------------
*
*/
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include "leibnizexts.h"
/*
*/
int main() {
/*
* include leibnizmacro.h file, which contains
*  - leibniz parameters for procedures
*  - alternate definitions of leibniz procedures
*/
#include "leibnizmacro.h"
/*
* miscellaneous variables
*/
char answer[10];
long pi;
PRGNAME_ARRAY prgname[2];
```

Three lines above deserve attention: The 'include' statements for files leibnizexts.h and leibnizmacro.h, and `PRGNAME_ARRAY prgname[2];` for memory allocation. We discuss these lines.

The file leibnizexts.h is needed for the transfer process. The file allows propositional variables and predicate variables of machine.log to appear in machine.c without any programming effort. For example, machine.log has the predicate conflict defined on the set scenarios = {z1,z2,z3,z4}. Thus, machine.log has the predicate variables conflict(z1), conflict(z2), conflict(z3), and conflict(z4). These same variables also occur in machine.c, by virtue of leibnizexts.h. Note that these variables have parentheses and not square brackets, in contrast to the usual notation of the C language. The parentheses set off these variables from other, user-defined, variables in machine.c.

There is one difference, though, between the variables conflict(z1)–conflict(z4) in machine.log and machine.c. In machine.log, they are logic variables that can take on the values True and False. In machine.c, the variables are long integers, and thus can take on 0, negative integer values, or positive integer values. The interpretation is as follows.

A positive value encodes True, a negative value encodes False, and the 0 denotes that the variable is deleted. The encoding also represents whether a True/False value was assigned in machine.c by the user or was determined by the solution program produced from machine.log. Specifically, a value of 3 or higher means that the value True was assigned in machine.c, while the value 1 means that the value True was assigned by the solution program. Similarly, -3 or lower means False assigned by machine.c, while -1 means False assigned by the solution program. The solution program does not assign the values 2 and -2. If the user assigns ± 1, then this means, from the viewpoint of the solution program, that the user has not fixed that variable to True or False and allows assignment of any value by the solution program.

The file leibnizmacro.h defines the following constants.

```
#define ACTIVE   (-1)
#define DELETED    0
#define TRUE       1
```

```
#define FALSE    (-1)
#define FIXTRUE    3
#define FIXFALSE (-3)
```

According to these and the earlier definitions, when a variable in machine.c has positive value, we could equivalently say that the value is >= TRUE. If the value is negative, it is <= FALSE. If the value is 3 (resp. -3), and thus enforces True (resp. False), one could equivalently assign FIXTRUE (resp. FIXFALSE). Finally, the value 0 signals deletion, and equivalently one could assign DELETED. These facts are used in machine.c.

A key aspect of the transfer process is that the elements of the sets of machine.log are declared to be integers in machine.c. For example, z1, z2, z3, and z4 of machine.log are the integers 5, 6, 7, and 8 in machine.c, as can be readily seen in leibnizexts.h. In this case, the integers are consecutive. But generally this need not be so. Indeed, consecutive integers are assigned by lbcc to the elements in the order in which they are encountered for the first time during processing of the SETS section. So if z1, z2, z3, and z4 are not consecutively introduced in that section, then they do not receive consecutive integer values. To force consecutive values, one should define as first set in SETS the set UNIVERSE, and then lists for that set list all subsequently used elements in the desired order. Given such order, one later can loop in machine.c to process predicate variables in an efficient manner. This feature is used in machine.c.

We turn to the file leibnizmacro.h. It contains short and long definitions of *Leibniz System* procedures. The short version omits all arguments of the procedures. An example is the short version 'solve_problem()' of the long version 'sol_prb(name,state,type,value,errcde)'.

CAUTION: Even though the short version does not list the arguments of the long version, those arguments nevertheless apply since `#define` statements in leibnizdefs.h and leibnizexts.h declare the short versions actually to be the long versions. In this manual, we always use the short versions and, for simplicity, omit the terminating parentheses "()."

The arguments of the long version are communication arrays defined in leibnizmacro.h. For almost all procedures, the communication arrays are as follows.

Name	Interpretation	Usage
name	Name of file, problem, variable, or clause, or output string	Input and Output
state	Status of problem, variable, or clause	Input and Output
type	Type of problem, variable, or clause	Input and Output
value	Cost or other integer data	Input and Output
errcde	Error code	Output only

Each procedure makes particular use of the communication arrays. The rules are listed later in this chapter for each procedure. Considerations that apply generally are listed next.

CAUTION: Except for variable names, clause names, and problem names, character strings passed to the execution procedures are not case sensitive, i.e., any combination of upper and lower case letters is allowed. The exceptional variable names, clause names, and problem names must be given exactly as in the defining .log file, except for the possibly used wildcard character "?".

CAUTION: When calling execution procedures, do not use string constants in the argument list instead of the arrays. Instead, copy such strings into the appropriate arrays, and then specify the array in the call. That defensive programming practice avoids certain programming errors that are almost impossible to identify.

CAUTION: When a procedure has been executed and control has been returned to the user program, then some output is typically returned to the user program via the interface arrays listed above. Details about that output are given subsequently for each procedure. Except for such specified output, the remaining entries of the interface arrays may or may not contain usable data. Thus, it is best to assume that any array entries not specifically declared to contain output data may have been changed from the input value, and thus are unusable.

CAUTION: The allocation of the communication arrays is done in leibnizmacro.h. It is recommended that this file is included within each user procedure so that the communication arrays are local variables. This prevents unwanted interference of communication arrays used in different user procedures.

There are three exceptional procedures that use communication arrays beyond those listed above: initialize_problem, maxcls_sat, and mincls_unsat. The line `PRGNAME_ARRAY prgname[2];` of machine.c defines one such additional communication array. The next section of machine.c makes use of that array when calling initialize_problem.

```
/*
* initialize problem in machine.prg
*/
value[0] = 1;
strcpy(name,"machine.err");
strcpy(prgname[0],"machine.prg");
strcpy(prgname[1],"");
initialize_problem();
```

The lines `value[0] = 1;` and `strcpy(name,"machine.err");` establish that execution warning and error messages are to be posted to the screen and in the file machine.err.

The line `strcpy(prgname[0],"machine.prg");` says that the solution program of the file machine.prg is to be loaded. If, say, n-1 additional programs are to be loaded, the names of these .prg files are specified in prgname[1], prgname[2], . . . , prgname[n-1]. The list is terminated by assigning to prgname[n] the empty string. Here, we do this by `strcpy(prgname[1],"");`

<u>CAUTION</u>: The allocation for prgname in machine.c must be sufficiently large to handle the list of .prg files. This is the main reason that the allocation is done in machine.c and not in leibnizmacro.h.

<u>CAUTION</u>: The initialization of solution programs does not initialize the related logic variables in the user program.

The name initialize_problem is the short version of inz_prb(prgname, name,state,type,value,errcde). The procedure fetches the parameter values in the preamble of machine.prg, assigns devices, allocates memory, and loads the solution program of machine.prg. The procedure carries out these tasks by invoking the procedures get_dimension, assign_device, begin_problem, store_problem, and status_problem. We need not concern ourselves with these procedures right now, but the reader interested in their function may want to look them up in the latter part of this chapter.

The procedure initialize_problem also defines default limits on the number of warnings, nonfatal errors, and fatal errors. The default limits are listed in the procedure limit_error later in this chapter. That procedure may also be used to reset the limits. When a limit is reached, the system calls the procedure of leibnizerroraction.c. We skip listing that file, but encourage the user to look at it. The user has this file under full control and may change it in any way to induce the desired response to warnings and errors.

We skip some output statements of machine.c and move on to the following part.

```
/*
* determine if crash may occur at all
*/
printf("Determine whether there is a worst case scenario\ n"
"where crash may happen\ n");
```

```
/*
* initialize all logic variables to ACTIVE (= -1)
* set crash to FIXTRUE (= 3) and solve problem
* any satisfying solution is a possible crash scenario
*/
strcpy(state,"A");
modify_allvar();
crash = FIXTRUE;
solve_problem();
if (strcmp(state,"U")==0) { ...
```

The lines `strcpy(state,"A");` and `modify_allvar();` initialize all logic variables to active by assigning ACTIVE (-1). Actually, one could assign $+1$ to achieve the same effect, since any one of these values means that the user has not fixed the variable. Such fixing is done by the line `crash = FIXTRUE;`, which fixes the variable crash to 3 and thus to True.

CAUTION: The procedure modify_allvar is one of two procedures that modify values of logic variables in the user program. The second procedure is solve_problem. We emphasize that no other procedure changes any of these values. The main role of modify_allvar is easy initialization of all logic variables in the user program to one value. If that procedure is not called, the user program should initialize the logic variables in separate statements since the default initialization values depend on the installation and should not be relied upon.

The procedure solve_problem transfers all True/False values of logic variables defined via integers in machine.c into the solution program, checks if there is a satisfying solution for the logic formulation of machine.log that respects the assigned True/False values, and, if there is such a solution, transfers that solution from the solution program to machine.c. Existence of such a solution is indicated by a string in the communication array state. If that string is "U" (short for "UNSATISFIABLE"), then no satisfying solution exists. If the string is "S" (short for "SATISFIABLE"), then a solution exists and is transferred into the program of machine.c. Based on the outcome, machine.c tells either that a machine crash is not possible, or gives the scenario of a crash.

The rest of machine.c consists of a loop where True/False are asked for some predicate variables, and where, based on those values, it is decided whether a machine crash is possible.

At this point, the user has enough insight to build a logic module for some application, and to encode the information in two files, say, application.log and application.c analogously to machine.log and machine.c.

Much more complicated situations can be handled using the various execution procedures described later in this chapter. One may also compile several .log files and use the resulting solution programs simultaneously. Such use requires careful management of the transfer process. This is covered in the next section.

Transfer Process for Several .prg Files

When the user program is to execute the solution programs of several .prg files, the transfer process must simultaneously manipulate the logic variables of the underlying .log files. To assure correctness of that process, one should assemble the .log files, the related parameter files, and some additional files in a certain manner and sequence, and should use a particular makefile to control the compilation steps. The details are as follows.

Define the sets, predicates, and propositional variables of all .log files simultaneously in one file called userlogdefs.h. Thus, userlogdefs.h consists of one SETS section, one PREDICATES section, and one VARIABLES section. Using the INCLUDE feature of .log files, insert userlogdefs.h into each .log file.

Using leibnizparams.machine as template, create a parameter file for each .log file and name it using the following convention. For processing of file xxx.log, use the name leibnizparams.xxx. In each parameter file, comment out the line

```
keep all variables even if not used in any fact
```

Thus, when lbcc compiles xxx.log, only the predicate variables and propositional variables occurring in the FACTS of xxx.log are retained for the solution program in xxx.prg.

On the other hand, the line

```
make transfer files
```

must be active so that the appropriate .trs file is generated. That option also generates files leibnizdefs.h, leibnizexts.h, and leibniztrans.c. The latter files are not used, in fact are overlaid later.

CAUTION: If the line

```
keep all variables even if not used in any fact
```

is not commented out, the procedures modify_allvar and solve_problem when called in the user program change values of logic variables that are not part of the facts of a given file xxx.log. This may cause hard-to-find errors.

Create a file all.trs that for each .log file contains one line. If the .log file is xxx.log, then the line is `#include"xxx.trs"` .

Create an additional .log file, say total.log . That file includes userlogdefs.h defined earlier, has one additional variable, say total, which is defined by the line `total` immediately after the statement `INCLUDE userlogdefs.h`. The FACTS section contains the line `total.` as single fact.

CAUTION: The file total.log is not represented in all.trs by an "#include" line.

Define the parameter file leibnizparams.total for total.log to be like leibnizparams.machine. It must have the two lines
`keep all variables even if not used in any fact`
and
`make transfer files`
activated. Comment out the line
`list warnings for nonuse of sets, predicates, variables`
since otherwise every predicate variable and every propositional variable except for the variable total produces a warning.

Create a makefile for processing of the above files and for compilation with the user program. Use as template the file makefile.multilog of Leibniz/Makedata. That makefile is listed shortly. The makefile supposes that two .log files app1.log and app2.log are to be handled. It further is assumed that the files userlogdefs.h, leibnizparams.app1, leibnizparams.app2, total.log, leibnizparams.total, and all.trs have been created as described above. The user code is assumed to be in multilog.c.

Copies of the files app1.log, app2.log, total.log, leibnizparams.app1, leibnizparams.app2, leibnizparams.total, all.trs, multilog.c, and of the makefile makefile.multilog are provided in Leibniz/Makedata.

Here is makefile.multilog. It contains explanations of the steps as comments.

```
LEIBNIZ = /home/Leibniz/Lbcc/Code/
include $(LEIBNIZ)makefile.lbcc.object.list
# uncomment just one of the two statements below to get
# compile options for optimized code or debugging
COMPOPTIONS = -Wall -O -c
#COMPOPTIONS = -Wall -g -c
all:  multilog total.prg app1.prg app2.prg

# compile total.log
# as last step, replace total.trs produced by lbcc
# by the file all.trs

total.prg:    total.log leibnizparams.total \
    app1.prg app2.prg
    cp leibnizparams.total leibnizparams.dat
    $(LEIBNIZ)lbcc
    cp all.trs total.trs

# compile app1.log

app1.prg:    app1.log leibnizparams.app1
    cp leibnizparams.app1 leibnizparams.dat
    $(LEIBNIZ)lbcc

# compile app2.log

app2.prg:    app2.log leibnizparams.app2
    cp leibnizparams.app2 leibnizparams.dat
    $(LEIBNIZ)lbcc

# compile multilog.c and link with lbcc output files

multilog:    multilog.c total.prg \
        leibnizerroraction.c
    gcc $(COMPOPTIONS) multilog.c
```

```
        gcc $(COMPOPTIONS) leibnizerroraction.c
        gcc $(COMPOPTIONS) leibniztrans.c
        gcc -o multilog multilog.o \
                     leibnizerroraction.o \
                     $(LBCCOBJS)
```

The next section explains the execution of several solution programs.

Execution of Several Solution Programs

The file multilog.c of Leibniz/Makedata is an example for the use of several solution programs. That file can be employed as template for other situations. Below, portions of the file are listed and explained.

The following lines of multilog.c carry out the initialization steps for app1.prg and app2.prg.

```
value[0] = 1;
strcpy(name,"multilog.err");
strcpy(prgname[0],"app1.prg");
strcpy(prgname[1],"app2.prg");
strcpy(prgname[2],"");
initialize_problem();
```

The solution program processed last—here, the solution program of app2.prg—is also loaded for execution. It is best not to rely on that convention, and always to retrieve a problem from storage whenever it is needed. The retrieval is done using the problem name specified in the .log file. Here, app1.log and app2.log contain the problem names application1 and application2, respectively.

Retrieval of the solution program for application1 is accomplished by the following two lines, using the procedure retrieve_problem described later in this chapter.

```
strcpy(name,"application1");
retrieve_problem();
```

The procedure modify_allvar is used to initialize the logic variables occurring in a given problem. This is done as follows.

```
strcpy(state,"A");
modify_allvar();
```

CAUTION: The procedure modify_allvar modifies only the logic variables of the currently loaded problem, in this case, of the problem application1. All other logic variables are not affected.

The rest of multilog.c is similar to the case of machine.c.

Procedure Summary

In alphabetical order, the procedures are as follows:

Assign Devices

Procedure:	assign_device
Function:	Allocate memory, and assign device for output of error messages.

Begin Problem

Procedure:	begin_problem
Function:	Load program for a specified logic problem into memory.

Delete Problem

Procedure:	delete_problem
Function:	Delete a stored problem.

Execute lbcc Compiler

Procedure:	execute_lbcc
Function:	Execute lbcc compiler.

Free Devices

Procedure: free_device
Function: De-allocate memory and de-assign device for output of error messages.

Get Clause Name or Goal Name

Procedure: get_clsname
Function: Get clause name or goal name.

Get Dimension

Procedure: get_dimension
Function: Get parameter minimum values for COLMAX, ROWMAX, ANZMAX, BLKMAX, STOSIZ of one .prg file.

Get Dual Value

Procedure: get_dual
Function: Get dual value for clause or goal with specified index.

Get Goal

Procedure: get_goal
Function: Get goal information for specified goal name.

Get Primal Classification

Procedure:	get_primal
Function:	Get primal classification of specified variable.

Get Value of Variable

Procedure:	get_value
Function:	Get value of specified variable. Cannot be executed when transfer process is used.

Initialize Problems

Procedure:	initialize_problem
Function:	Initialize any number of problems using get_dimension, begin_problem, store_problem, and status_problem.

Limit Warnings and Errors

Procedure:	limit_error
Function:	Define limits on number of warnings and errors during execution.

Maximal Subset of Satisfied Clauses

Procedure:	maxcls_sat
Function:	Find maximal subset of specified clauses for which problem is satisfiable.

Minimal Subset of Unsatisfied Clauses

Procedure:	mincls_unsat
Function:	Find minimal subset of specified clauses for which problem remains unsatisfiable.

Modify Status of All Variables

Procedure:	modify_allvar
Function:	Modify the status of logic variables in user program. Requires use of transfer process.

Modify Status of Clause

Procedure:	modify_clause
Function:	Modify the status of all clauses matching a given name and having likelihood level within a specified interval.

Modify Status of Goal

Procedure:	modify_goal
Function:	Modify the status of all goals matching a given name.

Modify Status of Current Problem

Procedure:	modify_problem
Function:	Modify problem type from minimization to satisfiability problem, or conversely.

Modify Status of Variable

Procedure:	modify_variable
Function:	Modify the status of all variables matching a given name.

Restore Problem

Procedure:	restore_problem
Function:	Restore current problem to its original form. The procedure removes some or all status modifications by prior procedure calls.

Retrieve Problem

Procedure:	retrieve_problem
Function:	Retrieve a stored problem and load as current problem.

Solve Current Problem

Procedure:	solve_problem
Function:	Solve current satisfiability or minimization problem.

Status of Clause

Procedure:	status_clause
Function:	Return status information about all clauses matching a given name and having likelihood level within a specified interval.

Status of Element

Procedure:	status_element
Function:	Return status information for specified element.

Status of Current Problem

Procedure:	status_problem
Function:	Return status information for current problem.

Status of Storage

Procedure:	status_storage
Function:	Return status information for storage of problems, plus name of problem for specified index.

Status of Variable

Procedure:	status_variable
Function:	Return status information for specified variable.

Store Current Problem

Procedure:	store_problem
Function:	Store current problem.

Procedure Hierarchy and General Rules

The following rules govern calling of the procedures. Except for these rules, the procedures may be called at any time and in any order.

initialize_problem	must be called first except if assign_problem and begin_problem are used. Other exceptions are get_dimension and limit_error, which may be called at any time.
assign_device	must be the first procedure except if initialize_problem is used. Other exceptions are get_dimension and limit_error, which may be called at any time.
begin_problem	must be the second procedure except if initial loading of problems is done by initialize_problem. Other exceptions are get_dimension and limit_error, which may be called at any time. Subsequently, begin_problem may be used whenever another solution program is to be loaded.
free_device	may be used only after allocation of devices has been done by initialize_problem or assign_device.
get_dual	may be used only if the current problem has been solved by approximate minimization and has been found to be satisfiable.
get_primal	may be used only if the current problem has been solved by approximate minimization and has been found to be satisfiable.
get_value	may be used only after a problem has been solved and has been found to be satisfiable. It can be called only if the transfer process is not used.
modify_allvar	requires the use of the transfer process.

The subsequent pages give for each procedure the required input parameters and the call statement.

assign_device

Description

Allocate memory, and assign device for output of error messages. The procedure is not needed when initialize_problem is used.

Overview

For the allocation of memory, the user must extract and use certain parameter values from the preambles of the .prg files that are to be loaded. We use the preamble of machine.prg to discuss this step.

```
*********************************************************
Program for Automated_Assembly                          *
SATISFY problem                                         *
Program compiled with Leibniz System Version 16.0       *
Execution time bound (sec) = 0.0319                     *
if machine speed (mips) = 150                           *
Minimum parameter values for execution                  *
Parameter Min Value                                     *
COLMAX       32                                         *
ROWMAX       62                                         *
ANZMAX      146                                         *
BLKMAX        2                                         *
STOSIZ      133                                         *
End of parameter data                                   *
```

For memory allocation, the entries below the line `Parameter Min Value` contains all needed information.

Specifically, the subsequent lines starting with COLMAX, ROWMAX, ANZMAX, and BLKMAX contain under the heading "Min Value" the minimum values that must be assigned to the parameters COLMAX, ROWMAX, ANZMAX, and BLKMAX. For example, COLMAX must have at least value 32, ROWMAX must be at least 62, ANZMAX must be at least 146, and BLKMAX must be at least 2.

If several problems are to be loaded, then one must select for each parameter COLMAX, ROWMAX, ANZMAX, and BLKMAX the maximum of the values displayed under "Min Value' in the corresponding .prg files. For example, if for another problem the .prg file has for COLMAX the "Min Value" equal to, say, 283, then for the purposes of memory allocation, COLMAX must take on the maximum of 30, which is the value from the machine.prg file, and 283. Thus, COLMAX must be at least 283, and it suffices that COLMAX be selected as 283. The same arguments apply to the parameters ROWMAX, ANZMAX, and BLKMAX.

Two additional parameters, called PRBMAX and STOMAX, are required for memory allocation. The first parameter PRBMAX is the total number of problems that are to be stored plus 1. The additional 1 is for the loaded problem. For example, if one intends to store internally the machine problem of machine.prg plus one additional problem, then in total two problems will be stored internally, and PRBMAX must be $2 + 1 = 3$. The second parameter STOMAX concerns the memory allocation of all stored problems. STOMAX must be at least the sum of the "Min Values" of the STOSIZ parameter of the .prg files for the problems that are to be stored internally.

CAUTION: STOMAX must be at least the sum of the STOSIZ parameters, not just the maximum of these parameters.

If just one problem is to be loaded, then it suffices to select the COLMAX, ROWMAX, ANZMAX, and BLKMAX values to be the "Min Value" entries of the .prg file. Furthermore, PRBMAX is equal to 2, and STOMAX is equal to the STOSIZ value. Indeed, if the user does not intend to store the problem internally at all, then it suffices to set STOMAX equal to 2.

<u>CAUTION</u>: COLMAX, ROWMAX, ANZMAX, BLKMAX, PRBMAX, STOMAX must always be greater than or equal to 2.

Error messages are recorded in a file whose name is specified by the user in the parameter name. The user may request that error messages are also written to the screen, by setting the parameter value[0] = 1. Any other assignment to value[0] causes writing of error messages to the screen to be suppressed.

<u>CAUTION</u>: The error file specified in the parameter name is opened by assign_device, and must not be opened by the user.

<u>CAUTION</u>: If the name of the error file specified in the parameter name is the empty string, the system assumes an inadvertent user error and substitutes the name execute.err.

<u>CAUTION</u>: The procedure assign_device resets limits on the number of warnings, nonfatal errors, and fatal errors to the default values. See limit_error for these default values. To change these values, execute limit_error immediately after assign_device.

Execution from User Program

```
strcpy(name,⟨error file name⟩);
value[0] = ⟨flag for writing error messages on screen⟩
           (= 1: write error messages on screen)
           (≠ 1: suppress error messages on screen)
value[1] = ⟨COLMAX⟩;
value[2] = ⟨ROWMAX⟩;
value[3] = ⟨ANZMAX⟩;
```

```
  value[4] = ⟨BLKMAX⟩;
  value[5] = ⟨PRBMAX⟩;
  value[6] = ⟨STOMAX⟩;
  assign_device();
```

The statement `assign_device();` is the short version of
`asg_dev(name,state,type,value,errcde);`

Information Returned to User Program

type	=	Leibniz System Version
value[7]	=	Amount of memory used for arrays (K bytes)
errcde[0]	=	Error code (see Error Messages below)

begin_problem

Description

Load program for specified problem into memory. The procedure is not needed when initialize_problem is used. The procedure may be used repeatedly to load programs for various problems.

<u>CAUTION</u>: If the transfer process is used, the procedure does not initialize values of logic variables in the user program. Such initialization must be made by the user code or by calling modify_allvar.

Execution from User Program

```
strcpy(name,"⟨name of .prg file⟩");
begin_problem();
```

The statement `begin_problem();` is the short version of
`beg_prb(name,state,type,value,errcde);`

<u>CAUTION</u>: The name of the .prg file must include the entire path unless execution is done in the directory in which the .prg file resides.

<u>CAUTION</u>: The backslash "\" is an escape character of C. Thus if a backslash occurs in the path name, two backslashes "\\" must actually be used.

Information Returned to User Program

value[0]	=	Number of variables
value[1]	=	Number of clauses
type	=	Type of problem
	=	"SAT" if satisfiability problem
	=	"MIN" if minimization problem
errcde[0]	=	Error code (see Error Messages below)

delete_problem

Description

Delete a stored problem. Procedure does not affect currently loaded problem.

Execution from User Program

```
strcpy(name, "⟨name of problem to be deleted⟩");
delete_problem();
```

The statement `delete_problem();` is the short version of
`del_prb(name,state,type,value,errcde);`

Information Returned to User Program

value[1]	=	Total number of problems
value[2]	=	Total number of records used for storage of problems
value[3]	=	Total number of records still available for storage of problems
errcde[0]	=	Error code (see Error Messages below)

CAUTION: The statistics hold after the deletion.

execute_lbcc

Description

Read parameter file leibnizparams.dat in directory specified in name, and execute the lbcc compiler. When lbcc is done, the compiled program has been loaded and is ready for execution.

CAUTION: If the transfer process is used, the procedure does not initialize values of logic variables in the user program. Such initialization must be made by the user code or by calling modify_allvar.

CAUTION: If the transfer process is used for the compiled program, then the program must have been compiled earlier and the user code must have been linked with the code produced by that compilation. It is recommended that for that original compilation and for the compilation by execute_lbcc the option
`keep all variables even if not used in any fact`
is specified in leibnizparams.dat. If that is not done, the transfer process cannot be guaranteed to be reliable. Indeed, if the option is commented out and if in the two compilations different variables are used in the FACTS section, then different variables occur in the two solution programs, and the transfer process becomes necessarily faulty.

Execution from User Program

```
strcpy(name, "⟨directory of leibnizparams.dat⟩");
execute_lbcc();
```

Directory name must terminate with "/" (Unix) or "\\" (MS Windows). Current directory may be specified by name = empty string.

The statement `execute_lbcc();` is the short version of `exe_lbcc(name,state,type,value,errcde);`

Information Returned to User Program

If so specified in leibnizparams.dat, errors are displayed on the screen, and the .prg file of the program is created.

free_device

Description

De-allocate memory and de-assign output device for error messages.

Execution from User Program

```
free_device();
```

The statement `free_device();` is the short version of
`fre_dev(name,state,type,value,errcde);`

Information Returned to User Program

errcde[0]	=	Error code (see Error Messages below)

get_clsname

Description

Get clause name or goal name for specified index.

Overview

When a fact in the .log file of the logic formulation has been named, then all clauses produced by that fact receive that same name. In addition, each goal has always a name assigned in the .log file. One may retrieve the clause and goal names by specifying an index that may range between 1 and the total number of clauses. The latter number may be obtained by status_problem.

Besides named clauses and goals, a .log file may contain any number of unnamed facts. Accordingly, there may be a number of clauses without name. Actually, such clauses do have an internally assigned name, but that name is not accessible to the user. When a clause index refers to such an internally named clause, then it is indicated that the clause has a name that may not be accessed by the user.

For each index, the procedure indicates whether the clause or goal is active or deleted. For each index corresponding to a clause, the procedure also returns the likelihood level. Explanations for this information are included under status_clause.

CAUTION: get_clsname and get_dual are the only procedures that utilize indices in connection with clauses and goals. All other procedures involving clauses or goals must use names.

Execution from User Program

```
value[0] = ⟨index of clause or goal⟩;
get_clsname();
```

The statement `get_clsname();` is the short version of
`get_cgn(name,state,type,value,errcde);`

Information Returned to User Program

type	=	Type of name
	=	"LOGC" if index corresponds to a logic clause
	=	named by the user
	=	"GOAL" if index corresponds to a goal
	=	"INTL" if index corresponds to a logic clause with internal name (and thus is not accessible to user)
if type = "LOGC":		
name	=	Name of clause
state	=	"A" if clause is active
	=	"D" if clause is deleted
if type = "LOGC" or "INTL":		
value[1]	=	Likelihood level of clause
if type ="GOAL":		
name	=	Name of goal
state	=	"A" if goal is active
	=	"D" if goal is deleted
errcde[0]	=	Error code (see Error Messages below)

get_dimension

Description

Get dimensions for COLMAX, ROWMAX, ANZMAX, BLKMAX, PRB-MAX, and STOMAX, to be used in subsequent memory allocation.

Overview

The procedure reads the preamble of a specified .prg file and extracts the parameter values, to be used for memory allocation by assign_device or initialize_problem. The procedure can be called repeatedly to find correct parameter values for a collection of .prg files.

Execution from User Program

The variable value[0] must be set to 0 for the first call so that internal variables of get_dimension are properly initialized. Each call of get_dimension increments value[0] by 1.

If just one .prg file is to processed:

```
value[0] = 0;
strcpy(name,"⟨.prg file name⟩");
get_dimension();
```

If several .prg files are to be processed:

Requires the use of PRGNAME_ARRAY of leibnizmacro.h to define an array prgname[] of appropriate size. It is assumed that the .prg file names have been stored in prgname[] and that the empty string is the last .prg file name.

```
value[0] = 0;
while (strcmp(prgname[value[0]],"") != 0) {
  strcpy(name,prgname[value[0]]);
  get_dimension();
}
```

The statement `get_dimension();` is the short version of
`get_dim(name,state,type,value,errcde);`

Information Returned to User Program

Each call of get_dimension processes one .prg file. Such a call produces the following output, where COLMAX, ROWMAX, ANZMAX, BLKMAX, and STOSIZ refer to the min values of the .prg file.

value[0]	=	input value[0] + 1
value[5]	=	input value[0] + 2
	=	correct PRBMAX value
if input value[0] = 0:		
value[1]	=	COLMAX
value[2]	=	ROWMAX
value[3]	=	ANZMAX
value[4]	=	BLKMAX
value[6]	=	STOSIZ
if input value[0] > 0:		
value[1]	=	max of input value[1] and COLMAX

value[2]	=	max of input value[2] and ROWMAX
value[3]	=	max of input value[3] and ANZMAX
value[4]	=	max of input value[4] and BLKMAX
value[6]	=	input value[6] + STOSIZ of .prg file
	=	correct STOMAX value
errcde[0]	=	Error code (see Error Messages below)

get_dual

Description

Get dual value for clause or goal with specified index.

Overview

When a minimization problem has been solved by approximate minimization, then the procedure returns for each named clause and goal a so-called dual value. Dual values may be used to evaluate the importance of a clause or goal. Generally, the higher the dual value, the more important is the clause or goal. There are additional uses of dual values for the solution of huge logic minimization problems, with possibly hundreds of thousands of variables. For details about such methods, the user should contact the *Leibniz Company*.

For each index, the procedure returns the dual value if that index corresponds to a clause named by the user, or to a goal. For clauses not named by the user, the returned value is 0.

The procedure indicates whether the clause or goal is active or deleted. If the index corresponds to a clause, the procedure also returns the likelihood level. For interpretation of that information, see status_clause.

CAUTION: get_dual may be invoked only if the current problem has been solved by approximate minimization and has been found to be satisfiable.

CAUTION: get_clsname and get_dual are the only procedures that utilize indices in connection with clauses and goals. All other procedures involving clauses or goals must use names.

Execution from User Program

```
value[0] = ⟨index of clause or goal⟩;
get_dual();
```

The statement `get_dual();` is the short version of
`get_dul(name,state,type,value,errcde);`

Information Returned to User Program

type	=	Type of name
	=	"LOGC" if index corresponds to a logic clause named by the user
	=	"GOAL" if index corresponds to a goal
	=	"INTL" if index corresponds to a logic clause with internal name (and thus is not accessible to user)
value[2]	=	Dual value
	=	Dual value of clause if type = "LOGC"
	=	Dual value of goal if type = "GOAL"
	=	0 if type = "INTL"

if type = "LOGC":

name	=	Name of clause
state	=	"A" if clause is active
	=	"D" if clause is deleted

if type = "LOGC" or "INTL":

value[1]	=	Likelihood level of clause

if type ="GOAL":

name	=	Name of goal

state	=	"A" if goal is active
	=	"D" if goal is deleted
errcde[0]	=	Error code (see Error Messages below)

get_goal

Description

Get goal information for specified goal name: Goal quantity, cost of low usage, cost of high usage, indicator whether problem has been solved, and, if solved, goal usage.

Execution from User Program

```
strcpy(name, "⟨goal name⟩");
get_goal();
```

The statement `get_goal();` is the short version of
`get_gol(name,state,type,value,errcde);`

Information Returned to User Program

state	=	”A” if goal is active
	=	”D” if goal is deleted
value[0]	=	goal quantity
value[1]	=	unit cost of low usage
value[2]	=	unit cost of high usage
value[3]	=	solution indicator (approximate minimization)
	=	1 if problem solved and satisfiable
	=	0 else

if value[3] = 1 (problem solved and satisfiable):

value[4]	=	goal usage
errcde[0]	=	Error code (see Error Messages below)

For a summary discussion of the use of goals, see the section Goals and GOAL Variable in Chapter 3.

get_primal

Description

Get primal classification for specified variable.

Information may be requested by specifying the variable name, or by specifying the index of the variable. In the second case, the index must lie between 1 and the total number of variables. This feature allows looping. The total number of variables is obtained with status_problem.

Overview

Approximate minimization is carried out via the solution of certain linear programs (LPs). One may use approximate minimization iteratively to solve huge logic minimization problems, possibly with hundreds of thousands of variables. For such iterative applications of approximate minimization, one must know the classification of variables in one of the LPs. The procedure provides that classification. For details about iterative methods, the user should contact the *Leibniz Company*.

CAUTION: get_primal may be invoked only if the current problem has been solved by approximate minimization and has been found to be satisfiable.

Execution from User Program

either (using name):

```
  value[0] = 0;
  strcpy(name,"⟨variable name⟩");
```

or (using index):

```
  value[0] = ⟨index of variable⟩;
```

```
get_primal();
```

The statement `get_primal();` is the short version of
`get_pml(name,state,type,value,errcde);`

Information Returned to User Program

state	=	Primal classification of variable
	=	”B” if variable is in LP in-between True and False
	=	”T” if variable was fixed to True, or set to True in LP
	=	”F” if variable was fixed to False, or set to False in LP
	=	”D” if variable was deleted
value[2]	=	True cost
value[3]	=	False cost
if value[0] > 0:		
name	=	Name of variable with index = value[0]
errcde[0]	=	Error code (see Error Messages below)

get_value

Description

Get solution value for specified variable, and True/False cost in minimization case. Problem must have been solved and must be satisfiable.

Information may be requested by specifying the variable name, or by specifying the index of the variable. In the second case, the index must lie between 1 and the total number of variables. This feature allows looping. The total number of variables is obtained with status_problem.

CAUTION: The procedure cannot be executed if the transfer process is used.

Execution from User Program

either (using name):
```
  value[0] = 0;
  strcpy(name,"⟨variable name⟩");
```
or (using index):
```
  value[0] = ⟨index of variable⟩;
```
```
get_value();
```

The statement `get_value();` is the short version of
`get_val(name,state,type,value,errcde);`

Information Returned to User Program

state	=	Value of variable
	=	"T" if variable is True
	=	"F" if variable is False
	=	"D" if variable is deleted
value[2]	=	True cost if minimization case
value[3]	=	False cost if minimization case
errcde[0]	=	Error code (see Error Messages below)

if value[0] $>$ 0:

name	=	Name of variable with index = value[0]
errcde[0]	=	Error code (see Error Messages below)

initialize_problem

Description

Initialize any number of problems.

Overview

Determines parameters from specified .prg files for memory allocation using get_dimension. Allocates memory and assigns device for output of error messages using assign_device. Stores programs for the specified problems into memory using begin_problem and store_problem. Loads the program of the last problem of the list for execution and returns statistics about that problem using status_problem.

CAUTION: It is highly recommended that all .prg files have been compiled so that either all or none of the solution programs use the transfer process. If this rule is not adhered to, hard-to-find errors may occur.

CAUTION: If the transfer process is used for all files, the procedure does not initialize values of logic variables of the user program. Such changes must be made by the user code or by calling modify_allvar.

Error messages are recorded in a file whose name is specified by the user in the parameter name. The user may request that error messages are also written to the screen, by setting the parameter value[0] = 1. Any other assignment to value[0] causes writing of error messages to the screen to be suppressed.

CAUTION: The error file specified in the parameter name is opened by initialize_problem, and must not be opened by the user.

CAUTION: If the name of the error file specified in the parameter name is the empty string, the system assumes an inadvertent user error and substitutes the name execute.err.

CAUTION: The procedure initialize_problem resets limits on the number of warnings, nonfatal errors, and fatal errors to the default values. See limit_error for these default values. To change these values, execute limit_error immediately after initialize_problem.

Execution from User Program

Requires the use of PRGNAME_ARRAY of leibnizmacro.h to define an array prgname[] of appropriate size. It is assumed that the .prg file names have been stored in prgname[] and that the empty string is the last .prg file name.

```
strcpy(name,⟨error file name⟩);
value[0] = ⟨flag for writing error messages on screen⟩
           (= 1: write error messages on screen)
           (≠ 1: suppress error messages on screen)
initialize_problem();
```

The statement `initialize_problem();` is the short version of
`inz_prb(prgname,name,state,type,value,errcde);`

CAUTION: The long version does not have the standard form, since it contains the additional array prgname.

Information Returned to User Program

The problem stored last is also loaded for execution. For that problem, the following information is returned.

name	=	Name of current problem
value[0]	=	Number of variables
value[1]	=	Number of clauses and goals
if minimization problem and solved:		
value[2]	=	Total cost
value[3]	=	Lower bound on total cost
value[4]	=	Number of goals
if transfer process is used:		
state	=	”” (empty string)
if transfer process is not used:		
state	=	”S” if problem solved and satisfiable
	=	”U” if problem solved and unsatisfiable
	=	”N” if problem not solved
type	=	”SAT” if satisfiability problem
	=	”MIN” if minimization problem
errcde(0)	=	Error code (see Error Messages below)

limit_error

Description

Define limits for total number of warnings, nonfatal errors, and fatal errors.

Overview

During execution, the number of warnings, nonfatal errors, and fatal errors is counted. When any one of these numbers exceeds a specified limit, the procedure leibnizerroraction is called. A template for that procedure is supplied in the file leibnizerroraction of Leibniz/Makedata. The procedure may be defined in any way desired by the user.

Default limits are defined by initialize_problem and assign_device as follows.

Error Type	Limit
Warning	0
Nonfatal Error	0
Fatal Error	0

CAUTION: Whenever initialize_problem or assign_device is executed, the limits are reset to the above default values. If other values are desired, they must be specified by limit_error after each call of initialize_problem or assign_device.

Execution from User Program

```
value[0] = ⟨limit for number of warnings⟩
value[1] = ⟨limit for number of nonfatal errors⟩
value[2] = ⟨limit for number of fatal errors⟩
limit_error();
```

The statement `limit_error();` is the short version of
`lim_err(name,state,type,value,errcde);`

Information Returned to User Program

errcde(0)	=	Error code (see Error Messages below)

maxcls_sat

Description

Find maximal subset of specified clauses for which problem is satisfiable.

Overview

Requires use of CLAUSE_ARRAY of leibnizmacro.h to define the structure clauses[] of appropriate size. The structure consists of the arrays clause[].name and clause[].status.

Any number of clause names may be specified in clause[].name. The empty string is stored as last clause name.

The procedure finds a maximal subset of the set of clauses named in clause[].name such that the problem remains satisfiable when the clauses of the subset are activated and the remaining clauses of the set are deleted. The decision of activation or deletion is recorded in clause[].status, with 1 denoting activation and 0 denoting deletion.

CAUTION: It is assumed that all desired problem modifications—fixing or deleting of variables, deleting of clauses with names not in the specified set—have been done prior to the call of maxcls_sat. However, any activation or deletion of any clause with name in the specified set prior to maxcls_sat is ignored.

CAUTION: If several clauses have the same name, then the clauses with that name are either all activated or all deleted by maxcls_sat.

CAUTION: At termination of maxcls_sat, the clauses with name of clause[i].name are activated if clause[i].status = 1 and are deleted if clause[i].status = 0.

In some applications, one would like to reset the state of all clauses after maxcls_sat to the state existing prior to maxcls_sat. One can achieve that effect by storing the problem prior to the call of maxcls_sat using store_problem, and by retrieving the problem after maxcls_sat using retrieve_problem.

Execution from User Program

Requires use of CLAUSE_ARRAY of leibnizmacro.h to define the structure clauses[] of appropriate size. It is assumed that the clause names have been stored in clause[].name and that the empty string is the last clause name.

```
maxcls_sat();
```

The statement `maxcls_sat();` is the short version of
`mxcls_sat(clause,name,state,type,value,errcde);`

CAUTION: The long version does not have the standard form, since it contains the additional structure clause.

Information Returned to User Program

clause[i].status	=	status of clauses with the name of clause[i].name
	=	0 if the clauses are deleted and hence are not in the maximal subset

	=	1 if the clauses are activated and hence are in the maximal subset
	=	−1 if there is no clause with the specified name
state	=	"S" if problem is satisfiable when the clauses with clause[i].status = 1 are activated and the clauses with clause[i].status = 0 are deleted
	=	"U" if problem still unsatisfiable when the clauses of the entire set are deleted
errcde(0)	=	Error code (see Error Messages below)

mincls_unsat

Description

Find minimal subset of specified clauses for which problem remains unsatisfiable.

Overview

Requires use of CLAUSE_ARRAY of leibnizmacro.h to define the structure clauses[] of appropriate size. The structure consists of the arrays clause[].name and clause[].status.

Any number of clause names may be specified in clause[].name. The empty string is stored as last clause name.

The procedure finds a minimal subset of the set of clauses named in clause[].name such that the problem remains unsatisfiable when the clauses of the subset are activated and the remaining clauses of the set are deleted. The decision of activation or deletion is recorded in clause[].status, with 1 denoting activation and 0 denoting deletion.

CAUTION: It is assumed that all desired problem modifications—fixing or deleting of variables, deleting of clauses with names not in the specified set—have been done prior to the call of mincls_unsat. However, any activation or deletion of any clause with name in the specified set prior to mincls_unsat is ignored.

CAUTION: If several clauses have the same name, then the clauses with that name are either all activated or all deleted by mincls_unsat.

CAUTION: At termination of mincls_unsat, the clauses with name of clause[i].name are activated if clause[i].status = 1 and are deleted if clause[i].status = 0.

In some applications, one would like to reset the state of all clauses after mincls_unsat to the state existing prior to mincls_unsat. One can achieve that effect by storing the problem prior to the call of mincls_unsat using store_problem, and by retrieving the problem after mincls_unsat using retrieve_problem.

Execution from User Program

Requires use of CLAUSE_ARRAY of leibnizmacro.h to define the structure clauses[] of appropriate size. It is assumed that the clause names have been stored in clause[].name and that the empty string is the last clause name.

```
mincls_unsat();
```

The statement `mincls_unsat();` is the short version of
`mncls_unsat(clause,name,state,type,value,errcde);`

CAUTION: The long version does not have the standard form, since it contains the additional structure clause.

Information Returned to User Program

clause[i].status	=	status of clauses with the name of clause[i].name
	=	0 if the clauses are deleted and hence are not in the minimal subset

	=	1 if the clauses are activated and hence are in the minimal subset
	=	−1 if there is no clause with the specified name
state	=	"U" if problem is unsatisfiable when the clauses with clause[i].status = 1 are activated and the clauses with clause[i].status = 0 are deleted
	=	"S" if problem still satisfiable when the clauses of the entire set are activated
errcde(0)	=	Error code (see Error Messages below)

modify_allvar

Description

Modify status of all logic variables in the user program. The procedure requires use of the transfer process.

<u>CAUTION</u>: This procedure and solve_problem are the only procedures that modify values of logic variables in the user program. The changes are restricted to the variables of the solution program that is currently loaded.

Overview

The file leibnizexts.h included in the user program defines predicate variables and propositional variables of a .log file. The procedure modify_allvar may be used to assign a specified value to these logic variables. The value depends on the string assigned to state.

state	Value
"T"	3 (= True)
"F"	−3 (= False)
"D"	0 (= Deleted)
"A"	−1 (= Active)

There are two other possible string assignments for state that do not change the values assigned to the logic variables in the user program, but that store or remove starting solutions for approximate minimization.

State = "I": The values currently assigned to the logic variables are saved as initial solution values for approximate minimization. This option is useful if one has already obtained good solution values and wants the approximate minimization to start with those values. For details of that use, see solve_problem.

state = "R": Any initial solution saved previously for approximate minimization is removed. The same effect can be achieved using restore_problem with state = "I".

CAUTION: One may execute modify_allvar with state = "I" or "R" even if the currently loaded program does not call for approximate minimization. In that case, no warning or error message is produced. The reason is that the user may employ at one time exact minimization and at another time approximate minimization for the same logic formulation.

Execution from User Program

```
strcpy(state, ⟨string⟩);
where ⟨string⟩ is:
     = "T" - Assign 3 (= True) to all variables
     = "F" - Assign −3 (= False) to all variables
     = "D" - Assign 0 (= Deleted) to all variables
     = "A" - Assign −1 (= Activated) to all variables
     = "I" - Save current values as initial solution
               for approximate minimization
     = "R" - Remove any initial solution for
               for approximate minimization
modify_allvar();
```

The statement `modify_allvar();` is the short version of
`mst_allvar(name,state,type,value,errcde);`

Information Returned to User Program

errcde(0)	=	Error code (see Error Messages below)

modify_clause

Description

Modify status of specified clauses. Specifically, declare all clauses matching the name given in the name parameter and having likelihood level within specified bounds, to be active or deleted, or change likelihood level.

Overview

A fact of the .log file may produce a number of clauses. If a name has been assigned to the fact in the .log file, then all clauses have that same name. Similarly, if the fact specifies a likelihood level—using VERY FREQUENTLY, FREQUENTLY, .. , RARELY, or AT LEVEL n, with $1 \leq n \leq 100$—then all clauses of that fact receive that likelihood level. If the likelihood term is AT LEVEL n, then the likelihood level is n. For the other likelihood terms, the corresponding levels are as follows:

Likelihood Keyword used in .log file	Likelihood Level
VERY FREQUENTLY	85
FREQUENTLY	65
HALF THE TIME	50
SOMETIMES	25
RARELY	10

<u>CAUTION</u>: If no likelihood is assigned to a fact, then the fact is considered to be always valid, as one would expect. Formally, the lbcc compiler assigns to such a fact a likelihood level of 0. The value 0

does not imply that the clause is unlikely to hold. Indeed, the clause is assumed to be always valid. The value 0 signifies only that the user has not explicitly assigned a likelihood value to the corresponding fact of the .log file. Thus, the user may not change the value 0 at execution time to some positive value.

The procedure modify_clause allows one to delete, activate, or change the likelihood level of clauses, by referencing them by name and/or by likelihood level. The basic rules for such changes are as follows.

1. If the clause has a name, but does not have an explicitly assigned likelihood level, then one may delete or activate such a clause, but may not change the implicit likelihood level of 0.

2. If the clause has an explicitly assigned likelihood level, but does not have a name, then one may delete or activate the clause, or may change its likelihood level, by referencing the current likelihood level. Of course, all clauses with the same likelihood level have their level changed as well.

3. If the clause has both a name and an explicitly assigned likelihood level, then one may delete or activate that clause, or may change its likelihood level, by referencing the clause by name and/or by likelihood level.

See the discussion at the end of this procedure for the use of likelihood levels.

Execution from User Program

```
strcpy(name,⟨name string, wildcard ? allowed⟩);
```

The name is allowed to be the empty string; that is, name = "". In that case, the name specification is considered to be absent, and the selection of clauses for status change is solely based on the likelihood level interval. That interval is specified as follows.

`value[0] =` ⟨lowest likelihood level to be considered⟩`;`
`value[1] =` ⟨highest likelihood value to be considered⟩`;`
where $0 \leq \text{value}[0] \leq \text{value}[1] \leq 100$

`strcpy(state,` ⟨string⟩`);`
where ⟨string⟩ is:
= "D" - Delete all matched clauses
= "A" - Activate all matched clauses
= "L" - Change likelihood level of all matched clauses

if state = "L":
`value[4] =` ⟨new likelihood level⟩
where $1 \leq \text{value}[0]$ and $1 \leq \text{value}[4] \leq 100$

`modify_clause();`

The line `modify_clause();` is the short version of
`mst_cls(name,state,type,value,errcde);`

Example:

```
strcpy(name,"fail??");
value[0] = 30;
value[1] = 78;
strcpy(state,"L");
value[4] = 55;
modify_clause();
```

In this case, all clauses with name starting with “fail” and with two additional letters following “fail”, and with likelihood level between 30 and 78, are to receive the new likelihood level 55.

Information Returned to User Program

value[2]	=	Number of matching clauses
value[3]	=	Number of clauses modified
errcde[0]	=	Error code (see Error Messages below)

Use of Likelihood levels

Suppose we want to disregard all facts that were specified in the input with a likelihood lying between 1 and 45. Put differently, we want to consider clauses with likelihood level equal to 0 or with value between 46 and 100. According to the axioms of Fuzzy Logic, any conclusion then drawn is valid at the likelihood level of 46.

The following statements achieve the deletion of all clauses with likelihood level between 1 and 45:

```
strcpy(name,"");
value[0] = 1;
value[1] = 45;
strcpy(state,"D");
modify_clause;
```

CAUTION: The activation or deletion of clauses must be carefully managed. For example, if one works with likelihood levels and also activates or deletes clauses for other reasons, then the calls of modify_clause must be carefully sequenced so that the intended effect is achieved. Complicated activation/deletion sequences should and can be avoided altogether, as follows. In a first step, one activates all clauses using restore_problem. Then one deletes appropriate clauses with modify_clause. This approach is far preferable to one where one activates some clauses, then deletes others, all in the mistaken notion of reducing the total effort for such activation/deletion steps. Actually, the conceptually simpler approach, where one first activates all clauses, then deletes some as needed, is computationally about as fast as more sophisticated activation/deletion sequences.

modify_goal

Description

Modify goal quantities or costs of low/high usage for all goals matching the name given in the name parameter, or declare all goals matching that name to be active or deleted.

Overview

In certain logic formulations involving minimization, one may want to include goals. For example, one may desire that a given resource be used up, or that usage of a resource observe a specified upper limit. Such goals can be included in the logic formulation if each logic variable occurs in at most one such goal, and if the logic formulation is solved by approximate minimization. Details are discussed in the section Goals and GOAL Variable in Chapter 3. That chapter includes an example case.

In general, the logic formulation in the .log file includes the names of goals, but does not specify the goal quantities or the costs of low/high usage. The procedure modify_goal is used to insert these data into the problem.

In addition, modify_goal allows one to delete or activate goals.

<u>CAUTION</u>: The cost of low usage is the cost incurred for each unit of usage below the goal quantity. Analogously, the cost of high usage is the cost incurred for each unit of usage above the goal quantity. If one wants to prevent low or high usage altogether, then one should specify large cost of low or high usage.

CAUTION: The goal quantity must lie in the range $-16,383$ .. $16,383$. The cost for low or high usage must lie in the range 0 .. $16,383$

CAUTION: Goal quantities, costs of low and high usage, as well as goal coefficients specified by modify_variable, must be so selected that the total cost of any solution cannot exceed 2,000,000,000.

Execution from User Program

```
strcpy(name,⟨name string, wildcard ? allowed⟩);

strcpy(state, ⟨string⟩);
where ⟨string⟩ is:
    = "D" - Delete all matched goals
    = "A" - Activate all matched goals
    = "Q" - Change goal quantity of all matches
    = "L" - Change cost of low usage of all matches
    = "H" - Change cost of high usage of all matches
    = "P" - Change goal quantity and cost of low/high
            usage of all matches

if state = "Q":
  value[0] = ⟨new goal quantity⟩
if state = "L":
  value[1] = ⟨new cost of low usage⟩
if state = "H":
  value[2] = ⟨new cost of high usage⟩
if state = "P":
  value[0] = ⟨new goal quantity⟩
  value[1] = ⟨new cost of low usage⟩
  value[2] = ⟨new cost of high usage⟩

modify_goal);
```

The statement `modify_goal();` is the short version of
`mst_gol(name,state,type,value,errcde);`

Example:

```
strcpy(name,"budget");
strcpy(state,"P");
value[0] = 21;
value[1] = 0;
value[2] = 45;
modify_goal);
```

The example defines the goal quantity of the goal “budget” to be 21, the cost of low usage to be 0, and the cost of high usage to be 45.

Information Returned to User Program

value[3]	=	Number of matching goals
value[4]	=	Number of goals modified
errcde[0]	=	Error code (see Error Messages below)

modify_problem

Description

Modify current problem from minimization problem to satisfiability problem or conversely.

CAUTION: A change of problem type, that is, from satisfiability to minimization or vice versa, is allowed only if the original formulation has specified minimization by MINIMIZE as first record in the input .log file.

Execution from User Program

```
strcpy(type,⟨new type⟩);
where ⟨new type⟩ is:
      = "MIN" if change to minimization
      = "SAT" if change to satisfiability
modify_problem();
```

The statement `modify_problem();` is the short version of
`mst_prb(name,state,type,value,errcde);`

Information Returned to User Program

value[0]	=	Number of variables
value[1]	=	Number of clauses and goals
errcde(0)	=	Error code (see Error Messages below)

modify_variable

Description

Modify the status or costs of all variables matching the given name, or of just one variable with specified index. In the first case, the given name may contain the wildcard character "?". The procedure is restricted if the transfer process is used.

Overview

CAUTION: If the transfer process is used, only the costs and goal coefficients of variables may be changed. Not allowed are fixing, deleting, and activating of variables, and assignment of initial values for approximate minimization.

CAUTION: Changes of cost values are restricted depending on the original costs specified by TRUECOST and FALSECOST in the input .log file. The rules are as follows:
IF original TRUECOST $>$ original FALSECOST, then new TRUECOST must be $\geq$ new FALSECOST.
IF original TRUECOST $=$ original FALSECOST, then new TRUECOST must be $=$ new FALSECOST.
IF original TRUECOST $<$ original FALSECOST, then new TRUECOST must be $\leq$ new FALSECOST.
In all cases, the new costs must lie in the range $-16,383$ .. $16,383$.

CAUTION: The status of matching variables is changed in the expected way, except if state[0] = "A" and if one of the following situations applies:

If the variable has the ASSIGNTRUE option, or if the lbcc compiler declared the variable to be monotone with preferred value True, then the variable is fixed to True.

If the variable has the ASSIGNFALSE option, or if the lbcc compiler declared the variable to be monotone with preferred value False, then the variable is fixed to False.

CAUTION: If the goal usage coefficient is changed, then the new value must lie in the range −128 .. 128.

Execution from User Program

```
either (using name):
  value[0] = 0;
  strcpy(name,"⟨nonempty name, wildcard ? allowed⟩");
or (using index):
  value[0] = ⟨index of variable⟩;
strcpy(state,⟨string⟩);
where ⟨string⟩ is restricted if transfer process is used
  if transfer process is not used:
      ⟨string⟩ = "T" - Fix all matched variables to True
               = "F" - Fix all matched variables to False
               = "D" - Delete all matched variables
               = "A" - Activate all matched variables
               = "I"  - Change initial values for approximate minimization
  with or without transfer process:
      ⟨string⟩ = "M"- Change costs of all matched variables
               = "G" - Change goal name and usage coefficient
                       of all matched variables
if state = "M":
  value[4] = ⟨new TRUECOST⟩;
```

```
    value[5] = ⟨new FALSECOST⟩;

  if state = "G":
    strcpy(&name[60],"⟨new goal name⟩");
    value[6] = ⟨new usage coefficient⟩; ( = 0 means variable dropped
                    from goal constraint)

  if state = "I":
    type = ⟨initial value code⟩;
    where ⟨initial value code⟩ = "IT" if initial value is True
                               = "IF" if initial value is False
                               = "IN" if no initial value assigned

  modify_variable();
```

The statement `modify_variable();` is the short version of
`mst_var(name,state,type,value,errcde);`

Information Returned to User Program

value[2]	=	Number of matching variables
value[3]	=	Number of variables modified
errcde[0]	=	Error code (see Error Messages below)

if value[0] > 0:

name	=	Name of variable with index = value[0]

restore_problem

Description

Restore a portion of the current problem or the entire current problem to original values, that is, to the values on hand immediately after the loading of the problem by begin_problem.

<u>CAUTION</u>: If the transfer process is used, the procedure does not change values of logic variables of the user program. Such changes must be made by the user code or by calling modify_allvar.

<u>CAUTION</u>: The procedure restore_problem does not restore goal quantities, low/high goal usage costs, or goal usage coefficients. If these data are to be changed, modify_goal and/or modify_variable must be used.

Execution from User Program

```
strcpy(state,⟨string⟩)
where ⟨string⟩ is:
      = "A" - Restore entire problem
      = "M" - Restore costs of variables
      = "C" - Restore clauses and goals to active
      = "L" - Restore likelihood levels of clauses
      = "T" - Restore problem type (minimization or
                satisfiability)
      = "I"  - Restore initial values for approx. min. to
                'no value assigned'. If the transfer process
                is used, the same result can be achieved by
                modify_allvar with state = "R"
```

if transfer process is not used:
= "V" - Restore variables to active and default values

```
restore_problem();
```

The statement `restore_problem();` is the short version of
`rst_prb(name,state,type,value,errcde);`

Information Returned to User Program

errcde[0] = Error code (see Error Messages below)

retrieve_problem

Description

Retrieve a stored problem and load as current problem. The problem also remains in storage.

CAUTION: If the transfer process is used, then the values of logic variables in the user program are not affected by the retrieval of the problem. Indeed, modify_allvar and solve_problem are the only procedures that modify values of logic variables in the user program.

Execution from User Program

```
strcpy(name,"⟨name of problem to be retrieved⟩");
retrieve_problem();
```

The statement `retrieve_problem();` is the short version of
`ret_prb(name,state,type,value,errcde);`

Information Returned to User Program

value[0]	=	Number of variables
value[1]	=	Number of clauses and goals

if transfer process is used:

state	=	”” (empty string)

if transfer process is not used:

state	=	"S" if problem solved and satisfiable
	=	"U" if problem solved and unsatisfiable
	=	"N" if problem not solved
type	=	"SAT" if satisfiability problem
	=	"MIN" if minimization problem

if minimization problem and solved:

value[2]	=	Total cost
value[3]	=	Lower bound on total cost
errcde(0)	=	Error code (see Error Messages below)

solve_problem

Description

Solve current satisfiability or minimization problem and return outcome.

CAUTION: This procedure and modify_allvar are the only procedures that modify values of logic variables in the user program. The changes are restricted to the variables of the solution program that is currently loaded.

Overview

Special consideration apply when the problem is solved by approximate minimization:

1. The input must specify the precision level at which approximate minimization is to be carried out. The precision level must be in the range 1 .. 20.

2. For small problems, with up to 600 variables and up to 400 clauses and goals, the precision level should be in the range 16 .. 20. Higher levels require more run time, but also tend to produce better solution. Typically, level 17 or 18 produces good results with reasonable run times.

3. For large problems, the precision level should be in the range 11 .. 15. Here, too, higher levels require more run time, but also tend to produce better solutions. Typically, level 12 or 13 produces good results with reasonable run times.

4. Precision levels 1 .. 5 and 6 .. 10 may be used as substitutes for 11 .. 15. That is, one may substitute 1 or 6 for 11, 2 or 7 for 12, 3 or 8 for 13, etc. The run times for levels 1 .. 5 and 6 .. 10 are shorter than for 11 .. 15, but solution quality is usually inferior. However, there are exceptional cases where levels 1 .. 5 and 6 .. 10 are fast and produce very good solutions.

5. For very fast approximate minimization, select 1 or 6 as precision level.

6. Initial solution values may be specified by prior calls of modify_allvar and modify_variable. Such initial values can significantly speed up the solution process.

CAUTION: If transfer process is used, initial solution values can be assigned only by modify_allvar.

CAUTION: If transfer process is not used, initial solution values can only be assigned by modify_variable; and if a variable has been deleted or fixed to True or False, then any initial value specified with modify_variable is ignored. That is, deletion and fixing of variables override assignment of initial values.

7. For approximate minimization involving large problems, the user may elect to have progress statements printed on the screen while the approximate minimization method solves the problem. The progress statements are printed if the detail flag mentioned below is defined as "D".

8. The approximate minimization scheme not only produces a good solution, but also a lower bound on the total cost of an optimal solution. That is, there cannot possibly be a solution to the problem where total cost is below that lower bound. This feature allows one to judge the quality of the solution produced by approximate minimization.

Execution from User Program

```
If approximate minimization:
   value[4] = ⟨precision level⟩;
   strcpy(type, "⟨detail flag⟩");
   where
      detail flag = ”D” - Display progress
                  = ”N” - Skip display of progress

solve_problem();
```

The statement `solve_problem();` is the short version of
`sol_prb(name,state,type,value,errcde);`

Information Returned to User Program

If transfer process is used, the solution values are inserted into the logic variables of the user program.

type	=	”SAT” if satisfiability problem
	=	”MIN” if minimization problem
state	=	”S” if problem satisfiable
	=	”U” if problem unsatisfiable
if minimization case:		
value[2]	=	Total cost of solution
if minimization case and solved by approx. min.:		
value[3]	=	Lower bound on total cost
errcde[0]	=	Error code (see Error Messages below)

status_clause

Description

Return status of the deletable clauses with specified name and with likelihood level within specified bounds. A clause is deletable if the fact of the .log input file producing it contains one of the likelihood terms VERY FREQUENTLY, FREQUENTLY, .. RARELY, AT LEVEL n, or if the fact has a name.

Overview

A fact of the .log file may produce a number of clauses. If a name has been assigned to the fact in the .log file, then all clauses have that same name. Similarly, if the fact specifies a likelihood level—using VERY FREQUENTLY, FREQUENTLY, .. , RARELY, or AT LEVEL n, with $1 \leq n \leq 100$)—then all clauses of that fact receive the likelihood level. If the likelihood term is AT LEVEL n, then the likelihood level is n. For the other likelihood terms, the corresponding levels are as follows:

Likelihood Keyword used in .log file	Likelihood Level
VERY FREQUENTLY	85
FREQUENTLY	65
HALF THE TIME	50
SOMETIMES	25
RARELY	10

CAUTION: If no likelihood is assigned to a fact, then the fact is considered to be always valid, as one would expect. Formally, the lbcc compiler assigns to such a fact a likelihood level of 0. The value 0 does not imply that the clause is unlikely to hold. Indeed, the clause is assumed to be always be valid. The value 0 signifies only that the user has not explicitly assigned a likelihood value to the corresponding fact of the .log file.

Execution from User Program

`strcpy(name,`⟨name string, wildcard ? allowed⟩`);`

The name is allowed to be the empty string; that is, name = "". In that case, the name specification is considered to be absent, and the counting of clauses is solely based on the likelihood level interval. That interval is specified as follows.

`value[0] =` ⟨lowest likelihood level to be considered⟩`;`
`value[1] =` ⟨highest likelihood value to be considered⟩`;`
where $0 \leq \text{value}[0] \leq \text{value}[1] \leq 100$

`status_clause();`

The statement `status_clause();` is the short version of
`sts_cls(name,state,type,value,errcde);`

Information Returned to User Program

state	=	"A" if all matching clauses are active
	=	"D" if all matching clauses are deleted
	=	"B" if some matching clauses are active and others are deleted
value[2]	=	Number of matching clauses

value[3]	=	Number of matching active clauses
value[4]	=	Number of matching deleted clauses
errcde[0]	=	Error code (see Error Messages below)

status_element

Description

Return total number of elements specified in input .log file. In addition, return the index for a specified element name, or the element name for a specified index.

<u>CAUTION</u>: The element name must be specified without the wildcard character “?”.

<u>CAUTION</u>: This procedure can be executed only if the transfer process is used.

Execution from User Program

either (to get total number of elements only):
```
value[0] = -1;
```
or (to get index of element name and total number of elements):
```
value[0] = 0;
strcpy(name,"⟨element name⟩");
```
or (to get element name and total number of elements):
```
value[0] = ⟨index of element⟩;
```
```
status_element();
```

The statement `status_element();` is the short version of
`sts_elt(name,state,type,value,errcde);`

Information Returned to User Program

value[2]	=	Total number of elements in .log file

(This is the only return if value[0] $<$ 0)

if value[0] = 0:

value[1]	=	Index of specified element name

if value[0] $>$ 0:

name	=	Name of element with index = value[0]
value[1]	=	value[0]

status_problem

Description

Return status information for current problem.

Execution from User Program

```
status_problem();
```

The statement `status_problem();` is the short version of
`sts_prb(name,state,type,value,errcde);`

Information Returned to User Program

name	=	Name of current problem
value[0]	=	Number of variables
value[1]	=	Number of clauses and goals
value[4]	=	Number of goals
value[5]	=	Number of literals
type	=	”SAT” if satisfiability problem
	=	”MIN” if minimization problem

if transfer process is used:

state	=	”” (empty string)

if transfer process is not used:

state	=	”S” if problem solved and satisfiable

	=	"U" if problem solved and unsatisfiable
	=	"N" if problem not solved

if minimization problem and solved:

value[2]	=	Total cost
value[3]	=	Lower bound on total cost

errcde(0)	=	Error code (see Error Messages below)

status_storage

Description

Return status information for storage of problems, plus name of one stored problem for specified index.

CAUTION: As problems are added to or deleted from storage, the stored problem associated with a given index may vary. Thus, a given index may produce a certain name of a stored problem at one time, and a different name some time later since problems were added to or deleted from storage in the meantime.

Execution from User Program

```
value[0] = ⟨index of problem for which name is to be returned⟩;
           use value[0] = 1 for name of current problem
status_storage();
```

The statement `status_storage();` is the short version of
`sts_sto(name,state,type,value,errcde);`

Information Returned to User Program

value[1]	=	Total number of problems (stored plus current one)
value[2]	=	Total number of records used for storage
value[3]	=	Total number of records still available for storage

if value[0] = 1:		
name	=	name of currently loaded problem
value[4]	=	0
if value[0] > 1:		
name	=	name of stored problem with index = value[0]
value[4]	=	number of records used for storing problem with index = value[0]
errcde(0)	=	Error code (see Error Messages below)

status_variable

Description

Return status, type, and cost of True/False in minimization case, for specified variable. Information may be requested by specifying either the variable name or the index of the variable. The index must lie between 1 and the total number of variables. This feature allows looping.

CAUTION: The variable name must be specified without the wildcard character “?”.

Execution from User Program

either (using name):
```
  value[0] = 0;
  strcpy(name,"⟨variable name⟩");
```
or (using index):
```
  value[0] = ⟨index of variable⟩;
```
```
status_variable();
```

The statement `status_variable();` is the short version of
`sts_var(name,state,type,value,errcde);`

Information Returned to User Program

if transfer process is used:					
state	=	””			
if transfer process is not used:					
	=	”T” if variable is fixed to True			
	=	”F” if variable is fixed to False			
	=	”D” if variable is deleted			
	=	”A” if variable is active			
type	=	”⟨part 1⟩⟨part 2⟩⟨part 3⟩”			
		where ⟨part 1⟩	=	D	if delete option
			=	blank	if no delete option
		⟨part 2⟩	=	T	if True is initial value for approx. min.
			=	F	if False is initial value for approx. min.
			=	N	if no initial value is assigned
		⟨part 3⟩	=	AT	if option ASSIGNTRUE
			=	AF	if option ASSIGNFALSE
			=	MT	if monotone, default True
			=	MF	if monotone, default False
			=	blank	if not AT,AF,MT,MF
value[2]	=	Cost of True if minimization case			
value[3]	=	Cost of False if minimization case			
errcde[0]	=	Error code (see Error Messages below)			
if value[0] > 0:					
name	=	Name of variable with index = value[0]			
if value[4] > 0:					
&name[60]	=	Name of goal			
value[4]	=	Goal usage coefficient			
value[5]	=	Number of literals of variable			

store_problem

Description

Store current problem.

Overview

The problem may be stored under the name of the current problem or under a new name. The stored problem includes all changes that are part of the current problem. For example, goal values such as goal quantity or costs of low and high usage, and changes of likelihood are all recorded with the stored problem. If the transfer process is not used, values assigned to variables are stored as well. When later the stored problem is retrieved with retrieve_problem, it is loaded as current problem with exactly these same values.

The current problem is not affected by this procedure.

CAUTION: If the transfer process is used, values assigned to logic variables in the user program are not stored with the problem.

CAUTION: If a stored problem exists that has the same name as the problem to be stored, then that stored problem will be overwritten in storage by the latter problem.

Execution from User Program

```
strcpy(name,"⟨problem name⟩");
    where problem name = name to be given the stored problem;
    use name = "" (empty string) if name of current problem
                    is to be assigned.

store_problem();
```

The statement `store_problem();` is the short version of
`sto_prb(name,state,type,value,errcde);`

Information Returned to User Program

name	=	Name of stored problem
state	=	Action taken
	=	"E" if stored problem with same name existed and was overwritten
	=	"N" if there was no stored problem with same name
value[1]	=	Total number of problems (stored plus current one)
value[2]	=	Total number of records used for storage
value[3]	=	Total number of records still available for storage
value[4]	=	Number of records used to store current problem
errcde(0)	=	Error code (see Error Messages below)

Error Messages

The execution procedures create warning and error messages plus an error code prefixed by "LBEX". This section contains detailed explanations for each such code. There are three types of warning/error messages:

Warning	The procedure encountered an unexpected situation. Depending on the situation, the procedure tries to take corrective action. Subsequent results are likely correct, but could also be erroneous.
Nonfatal Error	The procedure detected an error that, by itself, does not preclude continuation of processing. However, subsequent results may be invalid.
Fatal Error	The procedure detected an error that precludes further processing by the procedure and almost surely causes errors in subsequently called procedures.

When a warning or error is encountered, it is displayed on the screen if that has been specified by assign_device or initialize_problem. The warning or error is also recorded in the error file defined in the call of assign_device or initialize_problem.

Warnings and errors may be conveniently monitored using limits on the number of warnings, nonfatal errors, and fatal errors. Default values for these limits are defined by assign_device or initialize_problem. The default values can be modified by limit_error.

When a limit is exceeded, the procedure leibnizerroraction is called, where the user can define remedial action. A template of that procedure is in Leibniz/Makedata. The default action of the template is to stop execution.

If all warnings, nonfatal errors, and fatal errors are to be monitored, the limits should be set to 0. Then any warning or error triggers a call of leibnizerroraction, where the user can specify appropriate action.

Warnings

LBEX0010: Problem was not loaded

An error was encountered upon reading of the specified .prg file.

LBEX0020: No matching name

No variable or clause name matches the name given in the array name.

LBEX0030: Variable may not be deleted: ⟨variable name⟩

An attempt was made to delete the specified variable for which the delete option has not been specified in the input .log file.

LBEX0040: Clause may not be deleted

An attempt was made to delete a clause that did not have the delete option included in the input .log file.

LBEX0050: Problem not solved

A procedure was executed which requires the problem to have been solved.

LBEX0060: Problem not satisfiable

A procedure was executed that requires the problem to be satisfiable for the requested information.

LBEX0070: Problem not solved by approx. min.

A procedure was executed that requires the problem to have been solved by approximate minimization.

Nonfatal Errors

LBEX1010: assign_device, begin_problem , or initialize_problem required

Except for get_dimension and limit_error, the first procedure called must be assign_device or initialize_problem. If assign_device is used, begin_problem must be the second procedure. A procedure was executed that is not consistent with these rules.

LBEX1020: Cannot open program file

The specified .prg file could not be opened. The path name provided in the array name must be the complete path of the file, including the file extension. If "\" occurs in the path, make sure that "\\" is used since "\" is the escape character of C.

LBEX1030: Unknown variable code

The following variable codes are allowed in modify_allvar and modify_variable:
"D" – delete variable
"A" – activate variable
"T" – set value to True
"F" – set value to False
"I" – change initial values for approx. min.

Additional code allowed in modify_allvar:
"R" – remove any existing initial values for approx. min.

Additional codes allowed in modify_variable:
"M" – change cost values
For case of "I", codes for initial values:
"IT" – change initial value to True
"IF" – change initial value to False
"IN" – no initial value assigned

LBEX1040: Index for variable out of bounds

An index was given for a variable which was not between 1 and the total number of variables. Use status_problem to determine the total number of variables.

LBEX1060: Unknown variable name

The specified variable name was not included in the input .log file.

LBEX1070: No matching clause

The given clause name plus the likelihood restriction do not match any clause.

LBEX1080: State and likelihood level incompatible

If the specified state is “L”, then the lower likelihood level bound must be at least 1, and the new likelihood level must lie in the range 1 .. 100.

LBEX1090: Unknown clause code

The following clause codes are allowed in modify_clause:
“D” – delete clause
“A” – activate clause
“L” – change likelihood level

LBEX1100: Improper likelihood level

A likelihood level has been specified outside the permitted range 1 .. 100.

LBEX1110: Likelihood levels not compatible

The specified lowest likelihood level is higher that the specified highest likelihood level.

LBEX1120: COLMAX parameter too small

The COLMAX parameter has been specified at a value less than 2, or the problem to be loaded requires a larger COLMAX parameter. The .prg file for the problem lists the minimum value required for COLMAX.
If devices have been assigned using assign_device, increase the COLMAX parameter for that call. If problems have been initialized with initialize_problem, then this error indicates that one cannot use initialize_problem for loading of problems, and must use assign_device, begin_problem, and store_problem instead..

LBEX1130: ROWMAX parameter too small

The ROWMAX parameter has been specified at a value less than 2, or the problem to be loaded requires a larger ROWMAX parameter. The .prg file for the problem lists the minimum value required for ROWMAX.
If devices have been assigned using assign_device, increase the ROWMAX parameter for that call. If problems have been initialized with initialize_problem, then this error indicates that one cannot use initialize_problem for loading of problems, and must use assign_device, begin_problem, and store_problem instead.

LBEX1140: ANZMAX parameter too small

The ANZMAX parameter has been specified at a value less than 2, or the problem to be loaded requires a larger ANZMAX parameter. The .prg file for the problem lists the minimum value required for ANZMAX.
If devices have been assigned using assign_device, increase the ANZMAX parameter for that call. If problems have been initialized with initialize_problem, then this error indicates that one cannot use initialize_problem for loading of problems, and must use assign_device, begin_problem, and store_problem instead.

LBEX1150: BLKMAX parameter too small

The BLKMAX parameter has been specified at a value less than 2, or the problem to be loaded requires a larger BLKMAX parameter. The .prg file for the problem lists the minimum value required for BLKMAX.
If devices have been assigned using assign_device, increase the BLKMAX parameter for that call. If problems have been initialized with initialize_problem, then this error indicates that one cannot use initialize_problem for loading of problems, and must use assign_device, begin_problem, and store_problem instead.

LBEX1160: Improper cost value

Cost values specified with modify_variable must lie in the range $-16,383$ .. $16,383$.

LBEX1170: Switch to minimization not allowed

A problem may be switched from satisfiability to minimization only if the original formulation in the input .log file specified minimization by the keyword MINIMIZE in the first record.

LBEX1180: Cost change not allowed

The change to the new costs for True and False given by value[4] and value[5] is not allowed since one of the conditions below is violated:

IF original TRUECOST $>$ original FALSECOST, then new TRUECOST must be $\geq$ new FALSECOST.
IF original TRUECOST $=$ original FALSECOST, then new TRUECOST must be $=$ new FALSECOST.
IF original TRUECOST $<$ original FALSECOST, then new TRUECOST must be $\leq$ new FALSECOST.

The index of the variable for which the change is not allowed, is given in value[6].

<u>CAUTION</u>: Some cost changes may have been already made during execution of modify_variable before the error was detected for the variable indexed by value[6].

LBEX1190: Unknown code for restore_problem

The following codes are allowed for restore_problem:
"A" – restore entire problem
"V" – restore variables to active
or fix to default values
"M" – restore costs of variables
"C" – restore clauses to active
"L" – restore likelihood level of clauses
"T" – restore problem type
(minimization or satisfiability)

LBEX1200: Improper precision level

The procedure solve_problem requires specification of the precision level in value[4] when approximate minimization is to be done. The allowed range for the level is 1 .. 20, but the number specified in value[4] is outside that range. The procedure remedies this by assigning value[4] = 18, which is a reasonable choice.

LBEX1210: Index for clause/goal out of bounds

An index was given that was not between 1 and the total number of clauses and goals. Use status_problem to obtain the total number of clauses and goals.

LBEX1220: PRBMAX parameter too small

The PRBMAX parameter has been specified at a value less than 2, or storage of more than PRBMAX problems has been attempted.

If devices have been assigned using assign_device, increase the PRBMAX parameter for that call. If problems have been initialized with initialize_problem, then this error indicates that one cannot use initialize_problem for loading of problems, and must use assign_device, begin_problem, and store_problem instead.

LBEX1230: STOMAX parameter too small

The STOMAX parameter has been specified at a value less than 2, or storage of more than STOMAX records has been attempted. Each .prg file lists under STOSIZ the number of records required to store the problem. STOMAX must be at least the sum of the STOSIZ values of the problems to be stored.

If devices have been assigned using assign_device, increase the STOMAX parameter for that call. If problems have been initialized with initialize_problem, then this error indicates that one cannot use initialize_problem for loading of problems, and must use assign_device, begin_problem, and store_problem instead.

LBEX1240: Problem cannot be stored

The currently loaded problem cannot be stored. Concurrent message LBEX1220 or LBEX1230 gives reason and required changes.

LBEX1250: Unknown problem name

There is no stored problem with the specified name, so the procedure specifying the problem name is ignored. Use status_storage to get the names of all stored problems.

LBEX1260: Index for problem out of bounds

A problem index was given that was not between 1 and the total number of problems. Use status_storage to determine the total number of problems.

LBEX1270: There is no stored problem

Retrieval or deletion of a problem has been requested, but there is no stored problem.

LBEX1280: Unknown goal name

The specified goal name was not included in the input .log file in a fact statement of the form GOAL. NAME ⟨goal name⟩. See the section Goals and GOAL Variable for details about the specification and use of goals.

LBEX1290: Unknown goal code

The following goal codes are allowed for modify_goal:

"D"	– delete goal
"A"	– activate goal
"Q"	– change goal quantity
"L"	– change cost of low usage
"H"	– change cost of high usage
"P"	– change goal quantity, cost of low usage, and cost of high usage

LBEX1310: Goal name missing

The string given in the name parameter as goal, starting in position name[0] or name[60], must be nonempty.

LBEX1320: Unknown goal name

The string given in the name parameter, starting in position name [0] or name[60], is not a goal name.

LBEX1330: Procedure requires approx. min.

The specified procedure, or the option used with the specified procedure, introduces, changes, or displays goal data. Goal data may be used only if the current problem involves approximate minimization. However, the current problem does not specify approximate minimization, so the procedure has not been executed. To utilize goals, compile the .log file again while requesting approximate minimization in the params.dat file.

LBEX1340: Improper goal usage coefficient

Usage coefficients for goals must lie in the range -128 .. 128.

LBEX1350: Improper goal quantity

Goal quantities must lie in the range $-16,383$ .. $16,383$.

LBEX1360: Improper goal cost value

Goal costs must lie in the range 0 .. $16,383$.

LBEX1370: assign_device or initialize_problem has already been executed

assign_device or initialize_problem has been requested for device allocation, but has already been executed. If different device allocation is desired, free_device must first be executed.

LBEX1380: assign_device or initialize_problem must be called prior to free_device

free_device has been requested for device de-allocation, but assign_device or initialize_problem was not previously executed. Hence, free_device cannot be executed.

LBEX1390: get_value cannot be used with transfer process

The currently loaded problem has been compiled with the transfer process option. Therefore, get_val cannot be used to obtain variable values.

LBEX1400: limit on warnings/errors is negative

At least one of the limits value[0]–value[2] specified for limit_error is negative. These limits must be nonnegative.

LBEX1410: modify_variable cannot fix, delete, or activate variables, or define initial solution

The currently loaded problem has been compiled with the transfer process option. Therefore, modify_variable is not allowed to fix, delete, or activate variables, or define initial solution values for approx. min. Hence, modify_variable cannot be invoked with state = "T", "F", "D", "A", or "I".

LBEX1420: mst_allvar requires transfer process

The procedure modify_allvar requires that the transfer process is used. Recompile problem with transfer process specified.

LBEX1430: prg file without parameter records

The prg file does not contain the header section with parameters COLMAX, ROWMAX, ANZMAX, BLKMAX, STOSIZ

LBEX1440: restore_problem with state = "V" cannot be used with transfer process

The currently loaded problem has been compiled with the transfer process option, and restore_problem with state = "V" cannot be used to revise values of variables in user program. The correct procedure for such changes is modify_allvar.

LBEX1450: Unknown element name

The specified element name was not included in the input .log file.

LBEX1460: Index for element out of bounds

An index was given for an element which was not between 1 and the total number of elements. The total number of elements is returned in value[2].

Fatal Errors

LBEX2010: Transfer process cannot handle current problem

The transfer process requires that each problem is represented by a file ⟨file name⟩.trs that is produced by the compiler. Moreover, the file leibniztrans.c must contain the statement
`#include"`⟨file name⟩`.trs"`
The error message occurs when ⟨file name⟩.trs has not been produced or has not been included in leibniztrans.c.

LBEX2020: Incompatible Leibniz System Versions

A .prg file produced under one *Leibniz System* version was to be executed using an incompatible *Leibniz System* version. In general, two versions a.x and b.y are compatible only if a = b. To correct the error, compile and execute programs using just one *Leibniz System* version.

LBEX2030: No elements specified

status_element (equivalently sts_elt) has been called to provide the index or name of an element, but the .log file used to create the files of the transfer process does not specify any elements.

System Errors

The execution procedures perform various compatibility checks and output an error message when an inconsistency is detected. An error message is of the form

```
*************LEIBNIZ SYSTEM ERROR***************
ERROR IN ⟨name⟩ LINE ⟨line number⟩
************************************************
```

The error is recorded in the error file defined by the call of assign_device or initialize_problem. In some cases, the system tries to recover from the error. When this is possible, processing continues, and the error message can be ignored. Thus, an error message is significant only if processing is halted due to the error.

Please save the error file, related .log and .prg files, and C code files, and report the error to the author. These files are needed for the analysis of the difficulty.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Chapter 5

Compiler Learning Process

Introduction

When a logic formulation becomes very large and complex, it may happen that the solution program produced by the lbcc compiler is somewhat slow. In most situations, the performance can be improved by requesting that the lbcc compiler perform a learning process in addition to the compilation of the solution program. This chapter covers that learning process.

Version 16.0 1 July 2015

Learn Option

The following section of the parameter file leibnizparams.dat controls learning.

```
*********************************************************
*** Learn option
* learn using file (filename or 'random') = random
max learning cycles = 100
output learned clauses
*********************************************************
```

The section as stated does not ask for learning. To request learning, one activates the line
`learn using file (filename or 'random') = random`
by removing the asterisk in column 1. There are two types of learning: User-directed learning and compiler-selected learning. They are invoked as follows.

User-directed Learning

The eventually created solution program invariably will solve problem instances derived from the given logic formulation by fixing some of the variables. To speed solution of these instances, the user collects typical cases in a file, say named cases.lrn. For example, suppose a case entails the following fixing of values. The variables x and y are fixed to True, and the variable z is fixed to False. That case is described by the following record in cases.lrn.

```
x y -z *
```

That is, if a variable is fixed to True, then the record simply contains the name of the variable. If a variable is fixed to False, the record contains the name of the variable prefixed with "-". Note that no blanks are allowed between "-" and the name. The end of the list of names, and thus of the case, is indicated by an asterisk. The file cases.lrn contains the cases consecutively, in any order. Each case is terminated by an asterisk. Several cases may occur in one record, and a single case may be spread over several records. That is, the end-of-record marks in the file are treated as white space, just as in C. The case of no fixing of variables can also be specified, by beginning cases.lrn with an asterisk.

The name of the file, which is cases.lrn in the example, is specified on the previously uncommented line as follows.
`learn using file (filename or 'random') = cases.lrn`

Compiler-selected Learning

If the user does not have typical cases of variable fixing available, the selection of such cases can be left to the compiler. This is done by just uncommenting the line
`learn using file (filename or 'random') = random`.

Essentially, the compiler makes up cases involving random selection of variables. This is done in a systematic way, by beginning with cases having few variables, and then gradually increasing the number of variables.

Limit on Learning Cases

The right hand side of the line `max learning cycles = 100` controls how many cases of learning are to be processed. That limit prevents a runaway situation where the compiler seemingly does not terminate. In the example, the limit is 100, a rather small number. Generally, a number up to 1000 is reasonable and prevents excessive run times for the learning process.

Output

The learning process is too complicated to be described in this manual. Roughly speaking, the compiler repeatedly runs an initially compiled solution program, checks during which steps the program uses excessive time, adds some logic expressions to the logic formulation, and recompiles the problem. The added logic expressions guide the new solution program and cause it to solve problem instances faster. The learning method may seem roundabout, but has proved to be very effective. The compiler outputs the added logic expressions in a file that has the same name as the input file of the logic formulation, except for the extension .log, which is replaced by .lrn. The format of the output file is such that the file can be inserted with an INCLUDE statement in the FACTS section of the input .log file. When this is done, subsequent compilations of the expanded input .log file do not require the learning step.

If the user comments out the line `output learned clauses` of leibniz-params.dat, then that output is omitted.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Chapter 6

qallsatcc for Solving Q-ALL SAT

Introduction

Program qallsatcc solves the problem Q-All Sat, which is as follows. Given are two formulas R and S in conjunctive normal form (CNF). There are three types of variables: q-variables, x-variables, and y-variables. The formula R uses the q-variables and x-variables, while the formula S uses the q-variables and y-variables.

Q-ALL SAT asks that the following question be decided: Are there values for the q-variables such that formula R can be satisfied with suitably selected values for the x-variables, and formula S cannot be satisfied no matter how the values for the y-variables are chosen?

If the answer to the question is “yes,” then program qallsatcc outputs on the screen “Have a solution.” If the answer is “no,” then the output is “Have NO solution.”

Version 16.0   1 July 2015

Execution

Program qallsatcc is called with the command

```
Leibniz/Qallsatcc/Code/qallsatcc <input file>
```

The input file is described next.

Input File

The input file provides the variables and formulas R and S in the format required by compiler lbcc, with three additional rules.

- The q-variables must start with the letter q, the x-variables with the letter x, and the y-variables with the letter y.
- The formulas of R and S must be in conjunctive normal form (CNF), that is, each clause combines possibly negated variables by “or.”
- Each clause of R and S must have a user-assigned name that is stated after the period ’.’ of the clause and after the mandatory keyword “NAME” or “name.” Moreover, the clause names of R must start with the letter r, and those of S with the letter s.

As an example, here is the file qprob.log in Leibniz/Qallsatcc/Makedata.

```
SATISFY
small

VARIABLES
q1
q2
q3
x1
y1
y2
y3
```

```
FACTS
*
* CNF R
*
not q1 or not q2 or x1.  NAME r_1
q1 or q2 or q3.  NAME r_2
*
* CNF S
*
not q1 or not q2 or y1.  NAME s_1
not q2 or not q3 or y2.  NAME s_2
not q1 or not q3 or not y3.  NAME s_3
ENDATA
```

Execution of

`Leibniz/Qallsatcc/Code/qallsatcc <input file>`

produces the following screen output.

```
INSTANCE small Time: 0
no.  R-conflict clauses: 0
no.  S-conflict clauses: 1
no.  nodes in Qallsat tree: 1
total no.  nodes in Sat trees: 0
No.  times easy part is used: 0
Have NO solution
```

Following some statistics of the computing process that we ignore here, the line `Have NO solution` concludes that there are no values for the q-variables such that formula R can be satisfied with suitably selected values for the x-variables, and formula S cannot be satisfied no matter how the values for the y-variables are chosen.

Applications

Problem Q-All SAT occurs frequently in logic-based intelligent systems. For a detailed discussion, see Chapters 4 and 10 of the book *Design of Logic-based Intelligent Systems*.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Chapter 7

lsqcc for Learning Logic

Introduction

Program lsqcc extracts logic formulas from the records of two data sets A and B. For example, the records of A may represent patients with a certain disease, while those of B represent patients without that disease. Together, the records of A and B are called training data.

The sets A and B are part of usually much larger sets called the $\mathcal{A}$ population and $\mathcal{B}$ population, respectively. In the cited example case, the records of the $\mathcal{A}$ population represent all patients with the specified disease, while the records of the $\mathcal{B}$ population represent all patients without the disease.

The logic formulas produced by lsqcc for given training data may be used to predict the classification of additional data taken from the union of the two populations $\mathcal{A}$ and $\mathcal{B}$. The additional data are called testing data. That prediction involves so-called votes produced from the formulas for each record of the testing data. For evaluation of the accuracy of the predictions, lsqcc derives from the training data certain probability distributions and error probabilities.

Version 16.0 1 July 2015

Three additional programs called optcc, pyrcc, and tstcc can be used in conjunction with lsqcc. They reside in the same directory as lsqcc, that is, in Leibniz/Lsqcc/Code. Summaries of these programs are included next.

Each program reads the parameter file lsqccparams.dat described later. If a different name is used for that file, then the name must be specified when the program is executed. For example,

```
Leibniz/Lsqcc/Code/lsqcc params.file
```

Program optcc

Program optcc derives so-called optimal records from given records of training sets A and B. The sets of optimal records are called A^* and B^*. In the output, the sets are labeled as `Astar` and `Bstar`, and not `A*` and `B*` since the asterisk `*` flags comments.

Program optcc should be used whenever entries of records can only be obtained at some cost. For example, medical tests may have to be performed to obtain patient data for diagnosis. In that setting, one desires logic formulas for diagnostic decisions where the cost of obtaining patient data for use in the formulas is minimum. Such formulas are called optimal. One derives these formulas as follows.

First, one applies program optcc to the given training sets A and B to obtain optimal sets A^* and B^*.

Second, one derives the desired optimal formulas by applying lsqcc two times: Once to A^* and B, and a second time to A and B^*.

Program pyrcc

Program pyrcc is helpful whenever lsqcc obtains logic formulas that are so large that one cannot gain intuitive insight into the logic relationships underlying the data.

When the formulas are too large, program pyrcc simplifies the formulas by creating additional attributes for the records. In a subsequent step, program lsqcc then derives logic formulas from the enlarged records. The new formulas will be smaller than the original ones. By repeating the application of program pyrcc followed by program lsqcc, one eventual obtains formulas of reasonable size.

One may view the derivation of the additional attributes by program pyrcc as a construction of variables of a decision pyramid. The additional variables are then variables at high levels of the pyramid.

The process employed by program pyrcc has its counterpart in real world applications. For example, in medical diagnosis, summarizing concepts such as pathologies may be view as attributes derived from patient data. These concepts help simplify the search for the correct diagnosis.

Program tstcc

Program tstcc applies formulas derived by program lsqcc to additional records. The program can also be invoked by lsqcc, by specifying the appropriate option in the parameter file lsqccparams.dat.

Alternate Execution of Programs

Programs lsqcc, optcc, pyrcc, and tstcc can be executed directly from user code as follows.

First, link with the user code all object code of Leibniz/Lbcc/Code and Leibniz/Lsqcc/Code. The linking requires the option "-lm". Second, in the user code carry out the steps specified in the comment section of the source code of subroutine lsqcc(), optcc(), pyrcc(), or pyrcc(), whichever is of interest. The source code is in file lsqcc.c, optcc.c, pyrcc.c, or tstcc.c, respectively.

Below, all concepts and programs are discussed in detail. We begin with the training data, which are part of the training file.

Training File

We explain the training file using as an example the file data1.trn of Leibniz/Lsqcc/Makedata.

```
*begin of logic definitions
MINIMIZE
partial_log_file

SETS
define cases
pos
neg

PREDICATES
define x1 on cases truecost = 1
define x2 on cases assignfalse *effectively deletes x2 entries
define x3 on cases truecost = 1
define x4 on cases truecost = 1
define x5 on cases truecost = 1

include data1.prd

VARIABLES
dummy_variable
AFORMULA
BFORMULA

FACTS
dummy_variable.

ENDATA
*end of logic definitions

*begin of training data
```

```
BEGIN A
-x1 -x2  x3 -x4 -x5.
 x1 -x2     -x4 -x5.
-x1 -x2 -x3  x4 -x5.
     x2 -x3 -x4  x5.
             x4  x5.

BEGIN B
    -x2  x3  x4  x5.
-x1  x2  x3  x4  x5.
 x1  x2  x3 -x4  x5.
 x1     -x3  x4  x5.
 x1  x2      x4  x5.

ENDATA
```

If an asterisk `*` occurs on a line, then that asterisk and the remaining entries on the line are considered comments and are ignored.

The line `MINIMIZE` is the first line of a partial logic formulation. Here, the user defines the variables that later in the file are employed to describe the training data. Also, the user can insert prior logic facts. These facts are then imposed on the logic relationships that are to be extracted from the training data. The name assigned to the partial logic formulation, here `partial_log_file` after the line `MINIMIZE`, is arbitrary and can be replaced by any other name.

The lines following `MINIMIZE` contain sets, predicates, variables and facts of the partial logic formulation. In the above training file, there is one set named `cases`, with elements `pos` and `neg`, and there are five predicates `x1`, . . . , `x5`.

At the end of the PREDICATES section, the line `include data1.prd` specifies the inclusion of the file data1.prd. This file is called the pyrpred file. The file is not defined by the user, but is created by lsqcc. The file specifies pyramid predicates, if any, that have been created previously by program pyrcc.

CAUTION: The user should not define the pyrpred file. Indeed, any user-created pyrpred file is changed by program lsqcc.

Before we explain the role of the set and the predicates, we need to understand the encoding of the training data. They take up the second half of the training file, starting with the line `BEGIN A`. In this case, the training data have five attributes x_1, ..., x_5. If the value of the attribute x_j is True in a record of the training data, then this is represented by the term `xj` in the record. On the other hand, if the value of the attribute x_j is False, then this is represented by `-xj`. If the True/False value of attribute x_j is not known, no entry is made in the record regarding that attribute. For example, the first record of set A, which is listed following the line `BEGIN A`, is

```
-x1 -x2  x3 -x4 -x5.
```

The record indicates that x_1, x_2, x_4, and x_5 have the value False, while x_3 has the value True. Thus, for all attributes, the True/False values are known. On the other hand, the last record of A, which precedes the line `BEGIN B`, is

```
x4 x5.
```

The record says that both attributes x_4 and x_5 have the value True, and that, for the remaining attributes, the True/False value is not known. Note the final period of each record. It signals the end of the record. The period is needed since, in general, the entries for a record may run over any number of lines, possibly interspersed with blank lines or comments.

The records of B start after `BEGIN B` and are terminated by the final `ENDATA` of the training file. The interpretation is as for the A records.

How are these training records related to the partial logic formulation? First, each attribute x_j of the training data requires a predicate in the partial logic formulation. Since the training data use `xj` to denote that attribute, the predicate is denoted by `xj` as well. The predicate must be defined on a set that includes the reserved elements `pos` and `neg`. Here, that set is called `cases` and consists of the two required elements. The predicates may be used to express logic conditions that the logic formulas yet to be extracted from the training data, must obey. To explain the relationships, we must make a detour and look at the general format of the logic formulas that eventually will be produced from the training data.

The formulas to be extracted from the training data are in disjunctive normal form (DNF). Thus, they are a disjunction of conjunctions of literals. A DNF example, using the above notation for attributes, is $(x_1 \wedge \neg x_3) \vee (\neg x_2 \wedge x_5)$. The logic relationships influence the form of the conjunctions, as follows. If the predicate instance `xj(pos)` (resp. `xj(neg)`) takes on the value True, then the conjunction contains the term x_j (resp. $\neg x_j$). Suppose we add in the FACTS section of the training file the statement `if x1(pos) then x2(pos)`. This would enforce the following. If in any DNF clause the term x_1 occurs, then the term x_2 must occur as well. Such statements are needed when, for example, the values of some attributes can only be acquired by a test. Then the partial logic formulation contains facts that enforce the test if any of the related attributes are used in a DNF clause.

No restrictions are imposed on the predicates and variables of the partial logic formulation, except that for each attribute used in the training data, there must be a corresponding predicate defined on a set that at least has the elements `pos` and `neg`.

The examples of attributes seen so far correspond to propositional variables in the extracted logic formulas. But attributes can also correspond to predicates in those formulas, with the restriction that just one argument is allowed. For example, the attributes $z(k_1)$, $z(k_2)$, and $z(k_3)$ are allowed. Then the partial logic formulation must contain a predicate with two arguments, and hence is specified by a line of the form

```
DEFINE z ON set1 X set2
```

where the sets `set1` and `set2` must obey the following conditions. The set `set1` must contain at least the elements `k1`, `k2`, and `k3`. The set `set2` must contain at least the elements `pos` and `neg`.

The partial logic formulation assigns certain TRUECOST to the predicates. Such cost should reflect the cost of acquiring the corresponding attribute value. In the example training file, all attributes have a nominal acquisition cost of $1.

One may use ASSIGNFALSE to exclude an attribute from the logic formulas without any changes in the training data records. In the example training file, the attribute x_2 is excluded using that approach.

In the above training file, the partial logic formulation leaves the logic formulas unconstrained. Since the partial logic formulation is later compiled by lbcc and checked for formulation errors, we invent a new variable, here `dummy_variable`, and in the FACTS section we insert the statement

```
dummy_variable.
```

That statement forces the variable to be True, but otherwise has no effect on the logic formulas. If the FACTS section contains statements involving predicates defined from the attributes, then the use of `dummy_variable` can be skipped, and the above, trivial fact about that variable can be omitted.

The VARIABLES section also lists two variables `AFORMULA` and `BFORMULA`. During the computation of formulas that evaluate to *True* on set A (resp. B), the variables are fixed to AFORMULA = *True* and BFORMULA = *False* (resp. AFORMULA = *False* and BFORMULA = *True*). This feature can be used to insert prior domain knowledge into the FACTS section. Section "Including Prior Knowledge" in Chapter 8 has details.

CAUTION: The variable names AFORMULA and BFORMULA may not be used for any purpose other than the insertion of domain knowledge. Any other use may invalidate results.

There is more than one way of not knowing the True/False value of an attribute x_j for a given record. Specifically, we may be able to acquire a True/False value of attribute x_j for this record, or we may be unable to determine that information. Let us characterize the first situation by the statement 'do not know value but could acquire it,' and let us define the second case by 'do not know value and cannot acquire it.' In the first case, we say that the entry is *absent*, and in the second case that it is *unavailable*.

The inability to acquire the value may be due to various causes. For example, a person who is to collect the information may deem the information irrelevant for classification of the record in A or B, and thus may refuse to obtain that information. Another cause may be that it is literally impossible to get the value of attribute x_j for that record.

The two cited cases are extremes of numerous in-between situations involving ignorance about attribute x_j. For example, the person collecting the information might have encountered considerable difficulty when trying to get the information, but in principle could have obtained it. For pragmatic reasons, we do not want to differentiate among all the possible cases, nor do we want to give special consideration for each unknown value and the reason why it is unknown. As an approximation, we therefore allow just the two cited cases, and also demand that all instances of unknown values fall into the same category. That is, either all such values could be acquired, or they cannot be obtained. As we shall see, the two cases are treated differently when logic formulas are extracted from the training data.

CAUTION: All cases of unknown values are considered to be of the same type. That is, either they all are of type absent or are of type unavailable.

The above restriction can be dropped, and thus the most general situation of knowing/not knowing attributes can be encoded, if one goes to a more elaborate description of the training data. That is, one associates with each original attribute a second attribute that describes whether one can obtain the value of the original attribute.

The restriction of attribute values to True and False does not mean that data having other values cannot be handled. In such a case, one translates the given data to a logic equivalent. Here is an example.

Suppose the attribute is the color of some object and can take on the values 'red,' 'blue,' or 'green.' We use two attributes `c1` and `c2`. In the training data, 'red' is encoded by `c1 c2`, 'blue' by `-c1 c2`, and 'green' by `c1 -c2`. If the color is not known, we omit any term involving `c1` and `c2`.

For another example, let the given data contain integers or rational numbers. Then the cutcc program described in the next chapter may be used to transform such data to logic data.

We have seen that program lsqcc requires the values of each attribute to be *True*, *False*, absent, or unavailable. In general, the training data may contain besides such values the following data: elements or subsets of given finite sets, integers, or rational numbers. Chapter 7 of the book *Design of Logic-based Intelligent Systems* plus Chapter 8 of this manual tell how all such data can be converted to the values *True*, *False*, absent, or unavailable.

CAUTION: Any record must not have more than MAX_ATTRIBUTE = 1200 attributes. The number of records of either A or B must not exceed MAX_RECORD = 2500. These limits can be changed in file lsqparms.h.

Testing File

Before we turn to the processing of training data by program lsqcc, we briefly cover the testing file. The records of such data come from the union of two populations $\mathcal{A}$ and $\mathcal{B}$, and are encoded like the training data. However, there is no partial logic formulation, and there are no records `BEGIN A` or `BEGIN B`. Here is an example. The file is available in Leibniz/Lsqcc/Makedata as data1.tst.

```
-x1 -x2  x3 -x4 -x5.
 x1 -x2     -x4 -x5.
-x1 -x2 -x3  x4 -x5.
     x2 -x3 -x4  x5.
             x4  x5.
    -x2  x3  x4  x5.
-x1  x2  x3  x4  x5.
 x1  x2  x3 -x4  x5.
 x1     -x3  x4  x5.
 x1  x2      x4  x5.

ENDATA
```

As in the training data, comments and blank records may be inserted at any point. The reader may note that the testing records are precisely equal to the training records. In applications, this typically is not the case. Here, the testing records were selected to be equal to the training records so that the user may see how the derived logic formulas classify records that were also used for training.

CAUTION: A maximum of MAX_RECORD = 2500 testing records may be specified. This limit can be changed in file lsqparms.h.

lsqccparams.dat

Program lsqcc extracts logic formulas from the training data. Before we discuss such formulas, we need to cover a parameter file that specifies for lsqcc the training file and a number of options and values relevant for the learning process. The file is called lsqccparams.dat.

Below the portion of the parameter file controlling execution of lsqcc, optcc, pyrcc, and tstcc is displayed, for the training file data1.trn. The parameter file is available in Leibniz/Lsqcc/Makedata as lsqcc-params.data1. Explanations follow below.

```
***  Parameter File for Leibniz Learning Logic Program lsqcc
***  and Related Programs cutcc, optcc, pyrcc, subcc, tstcc
***
*** To effectively remove any statement below, place
*** asterisk in column 1.  See manual for details.
*********************************************************
*** show steps on screen
show steps on screen
*********************************************************
*** File name without extension
file name without extension = data1
*********************************************************
*** Training/testing directory
*** Specify complete path including terminating '/' or '\\',
*** or specify './' or '.\\' for current directory
***
training/testing directory = ./
*********************************************************
*** File extensions: (roles of files defined subsequently
*** for each program)
***
cutpoint file extension          (default: cut) = cut
distribution file extension      (default: dis) = dis
optimal record file extension    (default: opt) = opt
```

```
pyrpred file extension          (default: prd) = prd
partial data file extension     (default: ptl) = ptl
pyramid file extension          (default: pyr) = pyr
separation file extension       (default: sep) = sep
logic training file extension   (default: trn) = trn
logic testing file extension    (default: tst) = tst
visualization file extension    (default: vis) = vis
vote file extension             (default: vot) = vot
*********************************************************
*** Interpretation of missing entries in records of
*** training/testing files
*** May be declared to be absent or unavailable
*** 'absent'     : means 'do not know value but could
***             acquire it'
*** 'unavailable': means 'do not know value and cannot
***                acquire it'
missing entries (absent, unavailable) = absent
*********************************************************
*********************************************************
*** Program lsqcc for learning logic formulas
*********************************************************
*********************************************************
***
*** Input files:
*** logic training file .trn (possibly generated by cutcc)
*** logic testing file .tst  (possibly generated by cutcc)
*** partial data file .ptl   (generated by cutcc)
*** pyrpred file .prd        (generated by pyrcc)
*** pyramid file .pyr        (generated by pyrcc)
***
*** Output files:
*** distribution file .dis
*** separation file .sep
*** vote file .vot
***
```

```
*** Options:
***
*** Do training, logic training file
do logic training
***
*** Number and type of separations
*** '4 total'   : 4 logic formulas, each fully separating A and B
*** '40 partial': 40 partial formulas, each separating a subset
***               of A from subset of B. Evaluation is done via
***               voting of formulas
number/type of separations (4 total, 40 partial) = 40 partial
***
*** Delete option for nested A/B records
*** 'not allowed': nestedness of records causes program
*** termination
*** 'least cost' : identical A/B records are deleted using
***               least cost criterion
***               nested records are deleted using criterion
***               of number of missing entries
*** 'all'        : all identical A/B records are deleted
***                nested records are deleted using criterion
***                of number of missing entries
nestedness delete option (not allowed, least cost, all) = least cost
***
*** Fraction of A population/(A population + B population)
*** if no fraction is specified, the numbers of training
*** records of A and B are used to estimate fraction
fraction A population (0.xx) = 0.30
***
*** Costs of type A and type B errors
*** if no cost is specified, cost = 1.0 is used
cost type A error = 3.0
cost type B error = 7.0
***
*** Do testing, logic testing file
```

```
do logic testing
***
*********************************************************
*********************************************************
*** Program optcc for computation of optimal records
*********************************************************
*********************************************************
***
*** Input files:
*** logic training file .trn (possibly generated by cutcc)
*** partial data file .ptl   (generated by cutcc)
*** pyrpred file .prd        (generated by pyrcc)
*** pyramid file .pyr        (generated by pyrcc)
***
*** Output files:
*** optimal record file .opt
***
*** Options: nestedness delete option value of lsqcc is used
***
*********************************************************
*********************************************************
*** Program pyrcc for computation of pyramid formulas
*********************************************************
*********************************************************
***
*** Input files:
*** separation file .sep (generated by lsqcc)
*** pyramid file .pyr    (previously generated by pyrcc)
***
*** Output files:
*** pyramid file .pyr     (= expanded input file)
*** pyrpred file .prd
*** pyramid file .pyr.bak (= backup of input file)
***
*** Options: None
```

```
***
*********************************************************
*********************************************************
*** Program tstcc for testing logic formulas
*********************************************************
*********************************************************
***
*** Input files:
*** logic testing file .tst (possibly generated by cutcc)
*** pyramid file .pyr       (generated by pyrcc)
*** separation file .sep    (generated by lsqcc)
***
*** Output files:
*** vote file .vot
***
*** Options: None
***
*********************************************************
ENDATA
```

If a line starts with an asterisk `*`, the compiler considers the line to be a comment and ignores it.

The line `show steps on screen` directs the compiler to output information about each step of the learning process. If this line is commented out, program lsqcc runs silently unless an error termination is encountered.

All error messages are written on the screen and into a file lsqcc.err.

The input and output filenames consist of one name plus various extensions. The line `filename without extension = data1` indicates that the name is data1. For example, the logic training file has the extension .trn. Thus, the complete filename is data1.trn.

The line `training/testing directory = ./` specifies the directory where the training and testing files reside. All output of lsqcc is placed into that directory. Here, the specified directory is `./` and thus is the current one.

The file extensions of the files used by lsqcc as well as of the related programs cutcc, optcc, pyrcc, and tstcc may be modified by a suitable change of the right-hand-side of the following lines.

```
cutpoint file extension           (default: cut) = cut
distribution file extension        (default: dis) = dis
optimal record file extension      (default: opt) = opt
pyrpred file extension             (default: prd) = prd
partial data file extension        (default: ptl) = ptl
pyramid file extension             (default: pyr) = pyr
separation file extension          (default: sep) = sep
logic training file extension      (default: trn) = trn
logic testing file extension       (default: tst) = tst
visualization file extension       (default: vis) = vis
vote file extension                (default: vot) = vot
```

The new extension names may contain up to thirty-two (32) characters.

The line `missing entries (absent, unavailable) = absent` defines how missing entries in records are to be treated. Specifically, if a missing entry in a record means that the value could be acquired, then `absent` should be specified. If a missing entry means that the value is unavailable or irrelevant, then `unavailable` is called for.

The next section contains specified information applying just to the program lsqcc. The section begins with the input and output files of the program, as follows.

```
*********************************************************
*********************************************************
*** Program lsqcc for learning logic formulas
*********************************************************
*********************************************************
***
*** Input files:
```

```
*** logic training file .trn (possibly generated by cutcc)
*** logic testing file .tst  (possibly generated by cutcc)
*** partial data file .ptl   (generated by cutcc)
*** pyrpred file .prd        (generated by pyrcc)
*** pyramid file .pyr        (generated by pyrcc)
***
*** Output files:
*** distribution file .dis
*** separation file .sep
*** vote file .vot
```

The line `do logic training` directs that learning of logic from the training data is to be done. If this line is commented out, the training step is skipped. In that case, program lsqcc performs the same steps as program tstcc.

The line `number of separations (4 total, 40 partial) = 40 partial` specifies how many separations, and thus, how many logic formulas, are to be determined.

The option `4 total` means that 4 formulas are to be found. Each such formula classifies correctly all records of the training sets A and B, save possibly for records removed due to nestedness. For the latter records, a formula may declare the classification to be undecided, or may classify the record correctly or erroneously as being in A or B. The option `4 total` is appropriate if one wants to obtain logic reasons why a record is in A versus B.

The option `40 partial` means that 40 formulas are to be found. Each formula is guaranteed to classify a certain subset of A and a certain subset of B correctly, but may not perform correctly on the remaining subsets. The formula also may not perform correctly on records that have been removed due to nestedness. The subsets of records for which correct classification is guaranteed vary for the different logic formulas. The reader may be baffled by this curious performance of the logic formulas. It turns out to be useful when the 40 formulas are used to classify testing records and when one desires probabilities that these classifications are correct, or when one wants to control the error rates with which $\mathcal{A}$ or $\mathcal{B}$ population records are misclassified. Details are covered later when the output of program lsqcc is discussed.

In Chapter 7 of the book *Design of Logic-based Intelligent Systems*, a concept called *weak nestedness* of records is introduced. We skip details here and mention only that exclusion of weak nestedness is necessary and sufficient for existence of logic formulas classifying all records of the training sets A and B correctly.

Program lsqcc offers an option where the treatment of weakly nested records can be specified. The relevant line is
`nestedness delete option (not allowed, least cost, all) = least cost`.

If the option `not allowed` is specified, then presence of any weakly nested records causes program termination following a suitable error message.

The option `least cost` results in the elimination of weakly nested records according to the following rules. If identical records occur in sets A and B, then such records are deleted either from A or from B so that the expected cost of errors in the classification of testing records is minimized. Additional weakly nested records of A and B are reduced so that the records with higher number of unknown entries are deleted first.

Under the option `all`, the occurrence of identical records in A and B results in the deletion of all such records. Additional weakly nested records are reduced as under the option `least cost`.

The line `fraction A population (0.xx) = 0.30` gives 0.30 as the ratio of $\mathcal{A}$ population size divided by the sum of the $\mathcal{A}$ population size and the $\mathcal{B}$ population size. Put differently, the given fraction is the probability that a random selected record is in the $\mathcal{A}$ population. If the value is not supplied by leaving the right hand side of the line blank, then program lsqcc estimates the value using the ratio of the number of A records divided by the sum of the number of A records and the number of B records.

CAUTION: The ratio of the number of A records divided by the sum of the number of A records and the number of B records possibly is a poor estimate of the fraction $\mathcal{A}$ population.

The lines `cost type A error = 3.0` and `cost type B error = 7.0` tell that misclassification of an $\mathcal{A}$ population record as a $\mathcal{B}$ population record is 3.0, while the cost of misclassification of a $\mathcal{B}$ population record as an $\mathcal{A}$ population record is 7.0. Rational costs are allowed. For example, a cost of 3.75 is accepted.

The line `do logic testing` directs program lsqcc to compute classifications of testing data using the logic formulas obtained from the training data. The same computational steps are performed by program tstcc.

Training

As directed by the file lsqccparams.dat, program lsqcc reads the training file, compiles the partial logic formulation, checks for nestedness of records, and removes weakly nested records. Next, program lsqcc checks if the conditions of the partial logic file prevent separation of any *A* or *B* record from the records of the second set. If this is so for any record, lsqcc analyzes which facts of the partial logic formulation cause the difficulty, and then stops. If such stopping does not occur, then program lsqcc derives logic formulas from the reduced training data and computes certain probability distributions and error probabilities.

The output of program lsqcc consists of two files called the separation file and the distribution file. The separation file contains the logic formulas, while the distribution file has the probability distributions and error probabilities.

In the default naming case, the two file names are the training file name with training extension .trn replaced by the following extensions. In the separation file case, the extension is .sep and in the distribution file case, the extension is .dis.

Below, we discuss the two files that lsqcc produces for the training file data1.trn shown above. Processing is controlled by the earlier listed file lsqccparams.dat. Before proceeding, the reader may want to run that case, as described in Chapter 2.

We skip a discussion of the various messages created by program lsqcc while processing data1.trn, since that output is self-explanatory. We mention here only that the record on line 33 of data1.trn, which is the last record of the *A* set and thus is `x4 x5.`, must be removed due to nestedness before the logic formulas can be determined.

Separation File

For the example training file data1.trn shown above, program lsqcc produces the following separation file data1.sep.

```
40 formulas with 5 logic variables, missing entries = absent
derived from 5 A records and 5 B records
Variables
    x1
    x2
    x3
    x4
    x5

Formula 1 B 1 min (True means vote for B, 1 min size clause(s))
clause size literals
    1    1      5
             Logic notation
             [ x5 ]

Formula 2 B 1 max (True means vote for B, 1 max size clause(s))
clause size literals
    1    2      3   5
             Logic notation
             [ x3 & x5 ]


 . . .
 . . .
Formula 40 A 2 max (True means vote for A, 2 max size clause(s))
clause size literals
    1    1     -3  -4
    2    2     -4  -5
             Logic notation
             [ -x3 & -x4] |
             [ -x4 & -x5 ]
ENDATA
```

The file first lists the variables of the logic formulas. These variables correspond to the attributes used in the training data.

We explain the formulas using formula 40 as an example. It has two clauses, and the clause size, which is the total number of literals, is 2 for both clauses. The literals are given in abbreviated form as `-3` and `-4` for the first clause and as `-4` and `-5` for the second one. This means that the literals of the first clause are the third and fourth variable, both negated, and that the literals of the second clause are the fourth and fifth variable, both negated. The `Logic notation` following the abbreviated notation lists the entire formula explicitly as `[ -x3 & -x4 ] |[ -x4 & -x5 ]`. In mathematical notation, the formula is $[\neg x_3 \wedge \neg x_4] \vee [\neg x_4 \wedge \neg x_5]$.

Recall that the entry `xj` in a record is equivalent to True for the attribute x_j, while a `-xj` corresponds to False. If the True/False value is not known for the attribute, no entry is made.

The formula is applied depending on the information in the first line for the formula, which is `Formula 40 A 1 max (True means vote for A, 1 max size clause(s))`. This means that, if True/False values for the logic variables as defined by a record cause the formula to evaluate to True (resp. False), then this is a counted as a vote for membership in $\mathcal{A}$ (resp. $\mathcal{B}$). We see next how the value of a formula is determined.

Size of Formula

The first line of the formula specifies by `min size` or `max size` whether each clause has as few or as many literals as possible. In the separation file, each min size formula is followed by a corresponding max size formula. For any pair of such formulas, the min size formula has the same number of clauses as the max size formula, and all literals of a give clause in the min size formula also occur in the related clause of the max size formula.

Value of Formula

A clause of a DNF formula evaluates to True if the True/False values of a record cause all literals of the clause to evaluate to True. The clause evaluates to False if, for each literal, the True/False value of the variable of the literal is known and causes the literal to evaluate to False, or if there is a literal for which the True/False value is unknown and is defined to be unavailable. If the clause does not evaluate to True or False according to the above definition, then the clause evaluates to Undecided. Note that the last case is possible only if missing entries are defined to be absent.

The evaluation of a DNF formula is as expected. That is, the value of the formula is True if at least one clause has the value *True*; the value is False if all clauses have value *False*; and the value is Undecided otherwise.

Vote and Vote Total

Suppose the formulas of a separation file are at hand. Take a record and the True/False values it implies for the logic variables. For each formula of that file, compute the True/False/Undecided value as described above. Convert these values to votes, as follows. If the value of the formula is True and the formula specifies `True means vote for A`, or if the value of the formula is False and the formula specifies `True means vote for B`, declare the vote to be 1. We also say that the formula votes for membership in the $\mathcal{A}$ population. If the value of the formula is False and the formula specifies `True means vote for A`, or if the value of the formula is True and the formula specifies `True means vote for B`, declare the vote to be -1. This is called a vote for membership in the $\mathcal{B}$ population. In the remaining case, where the value of the formula is Undecided, the vote is 0.

We add up the votes values and get the vote total for the record. If the separation file contains 40 formulas, then the vote total ranges from -40 to 40, and the number of Undecided cases ranges from 0 to 40. If the separation file contains 4 formulas, the vote total ranges from −4 to 4, and the number of Undecided cases ranges from 0 to 4. If the formulas are based on an interpretation of missing entries as unavailable, then the Undecided value cannot occur, and each vote is 1 or −1.

There is an alternate evaluation for the case where missing entries are interpreted as absent. Specifically, each Undecided case is viewed as a case of False and thus produces a vote of −1. That interpretation is used below when certain distributions are calculated.

Distribution File

Suppose testing records have been randomly selected from the union of the populations $\mathcal{A}$ and $\mathcal{B}$. Then the vote totals for these testing records are values of a random variable *Vote*. When lsqccparams.dat specifies `40 partial` as the number of separations, program lsqcc computes probability distributions, power functions, conditional error probabilities, and expected error costs concerning the random variable *Vote*. That information is outputted in the distribution file.

CAUTION: For the distribution calculations, any Undecided case is counted here as a False case. Among other things, this fact assures that the vote total is always even.

Below, we list the first part of the distribution file data1.dis that program lsqcc computes for the training file data1.trn.

```
    Conditional Distributions of Votes
-----------------------------------------
Notation
F_A(z) = P(Vote < z | A record)
G_A(z) = P(Vote > z | A record)
F_B(z) = P(Vote < z | B record)
G_B(z) = P(Vote > z | B record)
-----------------------------------------
```

```
  z   F_A(z)   G_A(z)    F_B(z)    G_B(z)
-----------------------------------------
-41   0.000    1.000     0.000     1.000
-39   0.000    1.000     0.000     1.000
  .
  .
  1   0.000    1.000     0.800     0.200
  3   0.025    0.975     0.800     0.200
  .
  .
 39   0.825    0.175     1.000     0.000
 41   1.000    0.000     1.000     0.000
-----------------------------------------
```

As summarized in the header of the above table, $F_{\mathcal{A}}(z)$ is the conditional probability that *Vote* $< z$ given that the record is taken from the $\mathcal{A}$ population. The related power function is $G_{\mathcal{A}}(z)$. The corresponding functions for the $\mathcal{B}$ population are $F_{\mathcal{B}}(z)$ and $G_{\mathcal{B}}(z)$.

Note that all z values of the table are odd. Since the vote total is always even, the probability that *Vote* takes on an odd value is 0. Thus, the probability of *Vote* $< z$ or *Vote* $> z$ is the same as for *Vote* $\leq z$ and *Vote* $\geq z$, respectively. Due to this fact, $F_{\mathcal{A}}(z)$ and $F_{\mathcal{B}}(z)$ are distribution functions.

The second part of the distribution file lists error probabilities and expected error costs. That part of the distribution file data1.dis is as follows.

```
        Conditional Error Probabilities and
                Expected Error Costs
-------------------------------------------------------
Prior P(record is in population A) = 0.300
Prior P(record is in population B) = 0.700
Cost of type A error              =       3.00
Cost of type B error              =       7.00
-------------------------------------------------------
```

```
Notation
P_Aerr(z)  = P(type A error if z = threshold)
            = F_A(z) * Prior_P(record in A)
P_Berr(z)  = P(type B error if z = threshold)
            = G_B(z) * Prior_P(record in B)
E_Acost(z) = expected cost of record due to
                type A error if z = threshold
          = P_Aerr(z) * cost of type A error
E_Bcost(z) = expected cost of record due to
                type B error if z = threshold
          = P_Berr(z) * cost of type B error
E_Rcost(z) = expected cost of record due to
                type A or B error if z = threshold
          = E_Acost + E_Bcost
--------------------------------------------------------
  z P_Aerr(z) P_Berr(z) E_Acost(z) E_Bcost(z) E_Rcost(z)
--------------------------------------------------------
-41  0.000      0.700      0.00       4.90      4.90
  .
  .
  1  0.000      0.140      0.00       0.98      0.98
  3  0.007      0.140      0.02       0.98      1.00
  .
  .
 41  0.300      0.000      0.90       0.00      0.90
--------------------------------------------------------

Minimum expected total cost =      0.25
Optimal threshold           =     13

--------------------------------------------------------
```

The header of the table contains data supplied by lsqccparams.dat and also lists the notation for probabilities and costs. We explain the role of the various probabilities and costs by an example.

Suppose we pick a threshold $z = 3$ and predict a testing record to be in the $\mathcal{A}$ population if the vote total is greater than 3 and to be in the $\mathcal{B}$ population otherwise. For the line $z = 3$ of the table, we see that $P_{\mathcal{A}err}(z)$, which is the probability that a record is an $\mathcal{A}$ population record that is misclassified as a $\mathcal{B}$ population record, is 0.007. Such misclassification is called a type $\mathcal{A}$ error. Similarly, $P_{\mathcal{B}err}(z)$, which is the probability that a record is a $\mathcal{B}$ population record that is misclassified as an $\mathcal{A}$ population record, is 0.140. Such misclassification is a type $\mathcal{B}$ error.

So far we have used the first three columns of the table. The next three columns contain expected costs due to classification errors. For $z = 3$, the expected cost of a type $\mathcal{A}$ error, $E_{\mathcal{A}cost}(z)$, is given in the fourth column as 0.02, while the expected cost of a type $\mathcal{B}$ error, $E_{\mathcal{B}cost}(z)$, given in the fifth column is 0.98. Note that $E_{\mathcal{A}cost}(z)$ and $E_{\mathcal{B}cost}(z)$ are not conditional, but are the cost of errors when a randomly selected record of the union of the populations $\mathcal{A}$ and $\mathcal{B}$ is classified.

The last column of the table provides the expected cost $E_{Rcost}(z)$ that is incurred on average when a randomly selected record of the union of the populations $\mathcal{A}$ and $\mathcal{B}$ is classified. That cost is the sum of $E_{\mathcal{A}cost}(z)$ and $E_{\mathcal{B}cost}(z)$, and for $z = 3$ is given as 1.00.

Any z value with minimum $E_{Rcost}(z)$ is an optimal threshold choice that minimizes average misclassification costs.

At the end of the table, the line `Minimum expected total cost = 0.25` gives the minimum of the expected total cost values for the various thresholds. The line `Optimal threshold = 13` provides the associated optimal threshold.

Testing

If lsqccparams.dat specifies `do logic testing`, then program lsqcc applies the learned formulas to records of the testing file and outputs the vote totals in the vote file.

Vote File

The format of the vote file depends on the interpretation of missing entries during the training process. This is due to the fact that the formulas where missing entries = absent, assign True, False or Undecided to each record of the testing file, while the formulas where missing entries = unavailable assign True or False only.

If missing entries = absent is specified, the vote file consists of two parts. In the first part, the counts of Undecided cases are given separately. For that reason, the vote counts should not be used in connection with probability distributions of the vote total. Here is an example of the first part.

```
Part 1 Counts

The vote counts below list Undecided cases separately.
These vote counts should not be used in connection with
probability distributions of the vote total.

  Record      Vote    Undecided |         Range          |
Count Line                      |  Low Vote   High Vote  |
   1     1      40          0   |    40          40      |
   2     2      38          2   |    36          40      |
   .
   .
  10    10     -38          2   |   -40         -36      |
```

The values Low Vote and High Vote are the Vote value minus respectively plus the Undecided count. Thus, Low Vote and High Vote give the lowest respectively highest value that the vote total may take on if one were to turn the Undecided cases into True/False cases by replacing missing entries of testing records by all possible attribute values.

In the second part of the vote file for the case missing entries = absent, the vote counts incorporate the Undecided cases, by giving them the value False. As result, the vote counts can be used in connection with probability distributions of the vote total. In addition, vote counts are given for the min, max, A min, and B min formulas. Here is the second part.

```
Part 2 Counts

The vote counts below consider each Undecided value
as False and therefore can be used in connection with
probability distributions of the vote total.

  Record         Votes of Formulas
Count Line     All   Min   Max  Amin  Bmin
   1     1      40    20    20    10    10
   2     2      36    20    16    10    10
   .
   .
  10    10     -36   -20   -16   -10   -10
```

The vote file for the case missing entries = unavailable has the same format as the second part of the vote file for strong separations.

Once the votes have been listed as described above, the file provides the number of positive, negative, and zero votes, and the total number of votes. The numbers are based on the second table of votes in the case of missing entries = absent, and on the single table of votes in the case of missing entries = unavailable. Here is the relevant portion of the file data1.vot.

```
Voting of Formulas         All  Min  Max Amin Bmin
No.  of positive votes  =    4    4    4    4    4
No.  of zero votes      =    0    0    0    0    0
No.  of negative votes  =    6    6    6    6    6

Total no.  of votes     =   10   10   10   10   10
```

Next in the vote file is the smallest absolute vote and the count and line number of the associated record. In the case of ties, the record with smallest index is chosen. This information can be used to identify outliers that should be removed from the training set. Here is that part of data1.vot.

```
Record With Smallest Absolute Vote
Count =    8
Line  =    8
Vote  =  -20
```

The final part of the vote file consists of importance and usage information of clauses and literals of the separating formulas. The section begins as follows.

```
CAUTION: Frequencies and significance values below may
be used only if the testing data contain A and B cases
approximately in the same proportion as the training
data that produced the separations

Number of records = 10
Estimated number of A records = 5
Estimated number of B records = 5
```

If the option `4 total` was used during training, then the following information is given for each clause of each of the 4 formulas: (1) the probability that a certain alternate random process (ARP) constructs a record for which the clause evaluates to *True*; (2) the number of records for which the clause evaluates to *True*; (3) the significance of the clause, which is an estimate of the probability that the clause is significant.

```
ARP Probability, True Count, and Significance of Clauses

Formula 1 B 2 min (True means vote for B, 2 min size clause(s))
Clause  ARP Prob  True Count  Significance
   1      0.28         3          0.95
   2      0.42         3          0.79
   .
   .
```

The vote file concludes with frequency and significance values for the literals used in the formulas. The frequency tells relative usage of the literal, while the significance is interpreted analogously to the clause case. If for a given literal we have frequency ≥ 0.4 or significance ≥ 0.6, then typically the associated attribute is important.

```
Frequency of Literals

Literal              Frequency
 x5 +                   0.98
 x4 +                   0.68
   .
   .
 x2 -                   0.00


Significance of Literals

Literal              Significance
 x5 +                   0.81
 x5 -                   0.73
   .
   .
 x2 -                   0.00
```

optcc Program

In some applications, the entries of the records can only be obtained at some cost. For example, certain patient data for medical diagnosis may only be obtained by some tests. In such situations, on desires logic formulas that classify the records while total cost for obtaining the data of the records is minimum. Such formulas are called optimal.

Program optcc produces such formulas. The relevant section of the parameter file lsqccparams.dat lists the input and output files of the program.

```
*********************************************************
*********************************************************
*** Program optcc for computation of optimal records
*********************************************************
*********************************************************
***
*** Input files:
*** logic training file .trn (possibly generated by cutcc)
*** partial data file .ptl   (generated by cutcc)
*** pyrpred file .prd        (generated by pyrcc)
*** pyramid file .pyr        (generated by pyrcc)
***
*** Output files:
*** optimal record file .opt
***
*** Options: nestedness delete option value of lsqcc is used
***
*********************************************************
```

The desired logic formulas are obtained in a two-step process. First, we find optimal record sets, and then we determine optimal formulas. Note that the process described here differs from the approach of Chapter 7 of the book of *Design of Logic-based Intelligent Systems*. Indeed, the computation of optimal records is much simpler here, due to the use of program pyrcc described later in this chapter. That program was not available when Chapter 7 of *Design of Logic-based Intelligent Systems* was written.

The process of obtaining optimal records is best described by an example. We begin with the records of data1.trn, where each record has values for attributes x_1, $x_2 \ldots$, x_5. Suppose one may obtain the values by three tests, as follows.

Test 1 produces values for x_1, x_2, and x_3.
Test 2 produces values for x_1 and x_4.
Test 3 produces values for x_3 and x_5.

The costs of performing tests 1, 2, and 3 are \$7, \$15, and \$4, respectively.

Finding an optimal record derived from a record r of A (resp. B) is equivalent to finding a logic formula with one clause that separates the record r from all records of B (resp. A), under the condition that the costs of getting the values for the literals of the formula are minimum. Indeed, the literals of the separating clause define the entries of the optimal record. For example, if the clause contains the literal x_j (resp. $\neg x_j$), then the optimal record has the value *True* (resp. *False*) for the attribute x_j. All entries of the optimal record that are not assigned a value by this process, are irrelevant and thus must be declared to be unavailable. It turns out that this assignment of unavailable is consistent with the declaration of unknown values when optimal records are computed. Details are included below, following discussion of the example training file.

The following training file data2.trn of Leibniz/Lsqcc/Makedata encodes the information about tests and related costs and also includes the training records of data1.trn.

```
*begin of logic definitions
MINIMIZE
partial_log_file
```

```
SETS
define cases
pos
neg
use

PREDICATES
define x1 on cases
define x2 on cases
define x3 on cases
define x4 on cases
define x5 on cases

include data2.prd

VARIABLES
test1 truecost = 7
test2 truecost = 15
test3 truecost = 4

AFORMULA
BFORMULA

FACTS

* if xj occurs in logic formula, enforce xj(use) = True
if x1(pos) or x1(neg) then x1(use).
if x2(pos) or x2(neg) then x2(use).
if x3(pos) or x3(neg) then x3(use).
if x4(pos) or x4(neg) then x4(use).
if x5(pos) or x5(neg) then x5(use).

* link use of xj in formulas with tests
if x1(use) then test1 or test2.
if x2(use) then test1.
if x3(use) then test1 or test3.
if x4(use) then test2.
if x5(use) then test3.

ENDATA
*end of logic definitions

*begin of training data
BEGIN A
-x1 -x2  x3 -x4 -x5.
 x1 -x2     -x4 -x5.
-x1 -x2 -x3  x4 -x5.
     x2 -x3 -x4  x5.
             x4  x5.

BEGIN B
    -x2  x3  x4  x5.
-x1  x2  x3  x4  x5.
 x1  x2  x3 -x4  x5.
 x1     -x3  x4  x5.
 x1  x2      x4  x5.

ENDATA
```

In the SET section, the set `cases` has three elements, `pos`, `neg`, and `use`. We have seen the first two elements before. They are used for the encoding of logic formulas. The third element is employed shortly in certain clauses.

The PREDICATES section contains the predicates `x1`, `x2`..., `x5` as before, except that this time no costs are listed. The predicates are defined on the set `cases`. For example, for `x1`, we have the variables `x1(pos)`, `x1(neg)`, and `x1(use)`.

CAUTION: No costs for *True* or *False* may be assigned to the predicates representing the attributes.

The line `include data2.prd` is of not importance to the user, since it concerns the pyrpred file created by lsqcc.

The VARIABLES section has three variables `test1`, `test2`, and `test3`. The value *True* for a variable means that the related test is performed. The cost of *True* for a variable is equal to the cost of performing the test.

The FACTS section links the predicates `x1`, `x2`..., `x5` and the tests `test1`, `test2`, `test3`. This is accomplished in two sets of clauses.

The first set of clauses forces `xj(use)` to be *True* if `xj(pos)` or `xj(neg)` is *True*. Since `xj(pos)` or `xj(neg)` is *True* if and only if x_j occurs in the separating logic formula, `xj(use)` is *True* if and only if x_j or $\neg x_j$ occurs in the formula.

The second set of clauses links `xj(use)` with the tests `test1`, `test2`, `test3`. For example, the first clause of the section is `if x1(use) then test1 or test2.` The clause reflects the fact that test 1 and test 2 are precisely the test that can determine the value of the attribute x_1.

The remainder of the file data2.trn contains the training data of the sets A and B.

If the computation of optimal records is to be effective, then values should be supplied for all attributes except where a value is irrelevant or impossible to obtain. Thus, all unknown attribute values should be declared unavailable, and the parameter file lsqccparams.dat should specify `missing entries (absent, unavailable) = unavailable`. Indeed, program optcc issues an error message and stops processing if the parameter file declares unknown values to be absent. We note that the declaration of unknown values as unavailable is consistent with the earlier stated interpretation of unknown values of the yet-to-found optimal records.

CAUTION: For execution of program optcc, the parameter file lsqccparams.dat must specify unknown values to be unavailable.

CAUTION: Deletion of nested records is done according to the nestedness delete option specified in the section of lsqccparams.dat for lsqcc.

Optimal Records

When program optcc is executed using the file lsqccparams.data2 of Leibniz/Lsqcc/Makedata as parameter file lsqccparams.dat, the following optimal record file data2.opt results.

```
Optimal records with 5 variables
Variables
    x1
    x2
    x3
    x4
    x5

* 'line' means line number of original training record
* 'cost' means minimum total cost for classifying record
* 'entries' means number of entries in record

BEGIN A   * optimal set Astar
* begin optimal record   line = 46   cost = 4   entries = 2
x3 -x5.
* end optimal record   line = 46
* begin optimal record   line = 47   cost = 4   entries = 1
-x5.
* end optimal record   line = 47
* begin optimal record   line = 48   cost = 4   entries = 2
-x3 -x5.
* end optimal record   line = 48
* begin optimal record   line = 49   cost = 7   entries = 2
x2 -x3.
* end optimal record   line = 49

BEGIN B   * optimal set Bstar
* begin optimal record   line = 53   cost = 4   entries = 2
```

```
x3 x5.
* end optimal record   line = 53
* begin optimal record   line = 54   cost = 4   entries = 2
x3 x5.
* end optimal record   line = 54
* begin optimal record   line = 55   cost = 4   entries = 2
x3 x5.
* end optimal record   line = 55
* begin optimal record   line = 56   cost = 7   entries = 2
x1 -x3.
* end optimal record   line = 56
* begin optimal record   line = 57   cost = 7   entries = 2
x1 x2.
* end optimal record   line = 57
ENDATA
```

We discuss the format of the optimal records of the file, using the first record of A^* as an example. The records immediately follows the line `BEGIN A   * optimal set Astar`. The record is specified by the three lines

```
* begin optimal record   line = 46   cost = 4   entries = 2
x3 -x5.
* end optimal record   line = 46
```

The first line says that the record of the training set A beginning on line 46 of the training file data2.trn can be separated from all records of the set B of data2.trn, by performing certain tests with minimum total cost equal to 4.

The tests produce two entries, which are given on the second line. Specifically, the values $x_3 = True$ and $x_5 = False$ are obtained by the tests and constitute the entries of the optimal record. All other entries are declared unavailable.

The third line signals the end of the optimal record.

Optimal Formulas

When program optcc has determined the optimal record sets A^* and B^*, one can compute optimal formulas by applying program lsqcc twice: once to the sets A^* and B, and a second time to the sets A and B^*. For the first case, the training file is constructed as follows.

– Replace the records of A in data2.trn by those of A^* of data2.opt.

– Assign to the predicates `x1`, …, `x5` a cost of *True* equal to 1, by adding `truecost = 1`.

– Eliminate the test variables `test1`, `test2`, `test3`.

– Introduce the variable `dummy_variable`.

– Delete all clauses of the FACTS section and instead introduce the clause `dummy_variable.`

Here is the resulting file data2.AstarB.trn, which is included in Leibniz/Lsqcc/Makedata.

```
*begin of logic definitions
MINIMIZE
partial_log_file

SETS
define cases
pos
neg
use

PREDICATES
define x1 on cases truecost = 1
define x2 on cases truecost = 1
define x3 on cases truecost = 1
define x4 on cases truecost = 1
define x5 on cases truecost = 1
```

```
include data2.AstarB.prd

VARIABLES
dummy_variable
AFORMULA
BFORMULA

FACTS
dummy_variable.

ENDATA
*end of logic definitions

*begin of training data
BEGIN A   * optimal set Astar
* begin optimal record   line = 46   cost = 4   entries = 2
x3 -x5.
* end optimal record   line = 46
* begin optimal record   line = 47   cost = 4   entries = 1
-x5.
* end optimal record   line = 47
* begin optimal record   line = 48   cost = 4   entries = 2
-x3 -x5.
* end optimal record   line = 48
* begin optimal record   line = 49   cost = 7   entries = 2
x2 -x3.
* end optimal record   line = 49

BEGIN B
    -x2  x3  x4  x5.
-x1  x2  x3  x4  x5.
 x1  x2  x3 -x4  x5.
 x1     -x3  x4  x5.
 x1  x2      x4  x5.
```

```
ENDATA
```

For execution of program lsqcc, the parameter file lsqccparams.dat must specify all unknown values to be unavailable, by the line `missing entries (absent, unavailable) = unavailable`.

CAUTION: The condition that unknown values must be specified to be unavailable, is not checked by program lsqcc. If the condition is not observed, program lsqcc may delete records to achieve separation of sets and thus may produce erroneous optimal formulas.

When program lsqcc has computed the logic formulas for the training set data2.AstarB.trn, we discard half of these formulas. Indeed, we discard all formulas which take on the value *True* on the set B. The remaining formulas have the value *True* on the set A^*.

The case of optimal formulas for the case of A and B^* is handled analogously. Here, the records of the set B in data2.trn are replaced by the records of B^* of data2.opt. This time, the logic formulas that evaluate *True* on the A records are discarded, and the formulas that evaluate to *True* on the records of B^* are kept. Here, too, the parameter file lsqccparams.dat must specify all unknown entries to be unavailable.

The use of optimal formulas depends on the application. For details, see Chapters 10 and 11 of the book *Design of Logic-based Intelligent Systems*.

pyrcc Program

Program pyrcc allows simplification of logic formulas that due to the complexity of the training data are too large. For example, this may happen when optimal formulas are computed using optimal record sets.

The relevant input and output files of program pyrcc are specified in the following section of the parameter file lsqccparams.dat.

```
*********************************************************
*********************************************************
*** Program pyrcc for computation of pyramid formulas
*********************************************************
*********************************************************
***
*** Input files:
*** separation file .sep (generated by lsqcc)
*** pyramid file .pyr    (previously generated by pyrcc)
***
*** Output files:
*** pyramid file .pyr     (= expanded input file)
*** pyrpred file .prd
*** pyramid file .pyr.bak (= backup of input file)
***
*** Options: None
***
*********************************************************
```

The use of program pyrcc is best explained via a small example. Suppose program lsqcc has been applied to the training set data1.trn as described earlier, except that this time the option
`number of separations (4 total, 40 partial) = 4 total`
is used in the parameter file lsqccparams.dat. That option must be specified when program pyrcc is to be invoked subsequently.

The logic formulas produced by program lsqcc are in the file data1.sep. Here is the file.

```
4 formulas with 5 logic variables, missing values = absent
derived from 5 A records and 5 B records
Variables
    x1
    x2
    x3
    x4
    x5

Formula 1 B 2 min (True means vote for B, 2 min size clause(s))
clause size literals
    1    2       3    5
    2    2       4    5
             Logic notation
             [ x3 & x5 ] |
             [ x4 & x5 ]

Formula 2 B 2 max (True means vote for B, 2 max size clause(s))
clause size literals
    1    2       3    5
    2    3       1    4    5
             Logic notation
             [ x3 & x5 ] |
             [ x1 & x4 & x5 ]

Formula 3 A 2 min (True means vote for A, 2 min size clause(s))
clause size literals
    1    1      -5
    2    2      -3   -4
             Logic notation
             [ -x5 ] |
             [ -x3 & -x4 ]

Formula 4 A 2 max (True means vote for A, 2 max size clause(s))
clause size literals
```

```
    1    1      -5
    2    2      -3   -4
             Logic notation
             [ -x5 ] |
             [ -x3 & -x4 ]
ENDATA
```

Using the file lsqccparams.dat specified at the beginning of this section, execute program pyrcc with

```
pyrcc lsqccparams.dat
```

to simplify the logic formulas of data1.sep. Of course, these formulas are already simple, so this is just for demonstration of program pyrcc.

Pyramid File

We ignore the screen output of pyrcc and look at the output pyramid file data1.pyr.

```
1 pyramid formula(s) with 6 variables including pyramid
derived from 5 A records and 5 B records
Variables
    x1
    x2
    x3
    x4
    x5
    pyr_var_1

Formula 1 pyr_var_1 2 (variable pyr_var_1, 2 clauses)
clause size literals
    1    1    3
    2    1    4
             Logic notation
             [ x3 ] |
             [ x4 ]
2 Clauses: 1 2
1 Variables: 5
              x5
ENDATA
```

Following the listing of the original variables of data1.trn, there is a new variable `pyr_var_1`, which is defined by the subsequent formula 1. The interpretation of the formula is exactly as for logic formulas of separation files, except that here the value of the formula is considered the value of the variable `pyr_var_1`. After the formula, the line `2 Clauses: 1 2` says that clauses 1 and 2 of the logic formula produced the pyramid formula. The next two lines tell which variables are deleted from the clauses to give the pyramid formula. In the example, the lines `1 Variables: 5` and `x5` say that, when the fifth variable of the variable list, that is, `x5`, is deleted from clauses 1 and 2, then the reduced clauses give the pyramid formula.

In the general case, program pyrcc may produce up to two pyramid formulas.

We execute program lsqcc again. This time, the program uses the formula for `pyr_var_1` of the pyramid file data1.pyr to define one additional attribute of the training data. That attribute is used to expand the records of the training sets A and B. For example, the first record of the training set A, which is

```
-x1 -x2 x3 -x4 -x5.
```

says that `x1` = *False*, `x2` = *False*, `x3` = *False*, `x4` = *True*, and `x5` = *False*. When these values are inserted into the formula of data1.pyr for the pyramid variable `pyr_var_1`, which is

```
[ x3 ] | [ x4 ]
```

then `pyr_var_1` takes on the value *True*. The latter value is considered to be part of the training record.

The logic formulas obtained by program lsqcc are given in the following, new separation file data1.sep.

```
4 formulas with 6 logic variables, missing values = absent
derived from 5 A records and 5 B records
Variables
    x1
    x2
    x3
    x4
    x5
    pyr_var_1

Formula 1 B 1 min (True means vote for B, 1 min size clause(s))
clause size literals
    1    2       5    6
             Logic notation
             [ x5 & pyr_var_1 ]

Formula 2 B 1 max (True means vote for B, 1 max size clause(s))
clause size literals
    1    2       5    6
             Logic notation
             [ x5 & pyr_var_1 ]

Formula 3 A 2 min (True means vote for A, 2 min size clause(s))
clause size literals
    1    1      -5
    2    1      -6
             Logic notation
             [ -x5 ] |
             [ -pyr_var_1 ]

Formula 4 A 2 max (True means vote for A, 2 max size clause(s))
clause size literals
    1    1      -5
    2    3       -3   -4 -6
             Logic notation
             [ -x5 ] |
             [ -x3 & -x4 & -pyr_var_1 ]
ENDATA
```

Note the use of the variable `pyr_var_1` in the above formulas.

One may repeat the above steps any number of times. That is, given a separation file produced by program lsqcc with the option
`number of separations (4 total, 40 partial) = 4 total`
one executes program pyrcc to get up to two additional pyramid formulas. Then one executes program lsqcc again to get a new separation file that utilizes the pyramid formulas at hand. Next, one executes program pyrcc, and so on.

During an alternating sequence of execution of programs lsqcc and pyrcc, a number of pyramid files are produced. At some point, the user may want to go back to the pyramid file that immediately preceded the current one. For this situation, program pyrcc makes a backup file, with extension .bak, of any existing pyramid file, before creating a new pyramid file. In the example case, the backup file is data2.pyr.bak. At the beginning of the execution sequence, when no pyramid file is present, the creation of the backup file is skipped.

The process stops when program pyrcc does not produce any additional pyramid formula. For the above example, that situation occurs when one applies program pyrcc to the most recently computed separation file listed above.

Once the construction process of pyramid formulas has stopped, one may execute program lsqcc with the option ‘4 total’ or ‘40 partial’ to get 4 or 40 logic formulas. In both cases, the pyramid formulas are utilized.

CAUTION: The option ‘4 total’ must be used during construction of pyramid formulas. On the other hand, pyramid formulas may be used subsequently for derivation of logic formulas under either the ‘4 total’ or ‘40 partial’ option.

One may also execute program tstcc described later to evaluate testing data sets when logic formulas involve pyramid variables. Here, too, the pyramid formulas are used together with the logic formulas to evaluate the testing records.

Consistency Check

The above recursive use of programs lsqcc and pyrcc is easily upset when these two programs are not used in the above described alternating manner, or when the training file is changed in midstream of such a sequence. Both programs lsqcc and pyrcc attempt to guard against such inappropriate use by a consistency check that compares the variables listed in the current separation file with those of the current pyramid file.

The rules of the consistency check are as follows.

(1) When program lsqcc is executed, all attributes of the training file must also occur in the pyramid file, and any additional attribute of the pyramid file must be a pyramid variable previously generated by program pyrcc.

(2) When program pyrcc is executed, each attribute of the separation file must also occur in the pyramid file, and any additional attribute of the pyramid file must be a pyramid variable previously generated by program pyrcc.

When an inconsistency is detected, program lsqcc or pyrcc puts out the following error message that includes suggestions for error analysis, and then stops.

```
The variables of the existing pyramid file
./xxx.pyr
and of the existing separation file
./xxx.sep
are not identical.  Such consistency is required.
```

```
Possible reasons for inconsistency:
- Logic variables in training file have been changed.
  and training file is no longer compatible with pyramid
  file.  Remedy: delete pyramid file.
- Pyramid file has been updated by pyrcc, but lsqcc
  has not yet been run again.  Remedy: Run lsqcc.
- Pyramid file has been run once, and currently
  lsqcc is applied.  Inconsistency is caused by missing
  statement 'include xxx.prd' in training file.
  Remedy: add 'include xxx.prd' to training file.
See manual for details.
Stop
```

The last error, where the statement 'include xxx.prd' is missing, is the most likely cause of the inconsistency. In the typical situation, the user has successfully run programs lsqcc and pyrcc once, in that order, and now wants to run program lsqcc again. At that time, program lsqcc complains about inconsistency, and it is due to `include xxx.prd` missing in the PREDICATE section of the training file xxx.trn. Addition of that line eliminates the inconsistency.

CAUTION: In directory Leibniz/Lsqcc/Makedata, the example training file data1.trn already includes the statement `include data1.prd`. When program lsqcc is initially executed, that statement triggers search for the file data1.prd. If that file is absent, the statement is simply ignored, and no harm is done by the statement. On the other hand, when iterative sequences executing lsqcc and pyrcc are done, the statement is there, as required. Thus, when execution of prpcc is contemplated, it is advisable to always include the statement `include xxx.prd` in the PREDICATE section of the initial training file xxx.trn.

If at any point one wants to eliminate the pyramid formulas, one simply deletes the entire pyramid file and reruns program lsqcc to obtain logic formulas that do not involve any pyramid variables.

To what extent should pyramid formulas be used? A pragmatic answer is as follows. After each execution of programs pyrcc and lsqcc, one applies the new logic formulas to testing data and checks the level of accuracy of classification. One then selects the pyramid formulas producing the highest accuracy among the cases produced by the repeated use of programs pyrcc and lsqcc.

tstcc Program

The earlier described vote files can also be produced by program tstcc. The following section of the parameter file lsqccparams.dat lists input and output files of the program.

```
*********************************************************
*********************************************************
*** Program tstcc for testing logic formulas
*********************************************************
*********************************************************
***
*** Input files:
*** logic testing file .tst (possibly generated by cutcc)
*** pyramid file .pyr       (generated by pyrcc)
*** separation file .sep    (generated by lsqcc)
***
*** Output files:
*** vote file .vot
***
*** Options: None
***
*********************************************************
```

Program tstcc applies the logic formulas of the separation file to the records of the testing file to compute the vote totals. The source C code of tstcc is a helpful template when the user carries out the computation of vote totals directly in user C programs.

Leibniz System

Development Tool
for Logic-based
Intelligent Systems

Chapter 8

cutcc for Data Transformation

Introduction

Program cutcc takes records with rational data and/or set data and converts them to records with logic data. This is a necessary preprocessing step when logic formulas are to be extracted from rational data and/or set data. Once the transformation to logic data has been accomplished, program lsqcc described in Chapter 7 can be applied to obtain the logic formulas. It is assumed that the reader has studied that chapter. Thus, we use the terminology of Chapter 7 without additional definitions or explanations.

The input for program cutcc essentially consists of two sets A and B of records, each of which can have as entries rational numbers, elements of sets, or the value absent or unavailable. The latter two values represent cases where the entries are not known. For details of the interpretation, see Chapter 7. Consistent with the assumption about unknown values made for program lsqcc, it is assumed here that just one type of unknown entry is used. Thus, the unknown entries are either all absent or all unavailable.

Program cutcc analyzes the records of the sets A and B and produces rules by which the records of the sets can be converted to records just having logic entries and/or the value absent or unavailable. The two sets so derived can then be processed by program lsqcc to get the desired logic relationships and formulas. Whenever additional records are to be processed by the formulas, one first uses the rules to convert the records to ones having just logic data and/or the value absent or unavailable, and then the logic formulas are applied to the resulting records.

The conversion of rational data and set data to logic data is carried out by program cutcc according to a method called Cutpoint. It has been found to be effective and robust in a variety of applications.

Program cutcc reads the parameter file lsqccparams.dat described later. If a different name is used for that file, then the name must be specified when the program is executed. For example,

```
Leibniz/Cutcc/Code/cutcc params.file.
```

Below, the files and steps of program cutcc are discussed in detail. We begin with the rational training file, which contains the records of the sets A and B.

Alternate Execution of cutcc

Program cutcc can be executed directly from user code as follows.

First, link all object code of Leibniz/Cutcc/Code with the user code. The linking requires the option "-lm". Second, carry out in the user code the steps specified in the comment section of the source code of subroutine cutcc(). The subroutine is included in file cutcc.c.

Rational Training File

We explain the rational training file using the example file heart.rtr of Leibniz/Cutcc/Makedata. Here is that file.

```
ATTRIBUTES
*
age
sex SET .5
cp SET 1.5 2.5 3.5
trestbps
chol MAXCUTS 2 UNCERTAIN
fbs SET .5
restecg SET .5 1.5
thalach
exang SET .5
oldpeak UNCERTAIN
slope SET 1.5 2.5
ca MAXCUTS 3
thal SET 4 6.5
num DELETE
*
BEGIN A
63 1 1 145 233 1 2 150 0 2.3 3 0 6 0
37 1 3 130 250 0 0 187 0 3.5 3 0 3 0
.
.
60 0 3 102 318 0 0 160 0   0 1 1 3 0
BEGIN B
67 1 4 120 229 0 2 129 1 2.6 2 2 7 1
53 1 4 140 203 1 2 155 1 3.1 3 0 7 1
.
.
57 1 4 165 289 1 2 124 0   1 2 3 7 4
ENDATA
```

If an asterisk `*` occurs on a line, then that asterisk and the remaining entries on the line are considered comments and are ignored.

The line `ATTRIBUTES` is the first line of the rational training file. Here, the user defines the attributes of the subsequent records. The names are used later to define variables names for the logic data obtained from the rational training data.

Each attribute is listed on one line. For example, the line `age` specifies that the attribute is age. The numerical value assigned to the attribute in the subsequent records is assumed to be an integer or rational number. For a second example, the line `sex SET .5` says that the attribute sex is given by elements of a set that is assumed to be a finite set of rational numbers. The elements of the set need not be given. Instead, the keyword `SET` following `sex` must be followed by rational values that strictly separate the elements of the set. Here, the line `sex SET .5` specifies that there are two elements in the set, and that the value 0.5 separates these elements. Generally, if the set has $k+1$ elements, then k separating values should follow the keyword SET. This rule may be violated, as follows. If k separating values are given, then the values of the attribute given later in the records are sorted into $k+1$ buckets defined by the values of the separating values. Thus, it is allowed that two or even more elements of the set may fall into the same interval. This feature effectively allows an aggregation of set elements.

CAUTION: None of the specified rational separating values may be a member of the set. If this condition is violated, then any member of the set equal to a separating value is treated as unknown value. See below for details about unknown values.

The second half of the input file starts with the line `BEGIN A`. The subsequent lines contain the records, with one line per record. The values must be given in the order specified under the `ATTRIBUTE` header. For example, the first record following `BEGIN A` is
`63 1 1 145 233 1 2 150 0 2.3 3 0 6 0`
and thus specifies age = 63, sex = 1, cp = 1, trestbps = 145, and so on.

The sequence of records of A is terminated by the line `BEGIN B`, which signal the beginning of the records of B. The format of the B records is the same as that for the A records.

When the value of some attribute in a record is not known, this is indicated by a question mark (`?`). The unknown value may be of type unavailable or absent, where absent means that the unknown value could be obtained, while unavailable means that the value cannot be obtained.

CAUTION: All cases of unknown values are considered to be of the same type. That is, either they all are of type absent or are of type unavailable.

The above restriction can be dropped, and thus the most general situation of knowing/not knowing attributes can be encoded, if one goes to a more elaborate description of the input data. That is, one associates with each original attribute a second attribute that describes whether one can obtain the value of the original attribute.

There are three options for attributes.

First, if one wants to delete the data of an attribute from a record, one simply inserts the keyword `DELETE` after the attribute name on the same line. For example, the last line of the example listing is `num DELETE` and thus indicates that the data for the attribute `num` should be ignored.

CAUTION: The keyword DELETE must follow after an attribute name on the same line.

Second, if one wants to limit the number of logic values for an attribute when SET is not used, then one includes `MAXCUTS` $\langle m \rangle$ after the attribute, where m is the limit. Similarly, one may enforce a minimum number of logic values with `MINCUTS` $\langle m \rangle$. For example, `chol MAXCUTS 3` defines the limit to be equal to 3, while `chol MINCUTS 2` enforces that at least two logic variables are used. Both options may be specified simultaneously, in any order.

CAUTION: The MAXCUTS and MINCUTS options may not be used in conjunction with SET.

Third, if one wants to rule out conversion of rational values near critical points called cutpoints, one specifies the option `UNCERTAIN`. Cutpoints are discussed in detail in the next section. The option results in an interval of uncertainty around each cutpoint and produces an encoding with the logic value of unavailable when a rational value falls into the interval. The option may be used only if file lsqccparams.dat declares unknown values to be unavailable.

CAUTION: UNCERTAIN may not be used in conjunction with SET, and it requires that lsqccparams.dat declares unknown values to be unavailable.

The second and third option may be used simultaneously. In that case, the order in which the options are listed does not matter. For example, `chol MAXCUTS 3 UNCERTAIN`, `chol MINCUTS 2 UNCERTAIN`, and `chol UNCERTAIN MINCUTS 2 MAXCUTS 3` are allowed.

CAUTION: There can be at most MAX_ATTRIBUTE = 600 attributes. The number of records of both A and B must not exceed MAX_RECORD = 4000. These limits may be changed in file cutcc.h.

Rational Data Versus Set Data

The reader may wonder about the distinction between rational data and set data, since the set elements must be rational numbers. The difference lies in the way these data are encoded.

For rational data, program cutcc selects cutpoints that divide the rational line into intervals. For each cutpoint, a logic variable is selected. All rational values to the left (resp. right) of a cutpoint produce the logic value True (resp. False) for the logic variable associated with the cutpoint. This definition allows a compact encoding of any condition that a rational value must be below or above a given cutpoint.

For set data, program cutcc takes the specified cutpoints and defines one logic variable for each interval implied by the cutpoints. This definition allows a compact encoding of any condition that a rational value must fall within a given interval or be outside that interval.

The reader may surmise that the above distinction is too subtle to be of importance. However, tests have shown the following. Suppose that rational training files are converted to logic training files, and that program lsqcc is used to deduce logic formulas from the logic training files. Then the accuracy of the formulas, when applied to additional data, is significantly influenced by the fact which attributes values have been declared to be rational numbers versus elements of sets. A good rule of thumb is that, if just a few distinct values are possible, then one should declare the attribute values to be elements of a set. Otherwise, the values should be declared to be rational numbers.

Rational Testing File

The records of the rational testing file come from the union of two populations $\mathcal{A}$ and $\mathcal{B}$. The testing data are encoded in the same way as the training data. However, there are no records `BEGIN A` or `BEGIN B`. Here is the example file heart.rts of Leibniz/Cutcc/Makedata. The first record, `*BEGIN A`, is a comment and thus disregarded by program cutcc. We have inserted the comment to tell that the subsequent records come from population $\mathcal{A}$. Later in the file, `*BEGIN B` tells that records from population $\mathcal{B}$ follow.

```
*BEGIN A
52 1 1 152 298 1 0 178 0 1.2 2 0 7 0
42 0 4 102 265 0 2 122 0 0.6 2 0 3 0
.
.
*BEGIN B
.
.
58 1 4 114 318 0 1 140 0 4.4 3 3 6 4
ENDATA
```

As in the training data, comments and blank records may be inserted at any point.

<u>CAUTION</u>: A maximum of MAX_RECORD = 4000 rational testing records may be specified. This limit can be changed in file cutcc.h.

lsqccparams.dat

Program cutcc converts the rational training and testing files to logic training and testing files. The program uses the parameter file lsqcc-params.dat introduced in Chapter 7. Below, the portion of the example file lsqccparams.heart in Leibniz/Cutcc/Makedata relevant for cutcc is listed.

```
***  Parameter File for Leibniz Learning Logic Program lsqcc
***  and Related Programs cutcc, optcc, pyrcc, tstcc
***
*** To effectively remove any statement below, place
*** asterisk in column 1.  See manual for details.
*********************************************************
*** show steps on screen
show steps on screen
*********************************************************
*** File name without extension
file name without extension = heart
*********************************************************
*** Training/testing directory
*** Specify complete path including terminating '/' or '\\',
*** or specify './' or '.\\' for current directory
***
training/testing directory = ./
*********************************************************
*** File extensions:  (roles of files defined subsequently
*** for each program)
***
cutpoint file extension           (default:  cut) = cut
distribution file extension       (default:  dis) = dis
optimal record file extension     (default:  opt) = opt
pyrpred file extension            (default:  prd) = prd
partial data file extension       (default:  ptl) = ptl
rational training file extension  (default:  rtr) = rtr
```

```
pyramid file extension            (default:  pyr) = pyr
rational training file extension  (default:  rtr) = rtr
rational testing file extension   (default:  rts) = rts
separation file extension         (default:  sep) = sep
logic training file extension     (default:  trn) = trn
logic testing file extension      (default:  tst) = tst
visualization file extension      (default:  vis) = vis
vote file extension               (default:  vot) = vot
*********************************************************
*** Interpretation of missing entries in records of
*** training/testing files
*** May be declared to be absent or unavailable
*** 'absent'     :  means 'do not know value but could
***             acquire it'
*** 'unavailable':  means 'do not know value and cannot
***                acquire it'
missing entries (absent, unavailable) = unavailable
*********************************************************
*********************************************************
*** Program cutcc for transformation of rational/set data
*********************************************************
*********************************************************
***
*** Input files:
*** rational training file .rtr
*** rational testing file .rts
*** if training option = old:
***   cutpoint file .cut
***
*** Output files:
*** partial data file .ptl
*** logic training file .trn
*** logic testing file .tst
*** visualization file .vis
*** if training option = new:
```

```
***   cutpoint file .cut
***
*** Options:
***
*** Do transformation of rational training file
*** using new or old cutpoint file .cut, or skip step
do training transformation (new, old, skip) = new
***
*** Min number of cuts
*** must be nonnegative; if no value is specified,
*** min cuts = 1 is used
min cuts = 1
*** Max number of cuts
*** must be positive; if no value is specified,
*** max cuts = 1 is used
max cuts = 6
***
*** Degree of separation
*** 'effective':  separation of sets A and B is effective
*** but not guaranteed to be perfect
*** 'perfect'  :  perfect separation is guaranteed unless
*** max cuts limit stops process early
degree of separation (effective, perfect) = effective
***
*** Do transformation of rational testing file
do testing transformation
*********************************************************
```

Explanations for the first part, up to the line `missing entries (absent, unavailable) = absent`, are given in Chapter 7. The interpretation of the remaining portion is as follows.

The line `do training transformation (new, old, skip) = new` directs that the rational training file is to be transformed using new cutpoints calculated from the training file data. If the option `old` is specified instead, then the cutpoints of an existing .cut file are used. If `skip` is specified, then the training step is skipped.

CAUTION: If the option `old` is used, then the list of attributes in the .rtr file must agree with that of the .cut file. This condition is NOT checked by cutcc. Also, if an attribute in one of the files is marked DELETE, then that attribute must be marked DELETE in the second file. This condition is not checked by cutcc. However, cutcc does check whether the total number of attributes is the same for both files, and if the number of deleted attributes is the same for both files.

CAUTION: If the option `old` is used, then the .cut file must follow the format described in the subsequent section Cutpoint File.

The line `min cuts = 1` specifies the minimum number of cutpoints that must be introduced by cutcc for any attribute with rational data. Typically, the value 1 provides good results. The minimum may be changed in the training file for any attribute by the option `MINCUTS` $\langle m \rangle$, where m is the limit. The specification `min cuts = 0` is allowed, in which case just enough cutpoints are determined so that the two training sets A and B can be separated. The selection of the needed cutpoints is based on certain significance values of the possible cutpoints.

The line `max cuts = 6` specifies the maximum number of cutpoints that may be introduced by cutcc for any given attribute. The maximum may be changed in the training file for any attribute by the option `MAXCUTS` $\langle m \rangle$, where m is the limit.

CAUTION: If the value for max cuts is selected too small, program cutcc may not be able to achieve a transformation of the rational training data to logic data for which program lsqcc can compute separating logic formulas. In that case, a logic training file is still produced, but program lsqcc will have to remove some records to achieve a separation.

CAUTION: If the specified maximum number of cutpoints is below the specified minimum, then the maximum is redefined to be the equal to the minimum.

The line `degree of separation (effective, perfect) = effective` defines whether the resulting logic records must be fully separable by program lsqcc. If the option `effective` is used, then full separation may not be attained, while the option `perfect` forces full separation, provided this is at all attainable under the specified value `max cuts` value. It may seem that one should always choose the option `perfect`. But this is a poor choice if, by the very nature of the training data, a training record of A is essentially equal to a training record of B. In such a case, program cutcc tends to introduce numerous and inappropriate cutpoints in a possibly futile attempt to achieve full separation. Thus, one should use the option `perfect` if the case of essentially equal records of A and B is not expected, and one should select the option `effective` if the case of such records is quite possible or even likely.

The line `do testing transformation` directs program cutcc to transform the records of the rational testing file to logic data. If this step is performed, the resulting records with logic data are placed into the logic testing file.

Output Files

As directed by the parameter file lsqccparams.dat, program cutcc reads the rational training file, establishes transformations for the attributes, and outputs the resulting logic training file. The program also produces several other files: a cutpoint file if the training option `new` is specified, a visualization file, and a partial data file. We explain these files using examples files produced from the rational training file heart.rtr using the file lsqccparams.dat in

```
cutcc lsqccparams.heart
```

We begin with the cutpoint file.

Cutpoint File

The transformation from rational data to logic data essentially consists of the following. For each attribute, the rational line is divided into intervals by cutpoints, and then each rational value, say falling into the kth bucket, is encoded by logic data representing that fact. If `UNCERTAIN` is specified for an attribute, then each cutpoint of the attribute is followed by an interval of uncertainty $[z_1, z_2]$. A rational value falling into the interval is encoded by unavailable for the logic variable associated with the cutpoint.

There are various methods for selecting cutpoints. Program cutcc uses a method that leads to good classification accuracy of the logic formulas computed by program lsqcc.

Below, the cutpoints displayed in the cutpoint point file heart.cut of Leibniz/Cutcc/Makedata produced from the rational training file heart.rtr is shown.

```
CUTPOINTS
*attribute type count cutpoints
age NUM 1 50.500000
```

```
sex SET 1 0.500000
.
.
chol UNCERTAIN 2 232.500000 [ 231.000000 , 233.000000 ]
        292.000000 [ 288.000000 , 302.000000 ]
.
.
thal SET 2 4.000000 5.000000
num DELETE
ENDATA
```

Following the headers `CUTPOINTS` and `*attribute type count cutpoints`, each line covers the cutpoints of one attribute. For example, the line `age NUM 1 50.500000` says the following. The attribute age is a rational number, the number of cutpoints is 1, and this cutpoint is 50.5. The line `sex SET 1 0.500000` has a similar interpretation, except that the attribute sex has values taken from a set. In such a case, the cutpoints come from the rational training file heart.rtr. A third example are the two lines

```
chol UNCERTAIN 2 232.500000 [ 231.000000 , 233.000000 ]
         292.000000 [ 288.000000 , 302.000000 ]
```

which say the following. The attribute chol has numerical values, the number of cutpoints is 2, and the cutpoints are 232.5 and 292.0. Associated with the first (resp. second) cutpoint is an interval of uncertainty given by $[231.0, 233.0]$ (resp. $[288.0, 302.0]$). The line `ENDATA` terminates the listing of cutpoints.

CAUTION: If the user manually modifies a .cut file or creates a .cut file from scratch, to be used for subsequent training by cutcc with training option `old`, then the .cut file must satisfy the following conditions. First, attributes marked DELETE in the .cut file must also be marked that way in the .rtr file, and *vice versa.* Second, the format of the .cut file must follow the above cases. In particular, each "[", "]", and "," used for intervals of UNCERTAIN cases must be preceded and followed by at least one blank space, and the cutpoint value must fall into the corresponding interval of uncertainty.

Records after the ENDATA terminating the listing of cutpoints specify the logic variables derived from the attributes and certain scaling factors. Here is a representative portion of these records.

```
LITERALS AND RULES
 age_1  ( age > 50.500000 )
-age_1  ( age < 50.500000 )
 sex_1  ( sex < 0.500000 )
-sex_1  ( sex > 0.500000 )
 sex_2  ( sex > 0.500000 )
-sex_2  ( sex < 0.500000 )
.
.
num DELETE
ENDATA

SCALING FACTORS
age     1.000000
sex     SET
cp      SET
trestbps        1.000000
chol    14.000000
.
.
num     DELETE
ENDATA
```

The section `LITERALS AND RULES` lists the logic variables derived from the attributes and defines when they take on the value *True*. For example, variable age_1 is derived from attribute age and is *True* if age $>$ 50.00000.

The section `SCALING FACTORS` lists certain values called scaling factors. These factors are only defined for attributes without the DELETE or SET option. Assume an attribute is of that kind.

If the attribute is listed with the UNCERTAIN option, then the scaling factor is the maximum of the widths of the uncertainty intervals around the various cutpoints of the attribute.

If the attribute does not involve the UNCERTAIN option, then the scaling factor is also a maximum of widths defined for the cutpoints, but this time each width is the difference between two training values closest to a cutpoint: One of the value is above the cutpoint, and the second one is below the cutpoint.

For example, the line

```
chol    14.000000
```

specifies that the scaling factor for the attribute chol is 14.000000.

The user can ignore both the `LITERALS AND RULES` and `SCALING FACTORS` sections. The data of the sections are processed by program subcc described in Chapter 9. In particular, the scaling factors are used during so-called expansions of explanatory polyhedra. For a summary of that process, see the section of Chapter 9 covering lsqccparams.dat.

Visualization File

To help the user evaluate the selected cutpoints, program cutcc outputs a visualization file that displays the cutpoints together with the numerical values for each attribute. Specifically, for each attribute, the numerical values are sorted and then listed in ascending order, together with the letter A or B. That letter tells whether the numerical value occurs in a record of the rational training set A or B.

Here is the visualization file heart.vis computed by program cutcc for the rational training file heart.rtr of Leibniz/Cutcc/Makedata.

```
*******************************************************
Attribute: age
*******************************************************
1) A 29.000000
2) A 34.000000
```

```
3) B 35.000000
.
.
44) B 50.000000
45) A 50.000000
---------------------------------------------------
46) B 51.000000
47) A 51.000000
.
.
*******************************************************
Attribute: sex
*******************************************************
1) B 0.000000
2) B 0.000000
.
.
ENDATA
```

Under the heading `Attribute: age`, the sorted values for that attribute are listed. Thus, the line `1) A 29.000000` is produced by the smallest value, 29.0, and that value occurs in a record of the set A. Moving down, we come to the line `45) A 50.000000`. That line is separated from the next value, given by `46) B 51.000000`, by a line of dashes. These dashes correspond to a cutpoint, which always lies midway between the two values it separates. In this case, the cutpoint value is $(50.0 + 51.00)/2 = 50.5$, as it should be according to the cutpoint value given for age in the file heart.cut.

Program cutcc selects the cutpoints so that, in the neighborhood of the cutpoint, values of mostly A records are separated from values of mostly B records. The reader may want to scan the entire file heart.vis to confirm that this feature is indeed achieved by the cutpoints for the various attributes.

Values falling into intervals of uncertainty are marked by `U`. For example, we have for the attribute `chol` the following display.

```
******************************************************
Attribute: chol
******************************************************
.
.
121)  B    286.000000
122)  B    288.000000    U
123)  B    288.000000    U
124)  B    289.000000    U
125)  B    290.000000    U
--------------------------------------------------
126)  A    294.000000    U
127)  A    302.000000    U
128)  A    302.000000    U
129)  A    303.000000
.
.
```

The output indicates that the attribute chol has a cutpoint with value (290.0+294.0)/2 = 292.0. The cutpoint has an associated interval of uncertainty ranging from 288.0 to 302.0. Accordingly, any rational value z for chol satisfying $288.0 \leq z \leq 302.0$ is encoded by unavailable for the logic variable corresponding to the cutpoint 292.0.

We have completed the discussion of cutpoints and related files. Next, we cover the output files of program cutcc. These files are used by program lsqcc to produce logic formulas.

Partial Data File

The partial data file contains the logic definitions needed by program lsqcc. Here is the example file heart.ptl of Leibniz/Cutcc/Makedata.

```
*Attribute age 1 cut(s): 50.50
*Attribute sex 1 cuts(s): 0.50
```

```
*Attribute cp 3 cut(s): 1.50 2.50 3.50
.
.
*Attribute thal 2 cut(s): 4.00 5.00

MINIMIZE
partial_log_file

SETS
define cases
pos
neg

PREDICATES
define age_1 on cases
  age_1(pos) truecost = 99
  age_1(neg) truecost = 101
define sex_1 on cases
  sex_1(pos) truecost = 99
  sex_1(neg) truecost = 101
define sex_2 on cases
  sex_2(pos) truecost = 98
  sex_2(neg) truecost = 102
.
.
define thal_3 on cases
  thal_3(pos) truecost = 97
  thal_3(neg) truecost = 103
include heart.prd

VARIABLES
dummy_variable
AFORMULA
BFORMULA
```

```
FACTS
dummy_variable.

ENDATA
```

The first part of the files contains the definition of the cutpoints defining the intervals for each attribute. Each line begins with an asterisk, since it is to be viewed as a comment when program lsqcc later processes the file. For example, the line `*Attribute age 1 cut(s): 53.50` says that for the attribute age there is one cutpoint, which is 53.50. For the attribute cp, the line is `*Attribute cp 3 cut(s): 1.50 2.50 3.50` and thus specifies the cutpoints 1.50, 2.50, and 3.50.

We skip a detailed discussion of the remaining records of heart.ptl since Chapter 7 on program lsqcc explains their role. But we do mention that the predicates listed in the partial data file are obtained from the attributes of the rational training file, by attaching an underscore followed by an integer. For example, for the attribute age, there is just one predicate age_1. For the attribute sex, there are two predicates sex_1 and sex_2.

CAUTION: At the end of the PREDICATES section, the line `include heart.prd` specifies the inclusion of the file heart.prd. This file is called the pyrpred file. The file is not defined by the user, but is created by lsqcc. The file specifies pyramid predicates, if any, that have been created previously by program pyrcc. This file must never be modified by the user.

Logic Training File

This file has the format described for logic training files in Chapter 7 on program lsqcc. We list an excerpt of the example file heart.trn to discuss the connection of this file with the rational training file heart.rtr.

```
include heart.ptl
```

```
BEGIN A
age_1
-sex_1 sex_2
cp_1 -cp_2 -cp_3 -cp_4
trestbps_1
.
.
*end rec 1 of line 19
.
.
BEGIN B
.
.
ENDATA
```

The line `include heart.ptl` specifies that the file heart.ptl, which has already been discussed, is to be included.

The line `BEGIN A` signals the beginning of the logic records for the training set A. The records are given next, where each line contains the logic data for one attribute. Thus, the line `age_1` represents the logic data for the value of the attribute age in the first record, while the line `-sex_1 sex_2` gives the logic data for the value of the attribute sex in the first record. The line `*end rec 1 of line 19` tells that the end of the logic data for the first record has been reached. All subsequent lines are interpreted analogously.

Logic Testing File

The logic testing file is derived from the rational testing file using the transformation rules determined when the rational training file was transformed to the logic training file. These rules have been stored in the cutpoint file. Thus, the testing step reads that file and uses it to carry out the transformation from rational testing data to logic testing data.

The interpretation of the logic testing file is identical to that of the logic training file, except that the records `BEGIN A` and `BEGIN B` are not present. Below, we show the logic testing file heart.tst.

```
age_1
-sex_1 sex_2
cp_1 -cp_2 -cp_3 -cp_4
trestbps_1
.
.
*end record 1 of line 1
.
.
ENDATA
```

Error File

The error file cutcc.err contains all error messages and detailed information about all cutpoints not defined by the SET option. Here is an example segment covering one cutpoint.

```
Checking separation of A from B.
Checking separation of B from A.
4 nested record combinations
Iteration 5
Variable = thalach_1
Potential = 4.4
Attribute = thalach
Sigma = 6
Attractiveness = 0.57
Usefulness = 0.52
Separation criterion = 1.09
New cut point = 146.500
Uncertainty interval (used only if UNCERTAIN specified) =
  [ 145.000 , 147.000 ]
Value immediately below cutpoint = 146.000
Value immediately above cutpoint = 147.000
Number of cuts increased from 0 to 1
```

The interpretation is as follows. Iteration 5 investigates attribute thalach and creates logic variable, or attribute, thalach_1.

`Potential = 4.4` measures the overall potential for separation of the cutpoint created in this iteration. Generally, the higher the potential, the more useful the cutpoint is.

`Sigma = 6` is the σ value used in a certain smoothing process called Gaussian convolution. A large value means that much smoothing is done.

`Attractiveness = 0.57` is a relative measure of the abruptness of the change of patterns of rational values at the cutpoint, when compared with changes produced by random noise. Generally, the higher the attractiveness, the less likely it is that the change of pattern is due to random noise. A value less than or equal to 1.0 means that the change of pattern may well be due to random noise.

`Usefulness = 0.52` is a measure of the degree of separation of records of training sets A and B by the cutpoint. Generally, the closer the value to 1.0, the higher the degree of separation.

`Separation criterion = 1.09` is a combination of the values for attractiveness and usefulness.

`New cut point = 146.500` gives the new cutpoint value for the attribute thalach.

`Uncertainty Interval (used only if UNCERTAIN specified) =`
`[ 145.000 , 147.000 ]` says that, if UNCERTAIN has been specified for age, then the associated interval of uncertainty is $[145.0, 147.0]$.

`Number of cuts increased from 0 to 1` means that attribute thalach now has 1 cutpoint.

Prior Knowledge

Often, the formulas computed via programs cutcc and lsqcc must represent not only the information contained in the training sets but also prior domain knowledge The processing described so far in this chapter and in Chapter 7 can be modified to achieve that effect.

We assume that the prior domain knowledge is in the form of if-then rules where the if-condition is a conjunction of terms, each of which is an equality or inequality involving just one variable, and where the conclusion is a claim of membership in one of the populations $\mathcal{A}$ or $\mathcal{B}$. Example: If $(x > 3)$ and $(y < 5)$, then the record is in population $\mathcal{A}$. Here, x and y are numerical attributes of the training sets A and B. We use this example to describe how formulas compatible with the prior knowledge can be learned from the training sets. Derivation of the general approach should then be obvious. We should mention that there are simpler cases of prior knowledge where just certain structural properties of the formulas are claimed. For example, the claim might be that in any A formula, the terms involving attributes a and b may only occur nonnegated. The formulation of this simpler case will be obvious from the subsequent discussion of the more complicated case where the learned formulas must reflect an if-then rule of prior knowledge.

<u>CAUTION</u>: The process below is valid only if all unknown values of the training file are unavailable.

<u>CAUTION</u>: Each B record must contain at least one value of the attributes in the rule of prior knowledge, here of x or y.

The algorithm for handling the rule "If $(x > 3)$ and $(y < 5)$, then the record is in population $\mathcal{A}$" consists of five steps.

First, run program cutcc with option `new` to discretize the given training sets. Add manually to the .cut file the cutpoint 3.0 for x and the cutpoint 5.0 for y. Of course, if such a cutpoint is already present, skip the addition. If the other cutpoints of x (resp. y) use uncertainty intervals, then also define such an interval for the cutpoint just added for x (resp. y). The width of the intervals should be the precision to which x or y are measured.

Second, rerun cutcc with the option `old` to discretize the training data using the modified .cut file.

Third, suppose `x_1`, ..., `x_k`,..., `x_r` are the predicates in the .ptl file that represent the cutpoints of x, where `x_k` corresponds to the cutpoint 3.0. Similarly, let `y_1`, ..., `y_m`,..., `y_n` be the predicates for y, where `y_m` corresponds to the cutpoint 5.0.

Add to the logic training file .trn the following record immediately after the `BEGIN A` statement.

```
x_1
 .
 .
x_k
-y_m
 .
 .
-y_n.
```

Add to the FACTS section of the .ptl file the following statement.

```
if BFORMULA then
     x_1(neg) or ...
     x_k(neg) or
     y_m(pos) or ...
     y_n(pos).  rule_1
```

<u>CAUTION</u>: The name `rule_1` allows diagnosis of potential conflicts between the training sets and the added logic conditions, see below.

Fourth, run program lsqcc. The A formulas, which are *True* on A records, incorporate the prior knowledge due to the record added to logic training file .trn in the third step. The B formulas, which are *True* on B records, contain the prior knowledge due to the logic conditions added to the .ptl file in that step.

CAUTION: Program lsqcc may detect conflicts between A and B records and the logic condition added to the .ptl file. The error messages point to specific records and logic conditions that require correction.

CAUTION: If program lsqcc removes the added A record due to nestedness within a B record, then any such B record is incompatible with the prior knowledge and must be corrected or removed.

Chapter 9

subcc for Discovery

Introduction

Module subcc has programs subcc, sub2cc, and suballcc. Program subcc analyzes rational or integer data and identifies important subgroups and corresponding explanations for specified targets. Program sub2cc invokes subcc, then processes the output of subcc and derives a compact and easy-to-read listing of important subgroups. Program suballcc invokes sub2cc recursively and potentially derives a large number of important subgroups, presented in the format of the sub2cc output.

The input is similar to that for program cutcc, except that there are no data sets A and B. Instead, in a given data set, each attribute is declared to be either a target variable, or an explanatory variable, or both a target and explanatory variable.

Goal is detection of important cases where some range of values of a target can be explained by the values of some explanatory variables. The range of target values so explained corresponds to a subset of the given records called a *subgroup* of the data set. Indeed, the process is a particular case of *Subgroup Discovery*.

The explanations are propositional logic statements where each literal is a linear inequality involving the attributes.

Program subcc first reads the parameter file lsqccparams.dat described later. If a different name is used for that file, then the name must be specified when the program is executed. An example command is

```
Leibniz/Subcc/Code/subcc params.file
```

Once the parameter file has been processed, program subcc repeatedly calls programs cutcc and lsqcc to identify important subgroups.

Program sub2cc requires the same input files as subcc. Indeed, sub2cc calls subcc, then processes the output of subcc to achieve a compact representation of important subgroups.

Program suballcc requires the same input files as subcc and sub2cc. The program calls sub2cc recursively, sifts through the results, and collates the final output in one file that contains important subgroups.

In the remainder of this chapter, the files and steps of program subcc are discussed in detail beginning with the input files called master file and alternate testing file. Toward the end, programs sub2cc and suballcc are discussed.

Alternate Execution of subcc

Program subcc can be executed directly from user code as follows.

First, link all object code of Leibniz/Subcc/Code with the user code. The linking requires the option “-lm”. Second, carry out in the user code the steps specified in the comment section of the source code of subroutine subcc(). The subroutine is included in file subcc.c. Analogously, programs sub2cc and suballcc can be called from user code.

Master and Alternate Testing Files

Program subcc always requires a master file .mst as part of the input. The derivation of subgroups is based on that file. For testing the significance of the subgroups, either the master file is used once more, or an alternate testing file .ats is used. In yet another option, the given master file may be split into a file for training and a second file for testing. Generally, the latter option is preferred, since the split is carried out for each target so that both training and testing file are likely to be representative of the underlying population.

Master File

We explain the master file .mst using the example heart.mst of Leibniz/Subcc/Makedata. Here is that file.

```
ATTRIBUTES
age
sex     SET .5
cp
trestbps
chol    TARGETCASES_1_10_10
fbs
restecg TARGETSET_1_3_3
thalach
exang
oldpeak UNCERTAIN
slope   TARGETINTERVAL_1_5_10 DELETE
ca      CLASS_1 TARGETCASES_1_10_10
thal    CLASS_1 TARGETCASES_1_3_5
num     DELETE

DATA
63 1 1 145 233 1 2 150 0 2.3 3 0 6 0
37 1 3 130 250 0 0 187 0 3.5 3 0 3 0
.
.
57 1 4 165 289 1 2 124 0 1 2 3 7 4
ENDATA
```

If an asterisk * occurs on a line, then that asterisk and the remaining entries on the line are considered comments and are ignored.

The line `ATTRIBUTES` is the first line of the master file. Here, the user defines the attributes of the subsequent records.

Types of Attribute Records

There are four different types of attribute records, as follows.

(1) `attr classes TARGETCASES_x_y_z <cutcc options>`

(2) `attr classes TARGETINTERVAL_x_y_z <cutcc options>`

(3) `attr classes TARGETSET_x_y_z <cutcc options>`

(4) `attr classes <cutcc options>`

In all four cases, `attr` is the name of an attribute, and the option `classes` is either omitted or is a sequence of the form `CLASSd1 CLASSd2 CLASSd3` ..., where `d1`, `d2`, `d3`, ... are any alphanumeric sequences that also may use underscore.

In case (4), the attribute `attr` is to be used only as explanatory variable and never as target variable. Any option of cutcc may be specified, including the DELETE option. In the latter case, variable `attr` is never considered for any explanation.

In case (1), (2), and (3), the attribute `attr` is a target variable. We discuss details of these cases.

Target Option (1)

The option `TARGETCASES_x_y_z` has `x`, `y`, and `z` as positive integers where $1 \leq x \leq y \leq z \geq 3$. The option is interpreted as follows.

A total number of `z` cutpoints are to be computed for the variable `attr`. Of these cutpoints, the ones numbered from `x` to `y` are to be actually used. Each of the actually used cutpoints divides the set of records into a low set and a high set. Program subcc attempts to explain the difference between the two sets. Informally, this is the case of low target values versus high target values.

After the option `TARGETCASES_x_y_z`, any option of cutcc may be specified. If the option `DELETE` of cutcc is used, then the variable `attr` is to be used only as target and not as an explanatory variable. Any other cutcc option implies that variable `attr` is also an explanatory variable.

For example, the record `slope    TARGETCASES_1_5_10 DELETE` specifies that the variable `slope` is a target. A total of 10 cutpoints are to be constructed, of which cutpoints numbered 1 though 5 are to define low/high intervals that are to be explained. The variable `slope` is not to be used in explanations of other targets since the record also specifies the `DELETE` option.

Target Options (2) and (3)

The rules for target Options (2) and (3), repeated here,

```
(2) attr classes TARGETINTERVAL_x_y_z <cutcc options>
(3) attr classes TARGETSET_x_y_z <cutcc options>
```

are the same as for `TARGETCASES`, except that explanations are computed for additional cases.

Option `TARGETINTERVAL`: The additional cases involve all possible pairs of cutpoints. Let c and d be the cutpoints of one such pair, where $c < d$. An explanation is computed for the difference between the records where the values for attribute `attr` fall between c and d, and the records where the values are either below c or above d.

Option `TARGETSET`: The additional cases are those of `TARGETINTERVAL`, except that the pairs are defined by c and d of successive cutpoints.

Class Option

The `CLASS` option is best explained using the following two records.

```
ca      CLASSd1 TARGETCASES_1_10_10
thal    CLASSd1 TARGETCASES_1_3_5
```

Suppose the target cases for the attribute `ca` are being evaluated. The option `CLASSd1` then specifies that the attribute `thal` cannot be used to explain the target values of `ca`. Analogously, when target values are evaluated for `thal`, then the attribute `ca` cannot be used in the explanations. In general, if the option `CLASSd` has been assigned to any number of attributes, then any explanation of target values for one of the attributes with `CLASSd` option cannot use any other attribute with the `CLASSd` option.

In the example, both attributes with `CLASSd1` option have a target option. But this need not be so. That is, an attribute may have a `CLASS` option and not a target option.

Any number of `CLASS` options may be specified with a particular attribute.

The data portion of the master file .mst begins with the record `DATA` and ends with the record `ENDATA`. The records are exactly in the format required by program cutcc. That is, each record lists the values for each variable, in order, with blanks or tabs separating values. Unknown values are represented by the question mark `?`.

Alternate Testing File

The alternate testing file .ats is very similar in format. We list the file heart.ats of Leibniz/Subcc/Makedata as example.

```
*ATTRIBUTES
*age
*sex SET .5
*cp
*trestbps
*chol TARGETCASES_1_10_10
*fbs
*restecg TARGETSET_1_3_3
*thalach
*exang
*oldpeak UNCERTAIN
*slope TARGETINTERVAL_1_5_10 DELETE
*ca CLASS_1 TARGETCASES_1_10_10
*thal CLASS_1 TARGETCASES_1_3_5
*num DELETE
*
*DATA
52 1 1 152 298 1 0 178 0 1.2 2 0 7 0
42 0 4 102 265 0 2 122 0 0.6 2 0 3 0
.
.
58 1 4 114 318 0 1 140 0 4.4 3 3 6 4
ENDATA
```

Note that all records up to and including `*DATA` are the same as those of the master file .mst, except that they are commented out by the asterisk `*`. In principle, the records listing the attributes may be omitted. It is strongly recommended that they be included so that later it can be easily determined which attributes and options were involved. The data records have the same format as the master file records. The final record is `ENDATA`.

Sort and Split Option

The user may also request that the given master file .mst be split into a training file and a testing file. The option is

```
sort and split with ats/mst ratio (0, 1, 2, ...)  = <k>
```

where k is a nonnegative integer. If $k = 0$, the split option is not used, and the given .mst file and, if supplied, .ats file are used as described above. If $k \geq 1$, then the .mst file is split into a training file and a testing file. The split is determined for each target as follows. First, the records of the .mst file are sorted according to values of the target. In the order of the sorted records, one record is removed for training, and the next k records are removed for testing. The two removal steps are repeated until the sorted file has been exhausted. The split has the attractive feature that the entire range of target values is covered in both the training file and the testing file. If any .ats file is given, then that file is completely ignored.

Target File

The user may directly define target cutpoints and intervals in the target file .tgt. In that situation, the identification of targets in the master file may be done using just the word `TARGET` instead of the more elaborate options `TARGETCASES`, `TARGETINTERVAL`, and `TARGETSET` listed above.

Here is an example target file.

```
TARGETS
attr 1 [ a , b ]             * target version 1
attr 2 [ a , b ]  [ c , d ] * target version 2
.
.
ENDATA
```

The line `attr 1 [ a , b ]` tells that for variable `attr`, one low/high target case is to be created. Specifically, the low set A is defined by the records with `attr` $< a$, while the set B is defined via `attr` $> b$.

The line `attr 2 [ a , b ]  [ c , d ]` tells that this is a middle versus low/high case. The set A consists of the records satisfying $b <$ `attr` $< c$, while the set B consists of the records satisfying one of the conditions `attr` $< a$, `attr` $> d$.

The target file may contain a number of lines for any given variable. The total number of such lines may not exceed 60,000. The limit may be changed by modifying the definition of MAX_TARGET in file subcc.h of Leibniz/Subcc/Code.

lsqccparams.dat

Program subcc uses the parameter file lsqccparams.dat introduced in Chapter 7 and discussed further in Chapter 8. Below, the relevant portion of example file lsqccparams.heart.dat of Leibniz/Subcc/Makedata is listed.

```
*********************************************************
*********************************************************
*** Program subcc for computing subgroups
*********************************************************
*********************************************************
begin subcc
***
*** Input files:
***   master file .mst
***   if testing file option = alternate:
***     alternate testing file .ats
***   if computing option = old tgt:
***     target file .tgt
***   if sort and split ats/mst ratio >= 1:
***     any given alternate testing file .ats is ignored
***     since training and testing files are derived for
***     each target from the input master file .mst
***
*** Output files:
***   if computing option != tgt only:
***     subgroup file .mst.sub
***     if testing file option = alternate:
***      subgroup file .ats.sub
***   if computing option = new tgt:
***     target file .tgt
***
*** Subcc detail directory
*** Has detailed output files.  See manual for interpretation.
*** Caution:  The directory is created by program subcc.  It is
*** removed by program subcc at termination unless the option
*** 'keep subcc details directory' is specified below.
*** Include terminating '/' or '\\'
subcc detail directory = /tmp/Subccdetaildir/
***
*** Options:
***
*** Show target processing steps
show target processing steps
***
*** compute subgroups
***   Options:
***   'new tgt':  compute target file .tgt and
***               subgroup file(s) .sub
***   'old tgt':  use existing target file .tgt
***               to compute subgroup file(s) .sub
***   'tgt only':  compute target file .tgt, and stop
compute subgroups (new tgt, old tgt, tgt only) = new tgt
***
*** Specify master2masterABTarget program for splitting master file
*** into masterAB file according to selected target.  Must include
*** complete path.
*** If the option is commented out, program master2masterABTarget
*** of Leibniz/Prpcc/Code is used.  That program uses the target
*** file information directly, without any modification of rules.
* master to masterAB program =
***
*** Selection of subgroups
***   Options:
***   'master':     selection uses .mst file; creates .mst.sub file.
***   'alternate':  selection uses .mst and .ats files; creates
***                 .mst.sub and .ats.sub files.  Significance
***                 threshold for subgroup selection is defined
***                 to be 0 for .mst case.
```

```
***
use (master, alternate) testing file = alternate
***
*** Sort and split input master file .mst
*** If ats/mst ratio = 0:
***   Sort and split are not done, and the input .mst file and,
***   if applicable, the input .ats file are used.
*** Else:
***   For each target, the input .mst file is sorted by target values
***   and split according to the specified ats/mst ratio = k > 0 into
***   a training file and testing file, where each record in the
***   training file corresponds to k records in the testing file.
***   The above testing file option and any input testing file .ats
***   are ignored.
***
sort and split with ats/mst ratio (0, 1, 2, ...)  = 1
***
*** Size of subgroup definitions
*** Options:
***   'max':  Amax and Bmax clauses are used for subgroup definitions
***   'min':  Amin and Bmin clauses are used for subgroup definitions
size of subgroup definitions (max, min) = min
***
*** Output variations of subgroups
*** For given target number and stage, displays all factors
*** produced by the clauses of applicable Amin and Bmin
*** formulas.
output variations of subgroups
***
*** Keep subcc detail directory when program subcc terminates
keep subcc detail directory
***
*** Threshold for attribute importance.  If no value is
*** specified, 0.55 is used
***
```

```
attribute importance threshold = 0.55
***
*** Maximum number of attributes used in explanations.
*** If no value is specified, 3 is used
***
max number of attributes used = 3
***
*** Significance measure of subgroups
*** Options:
*** 'unusualness':  1-Prob(alternate random process creates subgroup)
*** 'accuracy':     (correctness within group +
***                  correctness outside group)/2 =
***                 (fraction of subgroup within group +
***                   selectivity)/2
*** 'both':         average of accuracy and unusualness
significance measure (unusualness, accuracy, both) = both
***
*** Significance threshold for subgroup selection.
*** If no value is specified, 0.90 is used
***
subgroup selection threshold = 0.90
***
*** Maximum number of expansion iterations for
*** computation of explanatory polyhedra
*** Must be nonnegative; if no value is specified,
*** max number of expansions = 0 is used
max number of expansions = 1
***
*** Option used ONLY by sub2cc:  Output all subgroups, or output
*** representatives of logically identical subgroups
output (all, representative) subgroups = all
***
*** Options used ONLY by suballcc:  Suballcc detail directory
*** Has detailed output files.  See manual for interpretation.
*** Caution:  The directory is created by program suballcc.  It is
*** removed by program suballcc at termination unless the option
*** 'keep suballcc details directory' is specified below.
*** Include terminating '/' or '\\'
suballcc detail directory = /tmp/Suballccdetaildir/
***
*** Keep suballcc detail directory
keep suballcc detail directory
***
```

The options have the following interpretation.

The line `subcc detail directory =` should specify the directory that receives all intermediate files produced by program subcc.

The line `show target processing steps` says that all major processing steps are to be shown on the screen. The line should be commented out by an asterisk `*` if that output is not desired.

The line `compute subgroups (new tgt, old tgt, tgt only) =` defines the creation and usage of target files.

If the option `new tgt` is used, then a new target file is produced according to the TARGET specifications in the master file.

If the option `old tgt` is used, a target file .tgt is given and is to be used.

If the option `tgt only` is used, then a new target file is produced according to the TARGET specifications in the master file, and program subcc stops.

The line `master to masterAB program =` allows processing of any target definitions. For the format, see program master2masterABTarget of Leibniz/Prpcc/Code, which handles the above specified target cases. If the line is commented out, which is the usual situation, then program master2masterABTarget of Leibniz/Prpcc/Code is used.

The line `use (master, alternate) testing file = alternate` specifies whether the master file or the alternate testing file is to be used for evaluating the accuracy of the derived subgroup definitions.

If `master` is specified, then the master file .mst is used as test file. If `alternate` is specified, then the alternate testing file .ats is used as test file.

The line `sort and split with ats/mst ratio (0, 1, 2, ...) = 1` specifies by the value $k = 1$ of the right hand side that for each target case, the .mst file is to be split into a training file and a testing file. Specifically, for each target, the input .mst file is sorted by target values and split according to the specified ats/mst ratio = k into a training file and testing file, where each record in the training file corresponds to k records in the testing file. The above testing file option and any testing file .ats are ignored. If the ats/mst ratio k is specified to be 0, then the sort and split option is ignored, and the above rules about .mst and .ats files apply.

The line `size of subgroup definitions (max, min) = min` defines whether the derived subgroup definitions have a maximum or minimum number of logic terms. Typically, the case of minimum number of terms is preferred.

The line `output variations of subgroups` says that all variations of subgroup definitions obtainable by the program are to be explored. Typically, such exploration is useful, but can be turned off by commenting out the statement.

The line `keep subcc detail directory` says that at the end of program subcc, the contents of the subcc detail directory are not to be erased. It is recommended that this option is specified since analysis of the detail files allows further investigation of subgroups. If the statement is commented out, then the files of that directory are erased. Note that any subsequent subcc execution always removes those files.

The line `attribute importance threshold = 0.55` defines the threshold for selection of important variables in the definition of subgroups. Specifically, the threshold is used in feature selection step, where 40 formulas are evaluated for deciding significance of attributes. The significance values are in the 40partial.vot file. If significance $\geq$ threshold, then the attribute is used for computing explanations via 4 formulas. The stated value, 0.55, generally is a good choice.

The line `max number of attributes used = 3` limits the number of attributes that may be used in explanations of any target. The stated value, 3, generally is a good choice.

The line `significance measure (unusualness, accuracy, both) = both` defines how significance of subgroups is measured. If the option is `unusualness`, then the significance is 1 minus the probability that a certain random process creates the subgroup. If the option is `accuracy`, then the significance is equal to the correctness within group + correctness outside group)/2, which is (fraction of subgroup within group + selectivity)/2. If the option is `both`, then the average of the two measures is used. Generally, `both` is a good choice.

The line `subgroup selection threshold = 0.90` specifies the significance threshold above which subgroups are accepted. Generally, a threshold ≥ 0.90 is a good choice. But special cases may call for lower values. However, values below 0.80 should be avoided.

The line `max number of expansions = 1` limits the complexity of the terms of the logic formulas describing subgroups. If the max number of expansions is demanded to be 0, then each term is an inequality involving just one variable. If the max number is 1, than each inequality may utilize up to two variables. If the max number is 2, then each inequality may utilize up to four variables. Values beyond 2 should be avoided.

The line `output (all, representative) subgroups = all` is discussed below in Section Program sub2cc.

The line `suballcc detail directory =` specifies the directory that receives all intermediate files produced by program suballcc.

The line `keep suballcc detail directory` says that at the end of program subcc, the contents of the suballcc detail directory are not to be erased. It is recommended that this option is specified since analysis of the detail files allows further investigation of subgroups. If the statement is commented out, then the files of that directory are erased. Note that any subsequent suballcc execution always removes those files.

Additional Options of lsqccparams.dat

There are some specifications for lsqcc and cutcc that must be carefully selected when subcc is executed.

The user must specify the Leibniz directory on the line
`Leibniz directory =`

The option
`missing entries (absent, unavailable) = unavailable`
must be used, since program cutcc called by subcc generates cutpoints with the UNCERTAIN option. See Chapter 7 for details.

For program cutcc, the user should select appropriate values for

```
max cuts =
fraction A population =
cost type A error =
cost type B error =
```

The following choices are recommended.

`max cuts =` : The value 1 is usually a good choice, but the higher value 2 may at times be appropriate. Values larger than 2 should be avoided.

`fraction A population =` : 0.5 usually is a good choice. Leaving the value undefined implies use of actual ratios in training data and may or may not be appropriate.

`cost type A error =` and `cost type B error =` : Typically, the value 1.0 for both cases is a good choice in the absence of specific cost data.

Computation and Selection of Subgroups

The file algorithm.txt in subdirectory Leibniz/Subcc/Doc contains details of the mathematics underlying program subcc.

Output Subgroup Files of Program subcc

The output files .mst.sub and .ats.sub are self-explanatory, so we skip a detailed discussion. Another reason is that these files usually are too detailed to be of use. Instead, the use of the output files of program sub2cc or suballcc, discussed next, is recommended.

Program sub2cc

Program sub2cc in Leibniz/Sub2cc/Code is an enhanced version of program subcc. It uses the same parameter file as subcc, but produces a simpler and easier-to-interpret output file .sub2. The file lists the subgroups in decreasing order of significance. Declare two explanations to be logically identical if they become the same upon suitable adjustment of numerical values. If several logically identical explanations are computed, then the line
`output (all, representative) subgroups = representative`
of the file lsqccparams.dat specifies that the explanation with highest significance is listed and all others are skipped. This rule avoids listing of very similar explanations. On the other hand, if the line is
`output (all, representative) subgroups = all`,
then all logically identical explanations are listed. The latter display tells to what the extent an explanation persists when bounds of a given target attribute change.

If an alternate testing file .ats is used, then the output may also be used for validation of factors and formulas. Here is the first portion of an example file, listing results for the subgroup with highest significance.

```
SORTED SUBGROUPS

NOTE: Alternate test file is used for testing.
Output may be used for validation of
factors and formulas.

Subgroup definitions are of MIN size
Significance measure = both unusualness and accuracy

Output all subgroups

*****************************************
SUBGROUP 13 significance = 0.985002
```

```
TARGET 15 STAGE 1  'low' versus 'high'

ATTRIBUTE ca
  'low' definition:   (ca < 2.500000)
  'high' definition:  (ca > 2.500000)

  Factors:
   2 factors jointly imply 'low':
                ( cp < 3.500000 )
                ( thalach > 146.500000 )

TESTING significance = 0.971279

    Coverage for 'low' target values (= sensitivity):
      52 out of 138 'low' records correctly identified
      (= 37.7%)

    Coverage for 'high' target values (= specificity):
     10 out of 11 'high' records correctly identified
     (= 90.9%)

  Total for factors = 128.6%
  Unweighted average for factors = 64.3%
  VC dim estimate = 2

TRAINING significance = 0.998724

    Coverage for 'low' target values (= sensitivity):
     67 out of 141 'low' records correctly identified
     (= 47.5%)

    Coverage for 'high' target values (= specificity):
     9 out of 9 'high' records correctly identified
     (= 100.0%)
  Total for factors = 147.5%
  Unweighted average for factors = 73.8%
  VC dim estimate = 2

  Define N = number of 'high' cases satisfying above factors
  Testing has produced value 11 - 10 = 1 for N
  Probability[ N <= 1 | alternate hypothesis ] = 0.056174

  Guide for Selection of Additional Test Records

    No.  of Test   Est. N in   Est. Probability Given
    Records of     Subgroup     Alternate Hypothesis
    Type 'high'
        10             1            0.080381
        20             2            0.010629
        30             3            0.001561
        40             4            0.000240
        50             5            0.000038
        75             7            0.000000
       100             9            0.000000

*******************************************
```

The significance of the subgroup is 0.985002, which is relatively high and thus indicates that the subgroup is quite important.

The subgroup is characterized by 'low' `ca` values defined by the inequality `ca < 2.500000`, and by the condition
`( cp < 3.500000 ) & ( thalach > 146.500000 )`

The `TESTING` and `TRAINING` sections provide the size of the subgroup and the extent to which the condition
`( cp < 3.500000 ) & ( thalach > 146.500000 )`
is not satisfied by the 'high' cases. Indeed, for the collection of TESTING records, 52 of the 138 'low' TESTING records are in the subgroup. Furthermore, 10 out of the 11 'high' records are classified correctly by the condition

```
( cp < 3.500000 ) & ( thalach > 146.500000 )
```

so $11 - 10 = 1$ of the 11 records are erroneously declared to be 'low.'

The line `VC dim estimate = 2` says that the Vapnik-Chervonenkis (VC) dimension of the learning machine used by subcc, as constrained by the parameters of the parameter file lsqccparams.heart, is estimated to be equal to 2. Roughly speaking, if both the training error and the VC dimension are low, then the characterization of the subgroup by the given formula is likely to be reliable.

The section

```
Guide for Selection of Additional Test Records
```

tells for various values of N the following, where we use the line

```
10              1             0.080381
```

as an example. If $N = 10$ additional test records of type 'high' are obtained, then it can be expected that 1 record will be classified by the above condition

```
( cp < 3.500000 ) & ( thalach > 146.500000 )
```

erroneously as 'low.' Now consider an alternate hypothesis where the above formula is claimed to be useless and makes just random choices. Under that alternate hypothesis, classification of 1 record as 'low' and the other 9 record as high occurs with probability 0.080381. The smaller the probability, the more we would be inclined to reject the alternate hypothesis and consider the classification formula to be useful.

Suppose we want to reject the alternate hypothesis at level 0.0001. Then the lines

```
        40              1             0.000240
        50              1             0.000038
```

estimate that we cannot expect to achieve that goal with $N \leq 40$, but likely will do so with $N = 50$.

Program suballcc

Program suballcc in Leibniz/Suballcc/Code calls program sub2cc for individual targets with increasing deletion of variables. This produces a potentially large number of important subgroups. The output files produced by the sub2cc calls are simplified and then combined into one .suball output file.

The input files have the same format as those for program sub2cc. See the above section about details about these files.

The output file .suball has the same format as the above file .sub2.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Chapter 10

matchcc for Matching

Introduction

Program matchcc accepts a training file and a testing file, each of which contains a collection of vectors of same dimension, indeed is defined for the same set of attributes. A proper subset of the attributes is marked with the label DELETE.

For each record v in the input testing file, program matchcc finds records w in the input training file that are good matches of v using L_∞ distance of the values of non-DELETE attributes. Program matchcc then estimates values of the DELETE attributes of testing record v from the selected training records w.

Program matchcc is particularly effective when the marked attributes viewed as functions of the other attributes have irregular or disconnected contours that defy approximation by regression approaches.

CAUTION: Do not change the C code in matchcc.c to modify the format of input or output files, since program matchcc is used by program redcc.

Version 16.0 1 July 2015

Execution

Program matchcc does not use a parameter file. Instead, the input and output files are specified on the command line. An example call is

```
Leibniz/Matchcc/Code/matchcc \
    train.mst test.mst \
    match.result match.error match.learn \
    match.solution average.error
```

All arguments of the call are file names, with the following interpretation.

- train.mst: training file (input)
- test.mst: testing file (input);
- match.result: matching results file (output)
- match.error: matching error file, with matching errors and matching distances (output)
- match.learn: extended testing record file with matching errors (output)
- match.solution: extended solution record file with matching errors (output)
- average.error: average error as percentage of range of values (output)

The output files are not independent. Indeed, much information is duplicated. Several files are offered so that a user program need not parse a file to get specific output, and instead just reads one of the files.

The files are discussed next in detail.

Input Training and Testing Files

The input training and testing files are structured like the data files for the programs of module subcc. That is, a leading ATTRIBUTES section contains attributes names, and a DATA section has records. Each record in the DATA section has values for the attributes.

Here is the file small.train.mst in Leibniz/Matchcc/Makedata as an example.

```
ATTRIBUTES
obj_0    DELETE
obj_1    DELETE
xvalue_0
xvalue_1
DATA
 4     8    10    10
.
.
ENDATA
```

The testing file small.test.mst is as follows.

```
ATTRIBUTES
obj_0    DELETE
obj_1    DELETE
xvalue_0
xvalue_1
DATA
 7    12    15    16
.
.
ENDATA
```

The ATTRIBUTES section of the training file has obj_0 and obj_1 marked with DELETE. Accordingly, the values of these attributes in the testing files are to be estimated from the remaining attributes of the testing files and from all data of the training file.

CAUTION: The ATTRIBUTES section in the testing file is completely ignored by program matchcc. Hence, the testing file could also have the format of .ats files, where the ATTRIBUTES section is commented out by asterisks `*`. The entire ATTRIBUTES section could also be omitted from the testing file without any effect on execution of matchcc.

CAUTION: In the records of the DATA section, all numbers must be integers ranging from 0 to at most 10^6. If other data are to be used, say rational data, then they must first be converted by scaling and rounding to integers falling within the demanded bounds. For example, if rational values for an attribute range from 0.0 to 1.0, then multiplication by 10^6 and rounding gives the desired integer values. The required range for the integer values implies that the precision of the input data is six digits.

If the user wants to remove an attribute in the training and testing from consideration, the desired effect is achieved by marking the attribute with `DELETE * DISCARD`. Effectively, the option allows removal of attributes from the matching process without reformulation of the DATA section.

Output Files

Program matchcc produces a number of output files. They are discussed below using the output files of the sample run with input files small.train.mst and small.test.mst.

match.result File

The match.result file provides comprehensive output information for each testing record.

```
ATTRIBUTES
obj_0    DELETE
obj_1    DELETE
xvalue_0
xvalue_1
DATA
Test = 1
 7      12      15      16
 8      16      15      16
-1      -4       0       0
Max Absolute Difference = 4   Sum Absolute Difference = 5

Test = 2
10      14      26      14
13      26      26      14
-3     -12       0       0
Max Absolute Difference = 12   Sum Absolute Difference = 15
.
.
.
ENDATA

Total of Absolute Differences = 58
```

```
Average Difference per estimated value = 3.625000
as percentage of 10^6 (= max allowed value) = 0.000362%
```

The ATTRIBUTES section is a copy of that section in the training file. In the DATA section, the line

```
Test = 1
```

implies that the subsequent four records are produced by the first testing record. The four records have the following interpretation. The first record

```
 7      12      15      16
```

is simply a copy of the testing record. The second record

```
 8      16      15      16
```

has 8 and 16 as estimated values for the attributes obj_0 and obj_1 of the first testing record, and the original values 15 and 16 for xvalue_0 and xvalue_1. The third record

```
-1      -4       0       0
```

gives the difference between original and estimated values. The fourth record

```
Max Absolute Difference = 4   Sum Absolute Difference = 5
```

says that the max absolute difference between the original and estimated values is 4, and the sum of the absolute differences is 5.

At the end of the file match.result, the lines

```
Total of Absolute Differences = 58
Average Difference per estimated value = 3.625000
as percentage of 10^6 (= max allowed value) = 0.000362%
```

tell that the total of the absolute differences is 58, resulting in an average of 3.625 per estimated value. That average difference is 0.000362% of 10^6, the max allowed value.

The remaining files offer subsets of the information of match.result plus additional results of the matching process.

match.error File

The file match.error has not ATTRIBUTE section. Instead, each line corresponds to a testing record, in the same order given in file small.test.mst.

```
6       4
5      12
.
.
6       2
```

Consider the first record

```
6       4
```

It gives results for the first testing record. The first number, 6, is the L_∞ distance between the testing record and the closest training record. The second number, 4, is the maximum difference between actual and estimated values for the testing record.

match.learn File

The file match.learn is an expanded version of small.test.mst. Specifically, an additional attribute called MaxError is inserted that represents the max absolute difference between the original and estimated testing record.

```
ATTRIBUTES
MaxError TARGETCASES_1_20_20
obj_0    DELETE
obj_1    DELETE
xvalue_0
xvalue_1
```

```
DATA
 4       7      12      15      16
12      10      14      26      14
.
.
ENDATA
```

For example, the first data record

```
 4       7      12      15      16
```

tells that the max absolute difference for the first testing record is 4. That number is followed by all values of that testing record.

CAUTION: Do not use file match.learn as another training or testing file due to the additional attribute MaxError. If such use is intended, replace the option `TARGETCASES_1_20_20` by `DELETE * DISCARD`, which effectively removes that attribute.

match.solution File

The file match.solution has the results of the matching process, including the additional attribute MaxError. But this time, the option is `DELETE * DISCARD`, which effectively removes the attribute.

```
ATTRIBUTES
MaxError DELETE * DISCARD
obj_0    DELETE
obj_1    DELETE
xvalue_0
xvalue_1
DATA
 4       8      16      15      16
12      13      26      26      14
.
.
ENDATA
```

Each data record corresponds to a testing record. In the first data record,

```
 4      8      16      15      16
```

the first number, 4, is the maximum error made when the values of obj_0 and obj_1 were estimated. The subsequent numbers are the estimated values, followed by the values for xvalue_0 and xvalue_1.

average.error File

The file always has just one record. For the example run, the record is

```
Average Error = 0.000004
```

The number 0.000004 is the average of the MaxError values in file match.learn or match.solution, viewed as a percentage of the max values allowed for any variable. That max value is 10^6. Program matchcc uses the constant 10^6 instead of the actual max value of the attribute values so that the average error can be compared with subsequent error values when different DELETE options are assigned and thus different attributes are to be estimate.

Chapter 11

redcc for Reduction

Introduction

Module redcc consists of programs redcc, redgreedycc, and an auxiliary program evaluateSub2file.

Programs redcc and redgreedycc use the same input information and produce output in identical format, but differ in performance quality. Hence, for the discussion of input files, we refer just to program redcc as a matter of convenience.

We need some mathematical background and concepts for the description of redcc.

Mathematical Background

Let X be a convex subset of n-dimensional real space. Define $I = \{1, 2, \ldots, n\}$. Let $f()$ be a vector of m functions each of which is defined on X. The functions are not explicitly given, but we have a sample Y of X where for each $x \in Y$, we are given the vector $f(x)$.

Let R be the set of pairs $(x, f(x))$, $x \in Y$. We also have a second set S structured like R. Later, R will be called the training set, and S the testing set.

We want to reduce I to a smallest possible subset I' and construct an estimating process Q that uses only the training set R, such that the following holds.

Let $(x, f(x))$ be an arbitrary pair of S. Reduce x to x' by restricting the indices to I'. Then the estimating process Q must derive from x' a vector f' of length m such that the L_∞-distance between $f(x)$ and f' does not exceed a specified max error.

Given a training set R, a testing set S, and a max error value, programs redcc and redgreedycc heuristically select the index set I' and provide an estimating process Q that is based on program matchcc.

Input Files

The training and testing sets are encoded essentially in the same format as training and testing sets of the subgroup discovery programs subcc, sub2cc, and suballcc. In particular, the file begins with the ATTRIBUTES section, followed by the DATA section.

Let's look at the example training file methane.plus.mst provided in Leibniz/Redcc/Makedata.

```
ATTRIBUTES
X         DELETE * DISCARD
Y         DELETE * DISCARD
U         DELETE
V         DELETE
P         DELETE
Temp      DELETE
H
OH
O
HO2
H2
.
.
C
C2              TARGETCASES_1_20_20
N2              TARGETCASES_1_20_20
Enthalpy        TARGETCASES_1_20_20
DATA
0 233333 109860 577972 293574 . . . 877900
0 216667 109860 576273 301366 . . . 877901
.
.
75443 586609 702131 22145 11566 . . . 881464
ENDATA
```

Attributes X and Y have option

```
DELETE * DISCARD
```

which signifies that these attributes are completely ignored during the entire solution process.

Attributes U, V, P, and Temp have option DELETE. This means that they cannot be part of the attributes that eventually are used by procedure Q to estimate function values.

The attributes H, OH, O, ... C are candidates for the attributes used by the estimating process Q. Then just above the ENDATA record, the three attributes C2, N2, and Enthalpy have the option

```
TARGETCASES_1_20_20
```

which signifies that these attributes are the functions whose values are to be estimated.

In the DATA section, each record has numerical values for all attributes.

CAUTION: All numbers must be integers ranging from 0 to at most 10^6. If other data are to be used, say rational data, then they must first be converted by scaling and rounding to integers falling within the demanded bounds. For example, if numbers for an attribute range from 0.0 to 1.0, then multiplication by 10^6 and rounding gives the desired integer values. The required range for the integer values implies that the precision of the input data is six digits.

The testing file methane.plus.ats has the same structure as the training file, except that the entire ATTRIBUTE section is commented out by asterisks `*` in column 1.

Execution

The command for execution of redcc with example files methane.plus.mst and methane.plus.ats is

```
Leibniz/Redcc/Code/redcc lsqccparams.methane.plus.dat
```

where the file lsqccparams.methane.dat is in Leibniz/Redcc/Makedata. The file name can be omitted from the command if it is the default `lsqccparams.dat`. The format of the parameter file is discussed shortly.

Execution of redcc is computationally expensive due to numerous runs of program sub2cc. If the run time of redcc becomes unacceptably large, the user may want to execute the alternate program redgreedycc. The method does not rely on sub2cc, and instead is a straightforward greedy method that directly evaluates reduction choices with program matchcc. As a result, redgreedycc may achieve less of a reduction than redcc. On the other hand, Program redgreedycc typically requires at most a few minutes of run time.

Parameter file

Relevant portions of the file lsqccparams.methane.plus.dat are as follows. The problem name near the beginning of the file is specified by

```
*********************************************************
*** File name without extension
 file name without extension = methane.plus
*********************************************************
```

The section covering redcc options is as follows.

```
*********************************************************
*********************************************************
*** Program redcc for reducing dimension of data
*********************************************************
*********************************************************
begin redcc
***
*** Input files:
***   master file .mst
***   alternate testing file .ats
***
*** Output file:
***   optimal attribute file .opt
***
*** Notes:
*** 1. redcc calls sub2cc and thus uses the parameters specified
***    for programs subcc and sub2cc.
*** 2. Use program match to estimate values for additional files.
***    See manual for details.
***
*** Redcc detail directory
*** Has detailed output files. See manual for interpretation.
*** Caution: The directory is created by program redcc. It is
*** removed by program redcc at termination unless the option
*** 'keep redcc details directory' is specified below.
*** Include terminating '/' or '\\'
 redcc detail directory = /tmp/Redccdetaildir/
***
*** Options:
***
*** Show reduction processing steps
 show reduction processing steps
***
*** Keep redcc detail directory when program redcc terminates
 keep redcc detail directory
***
*** Max error fraction allowed.  If no value is specified,
*** 0.01 is used.
***
 max error fraction = 0.01
***
*********************************************************
```

All records starting with an asterisk `*` are comments. The remaining records define the following four options.

`redcc detail directory = /tmp/Redccdetaildir/`

defines the temporary directory where intermediate results are stored. It preferably is a directory on the computer executing redcc and not accessed across a network.

`show reduction processing steps`

triggers displays of all steps on the screen.

`keep redcc detail directory`

retains all files in the redcc detail directory upon termination. This is useful if difficulties are to be analyzed.

`max error fraction = 0.01`

defines the max error allowed when the estimating process Q predicts the function vectors of the testing file. Specifically, for each pair $(x, f(x))$ in the testing file, the function vector f' computed by Q from x' is compared with $f(x)$. The error is the L_∞-distance between f' and $f(x)$, evaluated relative to the max function value, which is 10^6. According to the above option, the max relative error is not allowed to exceed 0.01.

Output File

Program redcc produces the following output file methane.plus.opt.

```
ATTRIBUTES
X       DELETE * DISCARD
Y       DELETE * DISCARD
U       DELETE
V       DELETE
P       DELETE
Temp    DELETE
H       DELETE * in round 10
OH      DELETE * in round 13
O       DELETE * in round 5
HO2     DELETE * in round 1
H2
H2O
O2      DELETE * in round 9
.
.
C2          TARGETCASES_1_20_20
N2          TARGETCASES_1_20_20
Enthalpy    TARGETCASES_1_20_20
DATA
```

The file is interpreted as follows. Almost all attributes are marked DELETE and hence are not used to estimate function values. There are only two exceptions: Attributes H2 and H20. They suffice to estimate the values of attributes C2, N2, and Enthalpy. The round information given for the deleted attributes, for example for OH by

```
OH      DELETE * in round 13
```

tells the importance of the deleted attributes for estimating function values. That is, the higher the round number, the more important the attribute for that estimation. Put differently, if the user decides to use some of the deleted attributed together with the selected ones, then the choice should begin with the attributes having the highest round number.

Estimation for Additional Records

The following process is used when function values are to be estimated for additional records using just values for H2 and H20.

First, append to methane.plus.opt records that have the values of interest for H2 and H20 and have 0 as values for all other attributes. Call the resulting file match.new.test.mst.

Second, replace the attribute records in the training file match.plus.mst by the attributes records of methane.plus.opt. Call the resulting file match.new.train.mst.

Third, execute matchcc with training file match.new.train.mst and testing file match.new.test.mst. The output files of matchcc have the desired estimated function values for C2, N2, and Enthalpy. However, all error information must be disregarded since actual function values are not provided in the testing file.

The run time for processing additional testing files is very short, typically just a few seconds.

Chapter 12

decc for Decomposition

Introduction

Program decc accepts a matrix or graph as input and constructs a decomposition into components that are linked in tree fashion or in linear fashion.

The input is similar to that for the lbcc compiler, except that the FACTS records represent the rows of a matrix or the edges of a graph.

Goal is a decomposition of the input matrix or graph into a number of components that are weakly linked and thus represent key portions of the matrix or graph. There are two ways in which the components may be joined to define the input matrix or graph. For a description, let each matrix or graph component be represented by a *component node* of a *connection graph*. Two component nodes are connected by an undirected *component edge* if in the matrix case the components have a common column, and in the graph case have a common node. In the first way of decomposition, the connection graph is a tree, and in the second way it is a path.

Program decc first reads the parameter file deccparams.dat described later. If a different name is used for that file, then the name must be specified when the program is executed. For example,

```
Leibniz/Decc/Code/decc params.file
```

Once the parameter file has been processed, program decc carries out the decomposition.

In the remainder of this chapter, the files and steps of program decc are discussed in detail, beginning with the input file.

In the interest of clarity, the discussion first assumes that a matrix is to be decomposed. The case of graph decomposition is covered later.

Alternate Execution of decc

Program decc can be executed directly from user code as follows.

First, link all object code of Leibniz/Decc/Code with the user code. The linking requires the option “-lm”. Second, carry out in the user code the steps specified in the comment section of the source code of subroutine decc(). The subroutine is included in file decc.c.

Example of Input Matrix

Program decc requires an input file that specifies the matrix. The extension .mat is used for that file. Different extensions may be used when specified in the parameter file deccparams.dat.

We explain the input file .mat using the example file matrix.mat of Leibniz/Decc/Makedata.

```
SATISFY
matrix_problem
VARIABLES
x1
x2
.
.
x359
x392
FACTS
x1 . c0
x3 . c1
x2 | x100 | x102 | x329 . c2
x4 | x200 . c3
x299 | x330 | x392 . c4
x300 | x331 . c5
x301 . c6
.
.
x81 | x85 | x102 | x249 . c415
x80 | x90 | x267 . c416
ENDATA
```

In general, the file may contain any number of blank lines and lines beginning with the asterisk *. All such lines are ignored by program decc.

The first line `SATISFY` is mandatory and stems from the fact that the program decc has its roots in logic decomposition.

The second line `matrix_problem` specifies the name of the matrix. Any alphanumeric sequence starting with alpha character, plus underscore "_", may be used for the name.

The line `VARIABLES` heads up the section telling the names of the columns of the matrix. The columns are specified one per line by an alphanumeric sequence beginning with alpha character, plus underscore "_".

The line `FACTS` begins the specification of the matrix entries. The rows of the matrix are encoded one by one according their nonzero entries. The actual numerical values are not important.

For ease of explanation, we treat the particular example

```
x2 | x100 | x102 | x329 . c2
```

The nonzero entries of the row are in columns x2, x100, x102, and x329. The entries are separated by the vertical bar "|". The period "." following the entry for x329 signals the end of the row.

After the period, the name of the row is specified as c2. In principle, the name can be omitted, in which case the program assigns a name of the form

@<number>

where number is a positive integer encoded with leading 0s.

The entries for a given row may run over several lines of the input file. Program decc parses the lines until the termination period "." is reached. Following the period, an optional name for the row may be specified. That name must be on the same line as the termination period "."

The last line `ENDATA` tells that the end of the input file has been reached.

deccparams.dat

Before the matrix analysis begins, decc reads the file deccparams.dat. The file specifies the name of the file containing the input matrix and a number of options.

The parameter file consists of keywords, followed by the = sign and user-defined parameters, in free format. Below, the parameter deccparams.matrix.dat file included in Leibniz/Decc/Makedata for the earlier-discussed input matrix matrix.dat, is listed as a typical example.

```
*** Parameter File for Leibniz Decomposition Program decc
***
*** To effectively remove any statement below, place an
*** asterisk in column 1.  See manual for details.
*********************************************************
*** show major steps or all details on screen
show major steps on screen
show all details on screen
*********************************************************
*** Input file with file extension (see below)
input file = matrix.mat
*********************************************************
*** Input directory - specify complete path
input directory = ./
*********************************************************
*** File extensions:
logic/matrix file extension      (default: log) = mat
program/blocks file extension    (default: prg) = blk
error file extension             (default: err) = err
*********************************************************
*** Max size of formulations
colmax (max number of columns)   =   5000
rowmax (max number of rows)      =  25000
anzmax (max number of nonzeros)  = 500000
blkmax (max number of blocks)    =     30
```

```
cutmax (max size of cutset)      =      4
*********************************************************
*** Decomposition process option
***
*** Decomposition, but no analysis of matrix structure
*** Version            Action
***  0  linear decomposition
*** 10  tree decomposition
***
*** Decomposition and analysis of matrix structure
*** Version            Action
***     no structure analysis of blocks
***  1  decomposition into connected blocks only
***     heuristic structure analysis of blocks
***     no permutation of non-structure columns
***  2  (same as #1)
***  3  decomposition into connected blocks only
***     optimal structure analysis of blocks
***     permutation of non-structure columns
***  4  full decomposition
***     heuristic structure analysis of blocks
***     no permutation of non-structure columns
***  5  full decomposition
***     optimal structure analysis of blocks
***     no permutation of non-structure columns
***  6  full decomposition
***     optimal structure analysis of blocks
***     permutation of non-structure columns
***  7  like 4, but decomp.  with max block last
***  8  like 5, but decomp.  with max block last
***  9  like 6, but decomp.  with max block last
***
decomposition process (version0,...)  = version 10
***
*** Cut selection option:
```

```
***    only cols (linear and tree decomposition)
***    prefer cols (linear decomposition only)
***    prefer rows (linear decomposition only)
cut selection (only cols, prefer rows/cols) = only cols
***
*** Block minimum size option:
***  Value k  Minimum size of each block
***    0         cut size + 1
***   > 0        max(cutsize + 1, k)
block min size = 5 ***
*** Output results
output results to program/blocks file
ENDATA
```

If a line starts with an asterisk `*`, the compiler considers the line to be a comment and ignores it.

The lines `show major steps on screen` directs the program to output information about each major step of the decomposition process. If this line is commented out, the program runs silently unless an error termination is encountered. Correspondingly, the line `show all steps on screen` causes all steps to be shown.

The line `input file = matrix.mat` indicates that the matrix in the file machine.mat is to be processed.

The line `input directory = ./` declares that the input .log file is in the same directory where program decc is executed. The program places all output files into the same directory.

The file extensions .mat, .blk, and .err may be modified by a suitable change of the right hand side of the lines

```
logic/matrix file extension      (default: log) = mat
program/blocks file extension    (default: prg) = blk
error file extension             (default: err) = err
```

The new extension names may contain up to thirty-two (32) characters.

The max size of matrices accepted by the compiler as well as the max number of component blocks produced by the decomposition is specified by the lines

```
*** Max size of formulations
colmax (max number of columns)   =   5000
rowmax (max number of rows)      =  25000
anzmax (max number of nonzeros)  = 500000
blkmax (max number of blocks)    =     30
cutmax (max size of cutset)      =      4
```

Here, colmax is the max number of columns of the matrix, rowmax is the max number of rows, anzmax is the max number of matrix nonzeros, blkmax is the max number of components, and cutmax is the max number of rows/columns that must be deleted to obtain components. Each one of the parameters must be at least 2 except for cutmax, which must be at least 3.

CAUTION: Program decc always adds to each formulation at least one, and sometimes more than one, column and row to achieve a certain standard form. The user need not be concerned with that standard form, but should always select colmax, rowmax, and anzmax so that some additional columns and rows can be accommodated. There is no simple formula to predict the required increase, but an increase up to about 50% may be needed. If the selected value for blkmax is too small, decc puts out a warning message saying that blkmax should be increased for a refined decomposition. The warning message does not trigger an error termination, and output is provided for the decomposition obtained so far.

The line

```
decomposition process (version 0,...)  = version 10
```

specifies whether a linear or tree decomposition is desired. In the tree case, `version 10` must be specified. In the linear case, `version 0` must be specified. The remaining option values 1 – 9 can be ignored.

They carry out analysis for logic matrices that in the present context is irrelevant.

The line
`cut selection (only cols, prefer rows/cols) = only cols`
tells which type of decomposition is preferred. If `only cols` is specified, then only columns are deleted to obtain components. If `prefer cols` is specified, then deletion of columns is preferred for finding decompositions. The last case, `prefer rows`, tells that deletion of rows is preferred. If version 10 is selected, which specifies tree decomposition, then the option `only cols` must be used.

The line `block min size = 5` specifies that the number of columns plus the number of rows of each block must be at least 5. If min size = 0 is specified, this option is ignored. The latter effect is also achieved by commenting out the line.

The line `output results to program/blocks file` specifies that the output results are to placed into the program/blocks file .blk.

The last record of deccparams.dat must be `ENDATA`.

Tree Decomposition

CAUTION: Tree decomposition, which is invoked using version 10, allows only for deletion of columns.

Program decc first separates the given matrix into connected blocks. There is a trivial block 1, which contains an artificial row and column, both of which are labeled with an asterisk `*`. That row and column should be ignored.

Any column of the input matrix that does not contain any entry is also assigned to block 1. Also, if a matrix entry is the single entry in a row as well as a column, and thus constitutes by itself a block, then that column is listed in block 1 as well. This rule avoids expansive listing of trivial blocks with single entries. The remaining blocks contain the nontrivial components of the matrix.

Program decc recursively selects one component on hand and decomposes it into two blocks, as follows. Suppose that deletion of columns reduces a given block to two blocks. These columns are the *cut columns* of the tree decomposition. They are displayed in the log of the decomposition process. At the same time, the resulting two blocks are enlarged by additional *clique rows*. For example, an additional row may be labeled `clique.4`. The additional entries in the clique rows prevent that these columns are separated in any subsequent decomposition.

The entire decomposition may be depicted by a tree as follows. Each block and each clique row of the output blocks file produce a component node of the tree. An undirected component edge connects a block node with a clique node if the block contains the row of the clique.

The number of cut columns of any tree decomposition is limited by the value for the cutmax parameter specified in deccparams.dat. The entire process stops when either no block can be further decomposed while observing that parameter and while enforcing that each block must have the specified min size, or the number of blocks has reached the value of the blkmax parameter of deccparams.dat.

Linear Decomposition

Program decc carries out the linear decomposition similarly to the tree decomposition, except that *cut rows* may also occur. The graph structure representing the entire decomposition process then is a path instead of a general tree.

Graph Decomposition

Program decc decomposes graphs as follows. First, the graph is represented by the edge/node incidence matrix, where each node of the graph corresponds to a column of the matrix and each edge to a row. Specifically, if an edge connects nodes x and y, then the corresponding line of the FACTS section is `x | y. ename` where ename is the name of the edge. For graphs, both tree and linear decompositions can be carried out. In both cases, the cut selection option of deccparams.dat should specify `only cols`. In the graph, this corresponds to deletion of nodes.

In contrast to standard graph decomposition approaches where the edges of a clique are added to the nodes of a cutset for each of the two components, program decc adds just one row to each component. That row has entries in the columns of the cutset nodes. Thus, a more compact representation of the component blocks is achieved.

For an example input file, see edgegraph.mat of Leibniz/Decc/Makedata.

Chapter 13

prpcc for Preparation

Introduction

Module prpcc consists of data preparation routines for programs lsqcc, cutcc, subcc, decc, redcc, and matchcc. The routines perform numerous tasks. Here are some examples.

- Convert Excel files to comma separated version and then to master file format, or vice versa.

- Compute ridge regression coefficients from master data.

- Double the number of records of a master file.

- Convert a master file to master AB file.

- Add new attributes to a master file according to some rules.

- Extract from input master file a sample master file using the kmeans++ initialization method.

- Reduce input master file to output submaster file by deleting attributes and records according to specified limits.

- Split master file into training mst file and testing ats file.

- Split master file into training master file and testing ats file with the special division rule that groups of k records are assigned alternately to train.mst and test.ats.

Execution

For a complete list of the subroutines and instructions for use, see file prepare.txt in Leibniz/Prpcc/Doc.

Directory Leibniz/Prpcc/Makedata contains files that have proved useful to communicate file formats and rules to researchers interested in data analysis by Leibniz programs.

Chapter 14

multicc for Optimization

Introduction

Module multicc consists, in alphabetical order, of programs dencon, denpar, gencc, multicc, nomad, nsga, and seqpen. They are produced during installation of the *Leibniz System* when the main program in multicc.c is recompiled several times with different version.h files.

Several of the programs rely on prior work. For the complete license statement, see the file multicc.c in Leibniz/Multicc/Code.

Each program except multicc solves single objective or multiobjective minimization problems. Specifically, dencon, denpar, nomad, and seqpen handle single objective minimization, while gencc and nsga handle multiobjective minimization. Program multicc is the overall control program.

In principle, gencc and nsga can also be used for single objective minimization, but likely deliver inferior performance compared with the programs listed for that purpose.

The programs can be invoked individually, but the preferred method is execution via multicc. That way, a unifying format is used and complex execution sequences can be easily defined and executed. In this manual, we only cover execution via multicc.

The use of gencc or seqpen is no longer recommended due to equivalent or superior performance of combination sequences involving the other programs. For this reason, detailed discussion of gencc and seqpen has been eliminated from the manual.

It is recommended that the first-time user read the two papers "Simple Seeding of Evolutionary Algorithms for Hard Multiobjective Minimization Problems" available at

`http://www.utdallas.edu/~klaus/Wpapers/seed.pdf`

and "Parallelized hybrid optimization methods for nonsmooth problems using NOMAD and linesearch" available at

`http://www.utdallas.edu/~klaus/Wpapers/hybrid.pdf`

The two papers show that (1) single objective minimization can be effectively carried out by a sequence of methods, here involving dencon, denpar, and nomad; (2) multiobjective minimization can be effectively done by nsga when that method is started with solutions of certain single minimization problems.

The multicc module is so designed that additional programs can be readily added to multicc in two ways.

First, additional programs can be compiled and linked directly with multicc code. For that approach, the code implementing program dencon can be used as a guide.

Second, additional stand-alone programs can be inserted into multicc. For that approach, the construction of program nomad via subroutines calling program NOMAD of `http://www.gerad.ca/nomad` is a guide.

Problem Specification

Specification of an optimization problem for multicc execution can be done in two ways.

Explicit Problem Specification

In the first way of problem specification, C code of objective functions and constraints of the problem are inserted into the subroutine test_problem(), which is in the file problemdef.lb.c in directory Leibniz/Multicc/Code. That subroutine is part of each solution program. Subroutine test_problem() already contains formulations of a large number of optimization problems that can be run without user preparation.

For a trial run, create a separate test directory, say Usertest, and execute in that directory

```
Leibniz/Multicc/Code/multicc zdt6 \
     -input NOINPUTFILE -objrange 1 1
```

to solve the problem called zdt6, which has 10 variables, 2 objective functions, and no constraints besides lower and upper bounds on the variables. The option `-input NOINPUTFILE` of the call tells that the objective function values are to be computed by subroutine test_problem(). The option `-objrange 1 1` declares that the range of values of objective 1 is roughly the same as that for objective 2.

The example run produces voluminous screen output that is explained later. For the moment it suffices that the user examines the final portion of the screen output, which is as follows.

```
multicc evaluation counts
               vectors cycles
     phase 1 =   600
     phase 2 =  1198
     phase 3 =  4275
  ───────────────────
  total count =  6073
output in zdt6.params.multi and zdt6.final.multi
```

Phases 1 and 2 in the statistics refer to solution of two single objective minimization problems derived from the input problem zdt6. Each of the two phases is handled by a combination of nomad and dencon. The output of phases 1 and 2 is used as seeds in phase 3, where nsga solves zdt6.

The cited numbers are the number of solution vectors for which objective function and constraint values were computed during the three phases and in total. In particular, 6,073 evaluations of solution vectors were used in the entire run.

The two files zdt6.params.multi and zdt6.final.multi cited on the last line of the output contain the results of the phases. Details are covered later.

During solution of multiobjective problems with 2 or 3 objective functions, multicc produces plots of the Pareto front as default option. The above call has produced such a plot.

Black Box Problem Specification

In the second way of problem specification, a separate C program is created by the user. It accepts an input file containing one or more solution vectors and then outputs a file with objective function and constraint values of these vectors. That user program is called a black box. It may compute the objective function and constraint values by any method, for example, by simulation. The programs of multicc execute the black box via a system() call.

For an example run, the user should copy into Usertest from directory Leibniz/Multicc/Code the files optim.input and optim.blackbox. The latter file is executable. Indeed, during installation of the Leibniz System it was compiled from the file optim.c of Leibniz/Multicc/Code.

Problem optim has 5 variables, 3 objective functions, and 3 constraints. Objective function 1 is to be minimized, while objective functions 2 and 3 are to be maximized. This information is summarized in file optim.input. The file is covered in detail later.

Now execute in directory Usertest

```
Leibniz/Multicc/Code/multicc optim \
    -objrange 1 1 1
```

As before for zdt6, a voluminous, by now somewhat familiar, screen output is produced. When multicc stops after 6,853 evaluations, the following summary statistics are displayed.

```
 multicc evaluation counts
               vectors cycles
       phase 1 =   905
       phase 2 =   924
       phase 3 =   480
       phase 4 =  4544
   -------------------
   total count =  6853
 output in optim.params.multi and optim.final.multi
```

In addition, a plot of the Pareto front is displayed for the 3 objective functions. The plot correctly shows the values for objective function 1, which is to be minimized. But for objective functions 2 and 3, which are to be maximized, the values are scaled by -1. This is due to the fact that multicc scales by -1 any objective function that is to be maximized, thus obtaining an equivalent minimization case. The plot displays the scaled values.

The above runs were made with the default settings of multicc, which assume that the problem is difficult. We know that problem optim is easy since it is just a linear program. When that knowledge is taken into account and appropriate options are invoked, computational effort is drastically reduced. To see this, the user should execute

```
Leibniz/Multicc/Code/multicc optim \
     -objrange 1 1 1 -sequenceminone \
     nsga dencon -denconcoordinate 1
```

The total evaluation count is 1,799 and thus constitutes a 74% reduction from the earlier count of 6,853. The example shows that careful selection of options can make a substantial difference in computing effort.

Parallelization

All programs except for the eliminated gencc and seqpen allow parallelized execution using processors of multicore computers.

The parallelization applies to the evaluation of solution vectors and not to the programs themselves. Thus, it is useful when black box evaluation is employed and the black box is slow. In that case, copies of the black box can be invoked in parallel by the programs of multicc.

For the parallelization, cores of the machine running multicc as well as cores of other machines can be used with minimal preparatory effort by the user.

To see the effect of such parallelization, the user should copy the file list.processor.24 of Leibniz/Multicc/Code into Usertest and then solve zdt6 with 24 processors with the command

```
Leibniz/Multicc/Code/multicc zdt6 \
    -input NOINPUTFILE -objrange 1 1 \
    -processor list.processor.24
```

Here are the statistics displayed at the end of the run.

```
 multicc evaluation counts
              vectors  cycles
      phase 1 =   869     123
      phase 2 =  1410     239
      phase 3 =  5197     238
  ---------------------------
   total count =  7476     600
 output in zdt6.params.multi and zdt6.final.multi
```

The total number of evaluations is 7,476, and the total number of cycles in which 24 processors or less were invoked, is 600. Thus, the 6,073 evaluations required by the run on a single processor have been reduced roughly by a factor 10 when 24 processors were available.

For the black box case, execution of

```
Leibniz/Multicc/Code/multicc optim \
    -objrange 1 1 1 \
    -processor list.processor.24
```

results in the following final statistics.

```
 multicc evaluation counts
               vectors  cycles
      phase 1 =  1075     279
      phase 2 =   904     235
      phase 3 =   493     135
      phase 4 =  4547     208
   ___________________________
   total count =  7019     857
 output in zdt6.params.multi and zdt6.final.multi
```

The total number of evaluations is 7019, and the total number of cycles in which 24 processors or less were invoked, is 857. Thus, the 6,853 evaluations required by the run on a single processor have been reduced roughly by a factor 8 to 857 cycles when 24 processors are available.

Let's repeat the run with the options that exploit the fact that the latter problem is easy. With the additional processor option, we execute

```
Leibniz/Multicc/Code/multicc optim \
    -objrange 1 1 1 -sequenceminone \
    nsga dencon -denconcoordinate 1 \
    -processor list.processor.24
```

Here are the statistics.

```
multicc evaluation counts
              vectors  cycles
     phase 1 =   178      33
     phase 2 =   120      22
     phase 3 =    93      17
     phase 4 =  1459      68
  ____________________________
  total count =  1850     140
output in zdt6.params.multi and zdt6.final.multi
```

The run requires 1,850 evaluations and 140 cycles. Thus, the 1,799 evaluations required earlier by the run on a single processor have been reduced roughly by a factor 13 to 140 cycles when 24 processors were available.

Screen Output

For discussion of the screen output, the user should capture that output for the first example problem, by running it once more with

```
Leibniz/Multicc/Code/multicc zdt6 \
     -input NOINPUTFILE -objrange 1 1 > results.txt
```

The user can then compare the file results.txt with the subsequent explanations of the screen output.

Throughout the file, date/time stamps are inserted that demark the various steps. Below, we omit that information.

The screen output begins with a preamble that specifies the version by the compilation date.

```
          Leibniz System: Multicc Module
                Version: 16.0
                Compiled: Feb 11 2015
             Copyright 2015 by Leibniz
                Plano, Texas, U.S.A.
```

Next, license and copyright statements follow for certain programs used by the multicc module, together with relevant references.

Processing of the problem zdt6 begins with the statements

```
 Problem name = zdt6

 gOption.input = NOINPUTFILE
 gOption.initsol = NOINITSOLFILE
 gOption.output = zdt6.output
.
.
```

where the values of all options are listed. These options are discussed in detail later, so we skip coverage here. Processing of the problem begins with the statement

```
**** Phase 1 *** Iteration 1 *** Method nomad ****
```

Recall that in phases 1 and 2 certain single objective minimization problems are solved. Each phase involves one or more iterations. The above record marks the beginning of iteration 1 of phase 1, to be carried out by nomad.

The next lines give the options used by nomad, the entire screen output produced by nomad, and finally the command by which multicc has called nomad.

```
 Problem name = zdt6

 Basic Options:
 gOption.input = NOINPUTFILE
 gOption.initsol = NOINITSOLFILE
 gOption.output = zdt6.output
.
.

Processing of input, params, and initsol files done
Number of retained input candidate records = 0

NOMAD - version 3.6.2 - www.gerad.ca/nomad
.
.
blackbox evaluations   : 340
best feasible solution : ( 0 0 0 0 0 0 0 0 0 0 ) h=0 f=1

 at least one feasible solution found
 best feasible solution in file final_pop.out

 Command:
 ~/Leibniz/Multicc/Code/nomad zdt6 -input NOINPUTFILE
 -objrange 1 1 -params zdt6.params.multi -objfactor 1 0
 -maxeval 1000 -stopcriterion optimal -initialmesh 0.5000
```

At this point,

```
**** Phase 1 *** Iteration 2 *** Method dencon ****
```

begins. That is, we are still in phase 1, but are going to the second iteration, where dencon starts with the results just produced by nomad and then solves the same single objective minimization problem.

The subsequent screen output for the iteration is similar to that above, except that details produced by dencon instead of nomad are shown.

Next is the statement

```
**** Phase 2 *** Iteration 1 *** Method nomad ****
```

which says that phase 2 begins, where the second single objective minimization case is solved in the first iteration with nomad.

The second iteration of phase 2 starts with

```
**** Phase 2 *** Iteration 2 *** Method dencon ****
```

At the conclusion of iteration 2 of phase 2, the solutions of the two single objective minimization problems obtained by the above process are passed to the final phase handled by nsga. That phase starts with the statement

```
**** Phase 3 ****************** Program nsga ****
```

For that phase, all options used by nsga are displayed, followed by the screen output of the iterations nsga.

At the end of the nsga output, the evaluation counts for the three phases are listed as shown earlier.

We are ready to discuss key files used by multicc.

Input, Initsol, Blackbox Files

Multicc uses files for all inputs, data transfers between program, and outputs. This section discusses files provided by the user. The files have default names that can be replaced by user-selected names. The default cases are identified by various suffixes such as "'input" or "initsol." Below, we say "input file" or "initsol file" etc. to simplify the terminology.

Input File

If a problem is not specified in file problemdef.lb.c, an input file must be created that supplies problem parameters. In addition, a black box program must be provided. File optim.input is discussed as an example input file. The black box program is covered later.

```
# input for problem optim
# total number of variables = 5
# number of integer variables = 0
# number of objectives = 3
# number of constraints = 3
# objective: obj1 min -tolerance 0.1
# objective: obj2 max -tolerance 0.2
# objective: obj3 max -tolerance 0.05
# constraint: condition1 <= 5 -tolerance 0.01
# constraint: condition2 >= 2 -tolerance 0.001
# constraint: condition3 <= 10 -tolerance 0.05
# param: var1 1 5
# param: var2 3 30
# param: var3 5 15
# param: var4 7 28
# param: var5 2 10
# endformat
```

Below are detailed explanations of the records.

`# input for problem optim`: The name of the problem is optim.

`# total number of variables = 5`: the total number of variables of the problem is 5.

`# number of integer variables = 0`: There are no integer variables. At present, this must be so since some programs cannot handle integer variables explicitly. Instead, such variables are handled by suitable rounding done within the black box. More about that in a moment.

`# number of objectives = 3`: There are 3 objective functions that are to be minimized.

`# number of constraints = 3`: There are 3 constraints besides the lower and upper bounds on the variables.

`# objective: <name> <goal> -tolerance <alpha>`: Each such line specifies an objective function. The name of the function is followed by goal = max or min, depending on whether the function is to be maximized or minimized. The term `-tolerance <alpha>` is optional. If present, any difference of two objective values that does not exceed alpha is considered insignificant. The default tolerance FUNCTION_PRECISION is defined in file sample.lb.h.

For example, `# objective: obj1  min -tolerance 0.1` says that one of the objective functions is obj1, it is to minimized, and absolute value differences below 0.1 are considered insignificant.

CAUTION: The default tolerance FUNCTION_PRECISION in file sample.lb.h or the tolerance values explicitly specified in the input file via -tolerance can be overriden at execution time by the option -objtolerance, which is covered later.

`# constraint: <name> <inequality> <rhs value> -tolerance <beta>`: Each such line specifies a constraint. The name of the constraint is followed by inequality specified by `<=` or `>=`. Then rhs is the right hand side constant of the constraint. The term `-tolerance <beta>` is optional. If present, beta is the tolerance value for the constraint. That is, if the constraint is violated by at most beta, then it is still considered satisfied. The default tolerance is given by CONSTRAINT_PRECISION in file sample.lb.h.

For example, `# constraint: condition1 <= 5 -tolerance 0.01` says that one of the constraints is condition1, the associate inequality is $\leq$, and the right-hand-side constant is 5. If the constraint is violated by at most 0.01, then it is still considered satisfied.

CAUTION: The default tolerance CONSTRAINT_PRECISION in file sample.lb.h or the tolerance values explicitly specified in the input file via -tolerance can be overriden at execution time by the option -constrtolerance, which is covered later.

`# param: <name> <low> <high>`: Each such line specifies the name of a variable and the associated lower and upper bounds.

For example, `# param: var1 1 5` says that one of the variables is var1, with lower bound 1 and upper bound 5.

`# endformat`: terminates the listing of parameters.

Comments may be interspersed among the above records. They are indicated by `##` in the first two columns of the row.

Below the `# endformat` line, any number of records, in any format, may be included. In the example, two explanatory notes are included. All such records are ignored by multicc.

Handling of Integer Variables

Presently, the programs of multicc except nomad cannot process integer variables explicitly. But such variables can be handled approximately by the following rounding procedure.

Let x denote a solution vector produced by any program of multicc, and let z be the subvector of x for which the solution values must be integer.

1. Increase the upper bound for each variable of z by a constant close to 1.0, for example, by 0.9999.
2. All programs deal with x as if the integrality condition for the subvector z did not exist.
3. When a solution vector x produced by any program is evaluated for objective and constraint values, the program test_problem() of problemdef.lb.c or the black box first round down the values for the subvector z to the nearest integers, thus converting x to a vector x', and then evaluate x' for the objective and constraint values.
4. The output solution vectors x are modified by rounding down the entries of subvector z to the nearest integers.

Processor File

For parallel processingprocessor file, multicc module of the evaluation of solution vectors, the user must create a processor file that lists the various directories where execution of the black box copies is to take place. For simplified handling and elimination of access conflicts, multicc places the executable blackbox file into each such directory, together with other files that are needed by that program.

To cover the potential cases in a simple example setting, let's assume the following. Two cores of the machine running the multicc program are to run black box copies in directories path1 and path2. Two additional black box copies are to run on computer comp3 in directory path3 and on computer comp4 in directory path4. Three additional files are needed by each of the black box copies: ufile1, ufile2, and ufile3.

The file processor.txt available in Leibniz/Multicc/Code and listed below handles the described case.

```
# list of computers and paths for processors
# paths must have terminating '/'
# computer = 'home' means computer running main program
# 'fork()' and 'execl()' are used for subprograms
# != 'home' means different processors
# 'ssh' and related commands are used for subprograms
processor home path1/
processor home path2/
processor comp3 path3/
processor comp4 path4/

# list of user files needed for evaluation of solution vectors
# do no include main black box program of '-blackbox' call option
# these files, as well as the blackbox file, must be in the
# directory where multicc is called
userfile ufile1
userfile ufile2
userfile ufile3
```

The input file for the black box, called inbox file, is supplied to the directory containing the black box program. The black box program then processes the inbox file and writes the output into the outbox file in the same directory. Thus, from the standpoint of the black box, inbox and outbox file management involves only local files.

It is important that the black box copies do not create or reference any files outside the directories they reside in. Of course, they may call other programs stored outside the execution directories, for example, for file management. It is up to the user to make sure that the use of such additional programs does not create conflicting demands that cannot be handled by the operating system and lead to deadlocks.

When additional computers are used, the user must arrange for ready access since the Leibniz files use ssh and related commands with a key. For details of the processes carried out by multicc, see subroutine parallelQueryBlackbox() in eval.parallel.c of Leibniz/Multicc/Code.

A number of files named list.processor.⟨n⟩ are provided in directory Leibniz/Multicc/Code for parallel execution of black box copies on the same computer running multicc. Here, n is the number of processors used. If the computer running multicc has less than n cores, the operating system schedules the tasks using the available cores.

Initsol File

The initsol file is optional. It is created by the user and contains starting solutions that are to be used by multicc. Each record is either a specification record telling where the solution values start in a data record, or it is a data record containing a solution vector in rational or exponential format, or is a comment record. Here is optim.initsol as example file. In the data records, blanks or tabs may be used as white spaces.

```
# records: skip 6 entries
# feasible solution
4 13 16.5 7 12.5 5.5 ...
```

The starting point for the solution values in a data record is defined by the line

```
# records: skip <number> entries
```

where number tells how many record entries precede the solution values. For example,

```
# records: skip 6 entries
```

says that the solution values begin after the 6th entry of each record.

At least one specification record must be used, and it must precede all data records. In principle, any number of specification records may be used. Each new specification record redefines the starting point for solution values of subsequent data records.

The user may add comment records at any point, each starting with `#` in column 1. Of course, the user should avoid beginning a comment with `# records: skip` since this identifies a specification record.

If a number of initsol files are in use, it is advisable that the file contains the input file records up to and including `# endformat` as preamble for easy problem identification. This works since each such record starts with `#` and thus is considered to be a comment.

Blackbox File

When a black box is used to compute objective function and constraint values, the user must create an executable blackbox file. The program accepts as input an inbox file and then outputs an outbox file.

Besides the blackbox file, the user must also create the earlier discussed input file containing the parameters of the problem.

The black box program can have various forms; for example, it may be a python program or a C program. An example C program is provided in the file optim.c in Leibniz/Multicc/Code. Discussion of that file is skipped here since it contains detailed comments covering every step. Indeed, it is recommended that optim.c is used as template when black box programs are created. For such use, optim.c contains two sections marked by the comment

```
/*** section below must be modified for specific case ***/
```

where problem-specific code is to be inserted.

The user need not be concerned about the way the black box program is handled. But for completeness, that information is included here.

The black box program is called in subroutine queryBlackBox() or parallelQueryBlackbox(), as follows.

```
<blackboxprogram> <problem> <inbox file> <outbox file>
```

params.multi, final.multi Output Files

On termination, multicc displays on the screen the line

```
output in <problem>.params.multi and <problem>.final.multi
```

where problem is the problem name. The file <problem>.params.multi has the following format.

First, the parameters of the problem are displayed.

Then any number of starting solutions are shown that were computed during the single objective minimization phases of multicc. The format of these solutions is as described for the initsol file, except that additional information gives details about the creation of the solution by multicc. For a detailed explanation, we list a portion that describes one solution for optim. That portion contains several statements starting with `#`, followed by one data record.

```
# phase 1 iteration 0 program nomad completed
# of objectives = 3 , # of constraints = 3, ...
# records: skip 6 entries
# feasible solution
4 13 16.5 7 12.5 ...
```

The line `# phase 1 iteration 0 program nomad completed` tells when the solution was created.

The next line, `# of objectives = 3 , # of constraints = 3, ...` specifies how the subsequent data record is to be interpreted. That is, the first 3 numbers are objective function values, the next 3 numbers are constraint values, and so on. It is important to note that the constraint data are in *standard format.* That is, each constraint is so reformulated that the inequality and right-hand-side value have become ≥ 0. The cited constraint values are the left-hand-side values of the reformulated inequalities. This approach allows direct analysis of feasibility or infeasibility by inspection, without use of the input file definition of constraints.

The line `# records: skip 6 entries` should be familiar from the initsol file. It says that the solution values start after the 6th entry of the subsequent data record.

The line `# feasible solution` says the solution is feasible.

Finally, the line `4 13 16.5 7 12.5 ...` is the solution record.

The file <problem>.final.multi has all final solutions. The format is as for the file <problem>.params.multi. If just one function is minimized, then there is just one data record. But if several functions are minimized, then there are a number of records that constitute an approximation of the Pareto front.

At the end of the record, the summary statistics of the run producing the file <problem>.final.multi are included as comments. Except for the leading `#` of each record, the data are identical to the screen output of multicc at the end of the run.

params File

The params file is created by multicc from the input file or, if the input file does not exist, from problem data in subroutine test_problem() in file problemdef.lb.c.

The first portion of the params file has the same information as the input file up to an including the record `# endformat`. That information optionally is followed by solutions in the format of the initsol file.

CAUTION: In principle, the user can also manipulate the params file. Generally, such action can lead to complex execution errors and therefore should not be done. There is one exception: When the user invokes the cutout option discussed later, the user must define a certain params file.

Auxiliary Files

Multicc uses a number of additional files that in principle can be ignored by the user. This section covers the most important ones.

inbox and outbox Files

The inbox file transmits solution vectors to a black box program. The outbox file contains the output of the black box program.

nomad.params File

Program nomad is based on program NOMAD available from `https://www.gerad.ca/nomad`. The NOMAD manual can be found in the file user_guide.pdf of Leibniz/nomad.<version>/doc, where version refers to the version of NOMAD used by multicc.

The file nomad.params defines parameters used by nomad. The file is generated by subroutine defineNomadParams() in file params.nomad.c. The user may change nomad.params by modifying code in subroutine defineNomadParams(). Such changes are not recommended since they may cause subtle errors in the interaction of nomad with the other programs of multicc.

runprocess File

For parallel processing, multicc produces for each processor a runprocess file. For the nth process, the file is runprocess.<n>.sh. The file is created by subroutine writeProcessor() in file readParams.lb.c. The runprocess file specifies a max time for execution of the black box program, with default value 10 min. The limit prevents black box programs from ever becoming orphans that run indefinitely beyond control of multicc.

nomad Transfer Files

Data transfer to and from nomad is handled by executable file transfer.nomad.exe and data file transfer.nomad.params. These files should never be modified.

Multicc Options: Files and Directories

Multicc does not use a parameter file. Instead, all options are specified on the command line. For display of all options on the screen, execute

```
Leibniz/Multicc/Code/multicc
```

A summary of the options is included in manualMulticc.txt of Leibniz/Multicc/Doc. Details are covered below.

The first option of the multicc call is always the name of the problem to be solved. For example,

```
Leibniz/Multicc/Code/multicc zdt6
```

says that problem zdt6 is to be solved. Subsequently, the options can be listed in any order. If in the sequence of options a case is repeated, then the later choice is enforced. For example,

```
Leibniz/Multicc/Code/multicc zdt6 \
     -inbox fileX -inbox fileY
```

means that the filename given under option -inbox is first processed to be fileX, but then updated to fileY.

CAUTION: Errors made in the specification of options typically are caught by the parser and flagged with appropriate error messages. But there is a subtle error that cannot be analyzed by the parser and that may cause a baffling error message. That case occurs when a variable number of arguments follow an option statement. If a subsequent option is listed without the required hyphen prefix, the option string is interpreted as another argument for the preceding option, causing a confused interpretation. Hence, whenever there is a confusing error message, the user should first verify that all options are listed with hyphen prefix.

Below, the options are discussed in detail.

-input

`-input <input file>`

specifies the name of the input file.

If the problem is defined in subroutine test_problem() of file problemdef.lb.c, then absence of the input file is declared by

`-input NOINPUTFILE`

Default: `-input <problem>.input`

-initsol

`-initsol <initsol file>`

specifies the name of the initial solution file.

Default: `-initsol NOINITSOLFILE`

which means that there is no initial solution file.

-output

`-output <output file>`

specifies the name of the output file. This file is no longer used. Instead, the multicc output is supplied in files <problem>.params.multi and <problem>.final.multi. Accordingly, the specified file is entirely controlled by the user.

Default: `-output <problem>.output`

-blackbox

`-blackbox <blackbox file>`

specifies the name of the blackbox file.

Default: `-blackbox <problem>.blackbox`

-inbox

`-inbox <inbox file>`

specifies the name of the inbox file used by the black box.

Default: `-inbox <problem>.inbox`

-outbox

`-outbox <outbox file>`

specifies the name of the outbox file used by the black box.

Default: `-outbox <problem>.outbox`

-params

`-params <params file>`

specifies the name of the params file. If no params file is provided by the user, specify

`-params NOPARAMSFILE`

CAUTION: In principle, the user can create and manipulate params files. Such action can lead to complex execution errors and therefore should be avoided. There is one exception when the cutout option is used.

Default: `-params NOPARAMSFILE`

which means that the user has not specified a params file. Program multicc then will create and use a params file of the form <problem>.params.

-processor

`-processor <processor file>`

specifies the name of the processor file.

Default: `-params NOPROCFILE`

which means that there is no processor file and that all function evaluations are done by the processor running multicc.

-leibnizpath

`-leibnizpath <path to Leibniz/Multicc/Code>`

specifies the path to the Leibniz/Multicc/Code directory. The path should terminate with `/`.

Default: `-leibnizpath ~/Leibniz/Multicc/Code/`

-nomadpath

`-nomadpath <path to Leibniz/Nomad/NOMAD.<version>/bin/>`

specifies the path to the Leibniz/Nomad/NOMAD/bin directory, where NOMAD.<version> is the currently installed NOMAD version. The path should terminate with `/`.

Default: `-nomadpath ~/Leibniz/Nomad/NOMAD.<version>/bin/`

-tempdir

`-tempdir <path to temp directory>/<problem>`

specifies the path to the temporary directory. Due to frequent use of that directory, it shouldn't be a directory accessed across a network. Instead, it should be a local temp directory, or even better, a virtual directory. The path should terminate with `/`.

Default: `-tempdir /dev/shm/<problem>/`

which specifies a virtual directory.

Multicc Options: Computations

There are a number of options controlling computations of the various programs. For a start, the user may want to ignore these options except for sequenceminone, objrange, and denconcoordinate.

-singleminout

`-singleminout <flag>`

specifies whether function evaluations are collected during execution of the single-function minimization programs in the file <problem>.singleminout in the temporary directory specified by the tempdir option. If flag = 0, the evaluations are not collected. If flag = 1, they are collected.

Execution rules:

1. If multicc is run, then multicc resets the file <problem>.singleminout in the temporary directory to empty, and each single objective minimization program adds to the file. Headers such as `# multicc`, `# nomad`, `# dencon` etc. are used in the file to differentiate among the sections produced by the various programs. When subsequently nsga is run, the <problem>.singleminout file is used to guide initialization of the first population.
   Besides the headers `# multicc`, `# nomad`, `# dencon` etc., the file contains separators of the form `##cycle` that define clusters of data records. In the case of single processor runs, each cluster consists of just one record; it contains the evaluation results for one solution. In the case of parallel evaluations on n processors, each cluster contains up to n data records; they contain the results for the solutions processed in parallel in one cycle.
2. If a single objective minimization program is run by itself, then the following actions are taken: If the file <problem>.singleminout exists in the temporary directory, the output is appended to the

file. Otherwise the file <problem>.singleminout is created, and the output is written into the new file.

3. When nsga is executed by itself with the option singleminout, then nsga looks for the file <problem>.singleminout in the temporary directory. If the file does not exist, the program stops with error termination. Otherwise the file is used to guide initialization of the first population.

Default: `-singleminout 0`

-accuracy

```
-accuracy <value>
```

specifies the tolerance for deciding convergence of nsga, where value must lie between 0 and 1. The smaller value, the more demanding the convergence condition. Use 0.01 for low accuracy, and 0.001 for good accuracy.

Default: `-accuracy 0.001`

-sequenceminone

```
-sequenceminone nsga <prog1>
```

or

```
-sequenceminone nsga <prog1> <prog2>
```

defines the program sequence applied to the problem. Here, prog1 and prog2 are single function minimization programs taken from the list dencon, denpar, and nomad.

The effect of the choice depends on whether single objective or multi-objective minimization is carried out.

In the case of single objective minimization, the selected prog1 and prog2 are used to solve the problem. Details follow shortly.

In the case of multiobjective minimization, the following steps are carried out. First, certain single objective minimization problems are solved with prog1 and prog2 as described in the paper "Simple Seeding of Evolutionary Algorithms for Hard Multiobjective Minimization Problems," available at
`http://www.utdallas.edu/~klaus/Wpapers/seed.pdf`.
The results are used to define a starting population for program nsga, which then solves the multiobjective problem.

In both cases, the single objective minimization problems are solved as follows. If

```
-sequenceminone nsga <prog1>
```

is specified, then prog1 solves those problems. If

```
-sequenceminone nsga <prog1> <prog2>
```

is specified, then first prog1 is applied, then prog2, then prog1 again, then prog2 again, and so on, until one of two termination conditions is satisfied. Under the first condition, a bound on the number of such iterations is reached. The bound is defined by option maxiter described later. The second condition is based on convergence of objective function values.

In principle, even longer sequences are allowed, for example,

```
-sequenceminone nsga <prog1> <prog> <prog3>
```

where prog1, prog2, and prog3 are selected from dencon, denpar, and nomad. But that selection likely is a poor choice. Indeed, the following cases have proved to be effective, and from now on it is assumed that one of these cases is selected.

```
-sequenceminone nsga <prog1>
```

with prog1 selected as dencon, denpar, or nomad, when the single objective minimization problems are expected to be particularly well handled by the selected program.

```
-sequenceminone nsga <prog1> <prog2>
```

with prog1 = nomad and prog2 = dencon or denpar, when the single objective minimization problems are really difficult. Typically, prog2 = dencon will be a better choice than prog2 = denpar. But when the number of available processors is much larger than twice the number of variables, denpar may be the better choice, as well as for peculiar problems identified by trial runs.

It may happen that prog1 does not find a feasible solution. In that case, multicc attempts to find a feasible solution using nsga. If that attempt succeeds, prog1 is started with that feasible solution. If the attempt fails, the screen output at termination indicates which constraints likely cause the infeasibility, and multicc gives up on solving the minimization problem.

For a demonstration, change in the input file optim.input for optim the right-hand-side value of constraint3 from 10 to -4. This change makes the problem infeasible. Then execute

```
multicc optim -objrange 1 1 1
```

as before. The program terminates at the end of iteration 1 of phase 1. The final portion of the screen output contains the following lines.

```
 ***Caution***: obj values in output files for the
                following objective function(s) only:
                 function 1
 warning: no feasible solution found
          best infeasible solution in final_pop.out

 caution: problem is infeasible
          file optim.params.multi has information pointing to
          the constraint(s) causing the problem
```

We follow the advice and find at the end of file optim.params.multi the following information.

```
# infeasible case: constraints with internal
  min value < 0 likely cause the infeasibility
# infeasible case: constr# = 3   min = -1   max = -1
```

Hence, the third constraint, which is constr3, is claimed to be causing the problem. Indeed, we had modified the right-hand-side value of that constraint.

The output of optim.params.multi not only points to constraints causing the infeasibility, but also tells how we can change the right-hand-side values of these constraints to assure feasibility. Indeed, for each violated constraint with $\geq$ (resp. $\leq$) inequality, we just add to (resp. subtract from) the right-hand-side the stated min value and are certain that we will obtain a feasible solution.

There is a particular case of sequenceminone option when the single objective minimization problems are very easy. The option then specifies dencon for single objective minimization, and a second option called denconcoordinate reduces dencon to a very simple program. Here are the two options listed together.

```
-sequenceminone nsga dencon -denconcoordinate 1
```

Default: `-sequenceminone nsga nomad dencon`

-stopcriterion

```
-stopcriterion <goal>
```

specifies the goal of the problem. If goal = feasible, just a feasible solution is to be found. If goal = optimal, an optimal solution is to be determined.

CAUTION: Use `-stopcriterion optimal` only with direct execution of dencon, denpar, or nomad where multicc is not used as overall control program. Do *not* use `-stopcriterion optimal` with multicc since multicc always starts with `-stopcriterion feasible` and then modifies that option for execution of the various programs.

Default: `-stopcriterion feasible`

-minall

`-minall <method>`

specifies the method carrying out multiobjective minimization. Possible cases are method = nsga and method = gencc.

<u>CAUTION</u>: This option is obsolete. The default method = nsga should not be changed since the use of gencc is no longer recommended due to superior performance of combination sequences involving nsga and the single objective minimization programs.

Default: `-minall nsga`

-maxeval

`-maxeval <limit>`

defines the max number of evaluations done in the multiobjective minimization phase.

Default: `-maxeval 50000`

-maxiter

`-maxiter <limit>`

This option applies only if `-sequenceminone nsga <prog1> <prog2>` is used explicitly or by default. In that case, limit specifies the max number of times single objective minimization is done by prog1 or prog2.

For example, if limit = 2, then prog1 and prog2 are executed, and if limit = 4, prog1, prog2, prog1, and prog2 are executed.

It may happen that multicc stops due to convergence conditions before the iteration limit is reached.

CAUTION: If limit ≤ 1 is specified, multicc replaces that value by the smallest reasonable value, which is 2.

Default: `-maxiter 2`

-maxitereval

```
-maxitereval <limit>
```

specifies the max number of evaluations done during each execution of prog1 or prog2. For example, if limit = 1000, then each execution of prog1 or prog2 carries out at most 1,000 evaluations.

Default: `-maxitereval 1000`

Advanced Multicc Options

This section covers advanced options for controlling multicc computations.

-maxfirstitereval

`-maxfirstitereval <limit>`.

If `limit` $= 0$, then the option has no effect. If `limit` > 0, then it is assumed that a particular specification of option sequenceminone is used. That is, an additional program prog0 is specified by the declaration

```
-sequenceminone nsga <prog0> <prog1>
```

or

```
-sequenceminone nsga <prog0> <prog1> <prog2>
```

Then prog0 is run with max number of evaluations equal to the limit of the maxfirstitereval option, and the solution found by prog0 is used to start prog1. From then on, the process defined under option sequenceminone is carried out.

The program prog0 can be any program, so it could be dencon, denpar, nomad, or even nsga.

The seemingly odd process is useful when various program combinations are to be compared using a common starting solution. The latter solution is computed by prog0.

CAUTION: The execution of prog0 is done with single processor even if multiprocessor execution has been specified. That way, the same solution is produced regardless of the number of processors used for the other programs. This allows comparison of results where the number of processors is varied, since each case starts with the same solution computed by prog0.

If the user wants to compute a starting solution with parallel processors, then this should be done separately by

```
-sequenceminone nsga <prog0> -maxitereval <limit> \
-minsingleonly
```

The resulting solution is then transmitted to subsequent runs of multicc using the initsol file.

Default: `-maxfirstitereval 0`

-minsingleonly

The option applies only if the minimization problem has at least two objective functions.

```
-minsingleonly
```

specifies that only the starting solutions for nsga are computed by prog1 and possibly prog2 as described under option sequenceminone, and that multiobjective minimization by nsga is skipped.

-minallonly

The option applies only if the minimization problem has at least two objective functions.

```
-minallonly
```

specifies that the computation of the starting solutions by prog1 and possibly prog2 as described under option sequenceminone is skipped and nsga is applied immediately to solve the multiobjective minimization problem.

-ignoreconstraint

`-ignoreconstraint`

specifies that all constraints of the problem are to be ignored and only the lower and upper limits on variables are to be enforced.

-gradualconstraint

The option applies only if the minimization problem has at least two objective functions.

`-gradualconstraint <npop>`

specifies that constraints are to be gradually enforced by nsga in such a way that they are fully enforced, and continue to be fully enforced, when npop populations have been processed by nsga. If npop = 0, the constraints are fully enforced from the very beginning.

CAUTION: This option is obsolete and should not be used.

Default: `-gradualconstraint 0`

-scaleconstraint

The option applies only if the minimization problem has at least two objective functions.

`-scaleconstraint`

alters the way nsga computes constraint violations.

In the default process, constraint violations are simply added up by nsga to obtained an overall constraint violation value. When the option scaleconstraint is invoked, the constraint violations are scaled according to the maximum violation before being added up.

CAUTION: This option is obsolete and should not be used.

-objfactor

The option applies only if the minimization problem has at least two objective functions.

`-objfactor <flag1> <flag2> ... <flagnobj>`

specifies which of the nobj objective functions are ignored versus active. If flagi = 0, then objective function i is ignored. If flagi = 1, objective function i is active.

Default: `-objfactor 1 1 ... 1`

-objtolerance

`-objtolerance <alpha1> <alpha2> ... <alphanobj>`

specifies nonnegative tolerances alpha1, alpha2, ..., alphanobj for the nobj objective functions of the problem. That is, if two values for objective function i differ by at most alphai, then they are not considered significantly different.

The tolerance values override the default value for objective functions given by FUNCTION_PRECISION in file sample.lb.h as well as any tolerances specified for the objective functions in the input file.

Default: alphai = FUNCTION_PRECISION of file sample.lb.h, overriden by tolerances specified for the objective functions in the input file.

-constrtolerance

`-constrtolerance <beta1> <beta2> ... <betancon>`

specifies nonnegative tolerances beta1, beta2, ..., betancon for the ncon constraints of the problem. That is, if a violation of constraint i is at most betai, then the constraint is considered satisfied.

The tolerance values override the default value for constraints given by CONSTRAINT_PRECISION in file sample.lb.h as well as any tolerances specified for the constraints in the input file.

Default: betai = CONSTRAINT_PRECISION of file sample.lb.h, overridden by tolerances specified for the constraints in the input file.

-objgoal

```
objgoal <goal1> <goal2> ...  <goalnobj>
```

specifies goal1, goal2, ..., goalnobj as goals for the objective functions, in the following sense. If function i is minimized, then minimization is carried out as long as the value of function i is above goali. If function i is maximized, the maximization is carried out as long as the value of function i is below goali.

Default: for minimization, goali $= -\infty$, and for maximization, goali $= \infty$.

-objbound

The option applies only if the minimization problem has at least two objective functions.

```
-objbound <bound1> <bound2> ...  <boundnobj>
```

specifies bound1, bound2, ... boundnobj as bounds for the objective functions, in the following sense. In case of minimization (resp. maximization) of function i, boundi is the highest (resp. smallest) function value of interest.

Default: for minimization, boundi $= \infty$, and for maximization, boundi $= -\infty$.

-objrange

The option applies only if the minimization problem has at least two objective functions.

```
objrange <range1> <range2> ...  <rangenobj>
```

specifies range1, range2, ... rangenobj as range values for the objective functions, in the following sense. For objective i, rangei is an estimate of the range of values that function i may take on in the Pareto front.

For example, if the values of function 1 on the Pareto front vary from -10 to 75 and for function 2 from 70 to 200, then the range for function 1 is $75 - (-10) = 85$ and for function 2 it is $300 - 70 = 230$.

The estimates for the objectives may be simultaneously scaled by a positive factor without any impact on efficiency of computations. Thus, one only needs to estimate relative size of the ranges and not their absolute values. In addition, the estimates need not be precise and may be off by a factor 10 or even 100 without detrimental effect. For the example case of 85 and 230, one may simply use 1 as range value for both objectives since they differ only by a factor of 2.7.

There are no default range values. If the option is not specified, all multicc computations relying range values are skipped. For difficult problems, this may severely impact efficiency of the computations.

Hence, if no prior range estimates exist, it pays to explore the behavior of the functions and obtain a rough estimate of the range values before starting the multiobjective minimization process.

-cutout

The option applies only if the minimization problem has at least two objective functions.

```
-cutout <flag>
```

where flag = 0 or 1. If flag = 0, the option has no effect. Below, assume flag = 1, so the option is active.

The option is used to focus the computations of nsga on a portion of a partially computed Pareto front. Put differently, the option acts like a magnifying glass applied to a portion of the Pareto Front.

The portion of the Pareto front of interested is specified by the objbound option. That is, the bound value for objective j is the maximum value that is allowed for the reduced Pareto front.

CAUTION: For efficient execution of the cutout option, the bound values of the objbound option, say b1, b2, ... should be so chosen that, for each bound bj, the partial Pareto front on hand satisfies the following condition: There is at least one point, say (p1, p2, ...) of the Pareto front such that (1) the point is feasible for the bounds, that is, for all i, pi $\leq$ bi, and (2) the difference bj $-$ pj is small. Here it is assumed that all objectives are to be minimized. Later we simply say that the point satisfies all bounds bi and is close to bj.

Example: Suppose there are two objective functions. The values of the first function in the Pareto front range from 0 to 50, and for the second function from 0 to 90. Suppose we are interested in the portion of the front where simultaneously function 1 does not exceed 30 and function 2 does not exceed 10.

Assume that the selected bounds satisfy the above-mentioned condition involving the Pareto front on hand. For effective magnification of the Pareto front in that region, several steps are carried out, where the files <problem>.params.multi and <problem>.final.multi are assumed to be at hand from earlier computation of the Pareto front. In particular, all solutions in the <problem>.final.multi file have correct objective and constraint values and constitute a possibly incomplete representation of the Pareto front. Next carry out the following steps.

Copy <problem>.params.multi into new file <problem>.cutout.params.

Copy <problem>.final.multi into new file <problem>.cutout.initsol.

CAUTION: The names of the new files should be so chosen so that they do not overlay existing files. Here, this is accomplished by the additional "cutout" term in the file names.

Execute multicc with all previously used options, except that the objrange option may have to be adjusted due to the reduced Pareto region, and with the following additional options.

```
-params <problem>.cutout.params \
-initsol <problem>.cutout.initsol -objbound 30 10 -cutout 1
```

The following solution process is carried out by multicc:

The program discards all <problem>.cutout.params solutions that exceed any objbound value, then discards all <problem>.cutout.initsol solutions that exceed any objbound value or are infeasible. From the remaining <problem>.cutout.initsol solutions, multicc retains the following cases:

For each objective i: one solution that minimizes objective i, and one solution that maximizes objective i.

Since a given solution may satisfy several of these conditions, the total number of retained solutions may be less than twice the number of objective functions. Program multicc is started with these solutions.

CAUTION: Typical input errors are (1) forgetting one of the required options needed with the cutout option; and (2) specifying the three required options params, initsol, and objbound, but then forgetting to add the cutout option.

Here are the calls for an example involving problem zdt6. It is assumed that we have solved the original problem with

```
Leibniz/Multicc/Code/multicc zdt6 \
     -input NOINPUTFILE -objrange 1 1
```

For both objective functions, the values shown in the output plot of the Pareto front range from 0.0 to 1.0. We now decide to focus on the small subregion of the Pareto front where both objective functions have values ≤ 0.7. Note that these bounds satisfy the earlier mentioned condition. That is, for each bounds bj, there is a point of the Pareto front that satisfies all bounds bi and is close to bj. Now carry out the following steps.

1. Copy file zdt6.params.multi into new file zdt6.cutout.params and file zdt6.final.multi into new file zdt6.cutout.initsol.
   ```
   cp zdt6.params.multi zdt6.cutout.params
   cp zdt6.final.multi zdt6.cutout.initsol
   ```
2. Solve zdt6 with params, initsol, objbound, and cutout options. There is no need to change the values of the objrange option since the range of the objective functions for the reduced region is roughly the same for both objective functions. Here is the complete command.
   ```
   multicc zdt6 \
        -input NOINPUTFILE -objrange 1 1 \
        -params zdt6.cutout.params \
        -initsol zdt6.cutout.initsol \
        -objbound 0.7 0.7 -cutout 1
   ```

The output plot of the Pareto front covers the region where both objective functions do not exceed 0.7, as desired.

-dynamiclimit

The option applies only if the minimization problem has at least two objective functions.

```
-dynamiclimit <flag>
```

with flag = 0 or 1. The option controls convergence criteria, where flag = 0 means default action and flag = 1 activation of a dynamic limit convergence process.

CAUTION: This option is obsolete and should not be used.

Default: `-dynamiclimit 0`

-convergetesteval

The option applies only if the minimization problem has at least two objective functions.

```
convergetesteval <neval>
```

If neval $>$ 0, it specifies the frequency with which convergence is checked in nsga. Specifically, the next convergence test is performed when at least neval evaluations have been done. If neval = 0, the test frequency is decided dynamically by multicc. The value depends on total evaluation effort during the preceding minimization of single objective functions.

If the convergence test is to be avoided entirely, set neval $\geq$ max number of evaluations specified by maxeval option. In that case, nsga uses the max number of evaluations.

Default: `convergetesteval 0`

-paretosize

The option applies only if the minimization problem has at least two objective functions.

```
paretosize <size>
```

If size > 0, it specifies the minimum size of the Pareto front computed by nsga. If size = 0, multicc selects the minimum size of the Pareto front using several considerations, including the number of solutions computed during the preceding minimization of single objective functions.

Default: `paretosize 0`

-plotdetail

The option applies only if the minimization problem has two or three objective functions.

```
-plotdetail <flag>
```

where flag = 0, 1, or 2. If flag = 0, do not plot the Pareto front. If flag = 1, plot only final Pareto front. If flag = 2, plot all intermediate results and the final Pareto front.

Default: `-plotdetail 2`

-printdetail

```
-printdetail <flag>
```

where flag = 0, 1, or 2. If flag = 0, do not produce screen output. If flag = 1, produce screen output of summary for each step. If flag = 2, produce detailed screen output of all steps.

Here is the screen output with flag = 1 when zdt6 is solved using

```
multicc zdt6 -input NOINPUTFILE -objrange 1 1 \
-printdetail 1
```

```
          Leibniz System: Multicc Module
                Version: 16.0
                Compiled: Feb 11 2015
             Copyright 2015 by Leibniz
                Plano, Texas, U.S.A.

**** Phase 1 *** Iteration 1 *** Program nomad ****
**** Phase 1 *** Iteration 2 *** Program dencon ****
**** Phase 2 *** Iteration 1 *** Program nomad ****
**** Phase 2 *** Iteration 2 *** Program dencon ****
**** Phase 3 ****************** Program nsga ****

 multicc evaluation counts
                vectors cycles
       phase 1 =   600
       phase 2 =  1198
       phase 3 =  4275
   ___________________
   total count =  6073
 output in zdt6.params.multi and zdt6.final.multi
```

Default: `-plotdetail 2`

-slowmotion

The option applies only if the minimization problem has at least two objective functions.

`-slowmotion <flag>`

where flag = 0 or 1. If flag = 0, processing by nsga is done without any slowdown. If flag = 1, processing by nsga is done in slow motion so the plot showing development of the Pareto front can be studied as it evolves.

-seednsga

`-seednsga <value>`

specifies the rational seed for the nsga random number generator. Use $0.0 < \text{value} < 1.0$.

Default: `-seednsga <SEEDNSGA>`

where SEEDNSGA is defined in file sample.lb.h.

-seednomad

`-seednomad <value>`

specifies the integer seed for the nomad random number generator. Use $\text{value} > 0$.

Default: `-seednomad <SEEDNOMAD>`

where SEEDNOMAD is defined in file sample.lb.h.

-initialmesh

`-initialmesh <value>`

specifies the initial mesh value for nomad.

CAUTION: For a given phase, the value applies only to the first iteration of nomad. For subsequent iterations of nomad during the same phase, multicc reduces the value.

Default: `-initialmesh 0.5`

-finalmesh

`-finalmesh <value>`

specifies the final mesh value for nomad.

Default: `-finalmesh 0.005`

-maxproctime

`-maxproctime <value>`

specifies the max time limit in minutes for ssh execution during parallel processing. If the time limit is exceeded, execution of the shell is terminated regardless of circumstance. The limit prevents black box programs from ever becoming orphans that run on indefinitely beyond control of multicc.

Default: `-maxproctime 10`

-denconcoordinate

`-denconcoordinate <flag>`

controls a certain choice in program dencon by flag = 0 or 1.

If flag = 0, dencon uses coordinate, diagonal, and dense directions.

If flag = 1, dencon uses coordinate directions only and *not* diagonal or dense directions. On single processor runs, dencon then performs essentially the same as seqpen, thus justifying elimination of seqpen. This choice is appropriate when the single function minimization problems are particularly easy to solve.

Default: `-denconcoordinate 0`

Leibniz System

Development Tool for Logic-based Intelligent Systems

References

Almubaid, H., and Truemper, K., ”Learning to Find Context-based Spelling Errors,” in “Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques” (E. Triantaphyllou and G. Felici, eds.), Springer-Verlag, Berlin, 2006, pp. 597–627.

Atkeson, C. G., Moore, A. W., and Schaal, S., “Locally weighted learning,” *Artificial Intelligence Review* 11 (1997) 11–73.

Bartnikowski, S., Granberry, M., Mugan, J., and Truemper, K., “Transformation of Rational and Set Data to Logic Data,” in *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques* (E. Triantaphyllou and G. Felici, eds.), Kluwer Academic Publishers, New York, 2006, pp. 253–278.

Di Giacomo, G., Felici, G., Maceratini, R., and Truemper, K., “Application of a New Logic Domain Method for the Diagnosis of Hepatocellular Carcinoma,” *Proceedings of 10th Triennial Congress of the International Medical Informatics Association* (MEDINFO2001), 434–438.

Felici, G., Rinaldi, G., Sforza, A., Truemper, K., “Traffic Control: A Logic Programming Approach and a Real Application,” *Ricerca Operativa* 30 (2001) 39–60.

Felici, G., Rinaldi, G., Sforza, A., Truemper, K., “A logic programming based approach for on-line traffic control,” *Transportation Research Part C* 14 (2006) 175–189.

Felici, G., Sun, F., and Truemper, K., "Learning Logic Formulas and Related Error Distributions," in *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques* (E. Triantaphyllou and G. Felici, eds.), Springer-Verlag, Berlin, 2006, pp. 193–226;

Felici, G., Sun, F., and Truemper, K., "A Method for Controlling Errors in Two-class Classification," *Proceedings of 23rd Annual International Computer Software and Applications Conference* (COMPSAC '99), 186–191.

Janiga, G., and Truemper, K., "Dimension Reduction of Chemical Process Simulation Data," available at
`http://www.utdallas.edu/~klaus/Wpapers/reduce.pdf`.

Liuzzi, G., and Truemper, K., "Parallelized hybrid optimization methods for nonsmooth problems using NOMAD and linesearch." Available at
`http://www.utdallas.edu/~klaus/Wpapers/hybrid.pdf`.

Moreland, K., and Truemper, K., "The Needles-in-Haystack Problem," *Proceedings of International Conference on Machine Learning and Data Mining* (MLDM 2009) (2009) 516–524.

Remshagen, A., and Truemper, K., "A Solver for Quantified Formula Problem Q-ALL SAT," available at
`http://www.utdallas.edu/~klaus/Wpapers/qallsatcc.pdf`.

Remshagen, A., and Truemper, K., "An Effective Algorithm for the Futile Questioning Problem," *Journal of Automated Reasoning* 34 (2005) 31-47.

Riehl, K., and Truemper, K.,"Construction of Deterministic, Consistent, and Stable Explanations from Numerical Data and Prior Domain Knowledge," available at
`http://www.utdallas.edu/~klaus/Wpapers/explanation.pdf`.

Straach, J., and Truemper, K., "Learning to Ask Relevant Questions," *Artificial Intelligence* 111 (1999) 301–327.

Truemper, K., *Design of Logic-based Intelligent Systems*, Wiley, 2004.

Truemper, K., "Improved Comprehensibility and Reliability of Explanations via Restricted Halfspace Discretization," *Proceedings of International Conference on Machine Learning and Data Mining* (MLDM 2009) (2009) 1–15.

Truemper, K., *Effective Logic Computation – Revised Edition*, Leibniz Company, 2010. Available at `http://www.utdallas.edu/~klaus/lbook.html`.

Truemper, K., "Simple Seeding of Evolutionary Algorithms for Hard Multiobjective Minimization Problems." Available at `http://www.utdallas.edu/~klaus/Wpapers/seed.pdf`.

Vick, J., Campbell, T., Shriberg, L., Green, J., Truemper, K., Rusiewicz, H., and Moore, C., "Data-driven subclassification of speech sound disorders in preschool children," *Journal of Speech, Language, and Hearing Research*, 2015.

Walsdorf, A., Bader, K., Stein, E., and Kopp F. O., *Das letzte Original: Die Leibniz-Rechenmaschine der Gottfried Wilhelm Leibniz Bibliothek*, Gottfried Wilhelm Leibniz Bibliothek, 2014.

Zhao, Y., and Truemper, K., "Effective Spell Checking by Learning User Behavior," *Applied Artificial Intelligence* 13 (1999) 725–742.

Leibniz System

Development Tool for Logic-based Intelligent Systems

Subject Index

If a keyword can be written in upper case letters as well as lower case letters, then the index contains only the lower-case version.

Version 16.0 1 July 2015