libexact User's Guide

Version 1.0

Petteri Kaski, Olli Pottonen
HIIT Technical Reports 2008-1

Petteri Kaski, Olli Pottonen

libexact User's Guide Version 1.0

HIIT Technical Reports 2008–1

HELSINKI
INSTITUTE FOR
INFORMATION
TECHNOLOGY

Petteri Kaski
Helsinki Institute for Information Technology HIIT
Department of Computer Science, University of Helsinki
P.O. Box 68, FI-00014 University of Helsinki, Finland
petteri.kaski@cs.helsinki.fi

Olli Pottonen
Department of Communications and Networking
Helsinki University of Technology TKK
P.O. Box 3000, FI-02015 TKK, Finland
olli.pottonen@tkk.fi

Helsinki Institute for Information Technology HIIT
HIIT Technical Reports 2008–1
ISBN 978-951-22-9488-6 (printed)
ISBN 978-951-22-9489-3 (electronic)
ISSN 1458-9478 (electronic)

Printed at Yliopistopaino, Helsinki

Helsinki Institute for Information Technology HIIT
Tietotekniikan tutkimuslaitos HIIT

Kumpula site
Mailing address:
HIIT
P.O. Box 68
FI-00014 University of Helsinki
Finland
Visiting address:
University of Helsinki
Department of Computer Science
Gustaf Hällströmin katu 2b
00560 Helsinki
tel: +358 9 1911
fax: +358 9 191 51120

Otaniemi site
Mailing address:
HIIT
P.O. Box 5400
FI-02015 TKK
Finland
Visiting address:
Helsinki University of Technology
Department of Information and Computer Science
Konemiehentie 2
02150 Espoo
tel: +358 9 4511
fax: +358 9 451 3277

Spektri site
Mailing address:
HIIT
P.O. Box 9800
FI-02015 TKK
Finland
Visiting address:
Spektri Business Park
Pilotti Building
Metsänneidonkuja 4
02130 Espoo
tel: +358 9 4511
fax: +358 9 694 9768

www.hiit.fi

1 Introduction

This user's guide documents libexact, a software library for solving combinatorial exact covering problems. Such a problem instance can be formulated as a system of m integer linear equations over n variables,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \tag{1}$$

with variable bounds

$$0 \le x_1 \le u_1, \quad 0 \le x_2 \le u_2, \quad \ldots, \quad 0 \le x_n \le u_n. \tag{2}$$

It is furthermore required that $a_{ij} \in \{0, 1\}$, $b_i \in \{1, 2, \ldots\}$, and $u_j \in \{1, 2, \ldots\}$ for all $i \in \{1, 2, \ldots, m\}$ and $j \in \{1, 2, \ldots, n\}$. Given a problem instance, the task is to list all of its integer solutions, $x = [x_1, x_2, \ldots, x_n]$. To avoid listing an abundance of solutions in degenerate cases, only solutions with $x_j = 0$ whenever $\sum_i a_{ij} = 0$ are to be listed.

Example. The system

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

with variable bounds

$$0 \le x_1 \le 1, \quad 0 \le x_2 \le 1, \quad 0 \le x_3 \le 1, \quad 0 \le x_4 \le 1, \quad 0 \le x_5 \le 1$$

has exactly two integer solutions, namely $x_1 = x_3 = x_5 = 0$, $x_2 = x_4 = 1$ and $x_1 = x_2 = 0$, $x_3 = x_4 = x_5 = 1$.

A combinatorial interpretation of a problem instance is as follows. The system (1) in effect requires that each row i is covered exactly b_i times using the columns of the matrix $[a_{ij}]$, where a column j covers a row i if and only if $a_{ij} = 1$. The bounds (2) require that each column j is used at most u_j times in the covering. Each component x_j of an integer solution indicates how many times a column is to be used in a covering.

The libexact library is implemented in the C programming language. The solution algorithm used by the library is a backtrack search with a branching rule that always covers a row having the minimum number of candidate columns available for covering. A detailed description of this technique and its fast implementation appears in "D.E. Knuth, Dancing Links, *Millennial*

Perspectives in Computer Science (J. Davies, B. Roscoe, and J. Woodcock, Eds.), Palgrave, Basingstoke, England, 2000, pp. 187–214."

The library is arguably best suited for combinatorial listing applications in which (a) the system (1) and the values b_i and u_j are small, preferably $b_i = u_j = 1$; and (b) the practical challenge is more in listing all the solutions rather than in deciding whether a solution exists.

If you use the library in your work, scientific or otherwise, the authors are happy to hear about this. Also, any suggestions for improvement are greatly appreciated. If you want to acknowledge the use of libexact in your work, please do so by citing the technical report "P. Kaski, O. Pottonen, libexact User's Guide, Version 1.0, HIIT Technical Reports 2008–1, Helsinki Institute for Information Technology HIIT, 2008."

2 License

The libexact library is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

The libexact library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with the libexact library (see the file `LICENSE`); if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

3 Getting started

3.1 Obtaining the latest version

The latest version of libexact can be obtained from the web at

⟨`http://www.cs.helsinki.fi/u/pkaski/libexact/`⟩.

This user's guide documents version 1.0 of libexact.

3.2 Compiling

An honest attempt has been made to make the library source code conform to the ISO/IEC 9899:1999 standard (C99). The library and the example programs should compile on most modern UNIX (Linux) systems simply by running the command `make`. To compile manually, the main library routines reside in the file `exact.c`, which must be linked with the utility functions

in `util.c` to obtain a functional library. The public interface to the library is declared in `exact.h`.

3.3 Testing

After compiling, it is strongly recommended that the library is tested by running the test executable `test`, which, among other tests, checks that certain known combinatorial integer sequences are correctly evaluated. The sequence identifiers of the form `A??????` refer to the *Online Encyclopedia of Integer Sequences* available at

⟨`http://www.research.att.com/~njas/sequences/`⟩.

Please note that some of the tests do take some time to complete.

3.4 Using the library: A first example

To illustrate the use of the library, we will work through a few lines of C code that solve the example given in Section 1. To use the library, we first include the header file that declares the interface to the library.

```
#include "exact.h"
```

Next, we declare and allocate a data structure for the problem instance.

```
exact_t *e = exact_alloc();
```

The problem instance has four rows and five columns, both of which we choose to identify with integers starting from 1. Arbitrary integer identifiers can be used for the rows and columns.

```
exact_declare_row(e,1,1);  exact_declare_row(e,2,1);
exact_declare_row(e,3,1);  exact_declare_row(e,4,1);

exact_declare_col(e,1,1);  exact_declare_col(e,2,1);
exact_declare_col(e,3,1);  exact_declare_col(e,4,1);
exact_declare_col(e,5,1);
```

In declaring the rows, the second parameter is the row identifier i and the third parameter is the associated covering constraint b_i. In declaring the columns, the second parameter is the column identifier j and the third parameter is the associated upper bound u_j.

It remains to declare the matrix $[a_{ij}]$. We do this by declaring the positions of the 1-entries in the matrix. All the other entries are by definition 0-entries.

```
exact_declare_entry(e,1,1);  exact_declare_entry(e,1,2);
exact_declare_entry(e,1,5);  exact_declare_entry(e,2,1);
exact_declare_entry(e,2,4);  exact_declare_entry(e,3,2);
exact_declare_entry(e,3,5);  exact_declare_entry(e,4,1);
exact_declare_entry(e,4,2);  exact_declare_entry(e,4,3);
```

The example instance is now ready. To find a solution, we call the function `const int *exact_solve(exact_t *e, int *n)`. Repeated calls to this function cycle through all solutions of the problem instance; each solution found is signaled by a non-`NULL` return value. When all solutions have been listed, the return value is `NULL`, after which the next call will start the cycle again. Each solution $x = [x_j]$ is reported as follows. The `const int *` return value points to an integer array containing, in arbitrary order, each column identifier j exactly x_j times. The integer pointed by `n` is set to contain the *size* of the solution, $\sum_j x_j$.

The following fragment of code prints all solutions of our example.

```
int soln_size;
const int *soln;
while((soln = exact_solve(e, &soln_size)) != NULL) {
    for(int i = 0; i < soln_size; i++)
      printf("%d ", soln[i]);
    printf("\n");
}
```

Finally, we release the allocated problem instance.

```
exact_free(e);
```

The file `examples/example-first.c` contains the source code in this first example. When executed, the code outputs the desired two solutions $x_1 = x_3 = x_5 = 0$, $x_2 = x_4 = 1$ and $x_1 = x_2 = 0$, $x_3 = x_4 = x_5 = 1$ in the following form.

```
4 2
4 3 5
```

Observe that both the solutions and the column identifiers in each solution appear in no particular order.

3.5 Further examples

The file `examples/example-partition.c` implements a listing program for set partitions, and the file `examples/example-sudoku.c` implements a solver for sudoku puzzles. Example input for the sudoku solver can be found in the files `examples/sudoku-input*`.

4 Library interface

The header file `exact.h` declares the interface to the libexact library. Each problem instance is stored in a structure of type `exact_t` and manipulated using the functions documented in what follows. Multiple problem instances may be manipulated in parallel.

4.1 Errors

Any errors detected by the library are reported by printing an error message to `stderr` and aborting the program via `abort()`.

4.2 Memory allocation

Memory allocation is carried out automatically within the library via `malloc()` and `free()`. An error is reported if `malloc()` fails.

4.3 Modes of operation

Each problem instance is in one of three mutually exclusive internal states called *modes* that control the operations that are permitted on the instance. The modes are DECLARE, FORCE, and ITERATE.

When first initialized, a problem instance is in DECLARE mode, in which essentially all operations on the instance are permitted. When the iteration through the solutions is in progress, the instance is in ITERATE mode, in which most operations on the instance are forbidden. The FORCE mode is an in-between mode that occurs only in more advanced use when a partial solution has been forced to the solution stack. The transitions between modes and the permitted operations are documented in detail in what follows. For basic use of the library, these modes can essentially be ignored.

4.4 Initializing and releasing an instance

The following functions initialize and release problem instances.

▷ `exact_t *exact_alloc(void);`
Allocates and initializes an empty problem instance and returns a pointer to it. The instance is initially in DECLARE mode.

▷ `void exact_free(exact_t *e);`
Releases the problem instance `e`.

4.5 Declaring an instance

The following functions are used to declare a problem instance.

▷ `void exact_declare_row(exact_t *e, int i, int b);`
Declares a row with row identifier `i` to the problem instance `e`. The parameter `b` is the associated covering constraint b_i. The entries a_{ij} are set to 0 for each column j in the instance. An error is reported if (a) a row with identifier `i` already exists; (b) `b` is nonpositive; or (c) the instance is not in DECLARE mode.

▷ `void exact_declare_col(exact_t *e, int j, int u);`
Declares a column with column identifier `j` to the problem instance `e`. The parameter `u` is the associated upper bound u_j. The entries a_{ij} are set to 0 for each row i in the instance. An error is reported if (a) a column with identifier `j` already exists; (b) if `u` is nonpositive; or (c) the instance is not in DECLARE mode.

▷ `void exact_declare_entry(exact_t *e, int i, int j);`
Declares the entry a_{ij} at row `i`, column `j` in the problem instance `e` to be a 1. An error is reported if (a) the row or column does not exist; (b) the entry is already set to 1; or (c) the instance is not in DECLARE mode.

▷ `int exact_can_declare(exact_t *e);`
Returns a nonzero value if the problem instance `e` is in DECLARE mode.

4.6 Iterating through the solutions

For convenience of use, the interface to the solution algorithm is an iterator. Put otherwise, the state of the algorithm is completely maintained within the data structure, and each solution is signaled to the user by returning from the search procedure.

Each solution $x = [x_j]$ is reported by means of a *solution stack*, an integer array consisting of column identifiers, where each identifier j occurs exactly x_j times in the array, in arbitrary order. The *size* of the stack is $\sum_j x_j$. In particular, if $x_j \in \{0, 1\}$ for all columns j, then the solution stack consists of precisely the identifiers j for which $x_j = 1$. Only solutions that satisfy $x_j = 0$ whenever $\sum_i a_{ij} = 0$ are reported. In particular, whenever the instance has no rows, exactly one solution—the empty solution stack—is reported.

The following functions can be used in any mode.

▷ `const int *exact_solve(exact_t *e, int *n);`
Iterates over all solutions of the problem instance `e`. Each solution

found is signaled by a non-`NULL` return value, in which case the return value points to the solution stack; the integer pointed by `n` is set to equal the size of the stack. The solution stack is guaranteed to be valid until the next call to a library function with input `e` occurs. When all solutions have been reported (or when no solutions exist), the iteration resets and the value `NULL` is returned; the integer pointed by `n` is not accessed in this case. The next call restarts the iteration. The instance is in ITERATE mode during the iteration. When the iteration resets, the instance returns to the mode preceding the iteration.

▷ `void exact_reset_solve(exact_t *e);`
Resets the solution iterator of the problem instance `e`. If an iteration was in progress, the instance returns to the mode preceding the iteration.

4.7 Examining an instance

The following functions are used to examine the structure of a problem instance. These functions can be used in any mode.

▷ `int exact_is_row(exact_t *e, int i);`
Returns a nonzero value if the problem instance `e` has a row with identifier `i`.

▷ `int exact_is_col(exact_t *e, int j);`
Returns a nonzero value if the problem instance `e` has a column with identifier `j`.

▷ `int exact_is_entry(exact_t *e, int i, int j);`
Returns the entry a_{ij} at row `i`, column `j` in the problem instance `e`. An error is reported if the row or column does not exist.

▷ `int exact_num_rows(exact_t *e);`
Returns the number of rows in the problem instance `e`.

▷ `int exact_num_cols(exact_t *e);`
Returns the number of columns in the problem instance `e`.

▷ `int exact_get_rows(exact_t *e, int *i);`
Stores the identifiers of the rows in the problem instance `e` to the array pointed by `i`, returns the number of stored rows. If the solution stack is nonempty, only the identifiers of rows for which equality does not hold in (1) in the current state are stored.

▷ `int exact_get_cols(exact_t *e, int *j);`
Stores the identifiers of the columns in the problem instance `e` to the

array pointed by `j`, returns the number of stored columns. If the solution stack is nonempty, only the identifiers of non-conflicting columns in the current state are stored.

4.8 Forcing a partial solution

The following functions are used to push an initial partial solution into the solution stack. In many cases it is convenient to first define a template instance, and then push a partial solution to obtain the instance of interest. For example, the sudoku solver in `examples/example-sudoku.c` uses this approach.

▷ `void exact_push(exact_t *e, int j);`
Pushes the column with identifier `j` into the solution stack of the problem instance `e`. The instance is in FORCE mode after a push. An error is reported if (a) a column with identifier `j` does not exist; or (b) pushing the column would conflict with a row constraint or the variable bound; (c) the column has only 0-entries; or (d) the instance is in ITERATE mode. A complete list of non-conflicting columns can be obtained via `exact_get_cols`.

▷ `void exact_pop(exact_t *e);`
Removes the most recently pushed column identifier from the solution stack of the problem instance `e`. If the solution stack becomes empty after a pop, the instance returns to DECLARE mode. An error is reported if (a) the solution stack is empty; or (b) the instance is in ITERATE mode.

▷ `int exact_pushable(exact_t *e, int j);`
Returns a nonzero value if the column with identifier `j` can be pushed into the solution stack of the problem instance `e`. Otherwise returns the zero value. An error is reported if (a) a column with identifier `j` does not exist; or (b) the instance is in ITERATE mode.

▷ `int exact_can_push(exact_t *e);`
Returns a nonzero value if the instance is not in ITERATE mode.

▷ `int exact_num_push(exact_t *e);`
Returns the size of the pushed part of the solution stack of the problem instance `e`.

▷ `int exact_get_push(exact_t *e, int *j);`
Stores the pushed part of the solution stack of the problem instance `e` to the array pointed by `j`, returns the size of the pushed part.

4.9 Controlling the search

The following functions are used to control the algorithm that searches for the solutions.

- ▷ `void exact_level(exact_t *e, exact_level_t *l, void *p);`
  `typedef int exact_level_t(void *, int, const int *);`
  Sets the function pointed by `l` as the level function with user parameter `p` for the problem instance `e`. An error is reported if the instance is in ITERATE mode. A level function is used to prune the search tree in the search for solutions. The function is evaluated at each node of the search tree. It takes as input the user parameter `p`, the size of the current solution stack, and a pointer to the solution stack. A nonzero return value from the level function indicates that the node is to be traversed; a zero return value indicates that the node and all its children are to be pruned.

- ▷ `void exact_filter(exact_t *e, exact_filter_t *f, void *p);`
  `typedef int exact_filter_t(void *, int, const int *, int);`
  Sets the function pointed by `f` as the filter function with user parameter `p` for the problem instance `e`. An error is reported if the instance is in ITERATE mode. A filter function is used to restrict the columns considered in the search for solutions. The function is evaluated for all non-conflicting column identifiers after a new column identifier is pushed into the solution stack. The return value of the function determines whether the given column identifier should be regarded as conflicting. A nonzero return value indicates that the candidate column is non-conflicting; a zero return value indicates that the candidate column is conflicting and is to be ignored. The filter function takes as parameters the user parameter `p`, size of the current solution stack, pointer to the stack and the identifier of the candidate column.

The problem instance is in ITERATE mode when level and filter functions are invoked, with one additional restriction. Namely, an error will result if either `exact_solve` or `exact_reset_solve` is invoked for the current problem instance within a level or filter function.

5 Command-line interface

The program `solve` provides a plain command-line interface to libexact. The program is invoked with

```
solve [command] [file]
```

where both the command and the file argument are optional. There are two available commands:

`-l`	or	`--list`	Lists all the solutions (default).
`-c`	or	`--count`	Counts the number of solutions.

When no file argument is given, the input is read from the standard input stream; otherwise the given file is consulted for input. All normal output is printed to the standard output stream. Errors are signaled by printing an error message to the standard error stream and terminating with a nonzero exit status.

5.1 Input format

The input consists of a sequence of lines of the following types.

- ▷ A row is declared with a line of the form

  `r` $\langle i \rangle$ $[b_i]$

  where i is the row identifier (an integer) and b_i is the associated constraint (a positive integer). The parameter b_i may be omitted, in which case $b_i = 1$ is assumed.

- ▷ A column is declared with a line of the form

  `c` $\langle j \rangle$ $[u_j]$

  where j is the column identifier (an integer) and u_j is the associated upper bound (a positive integer). The parameter u_j may be omitted, in which case $u_j = 1$ is assumed.

- ▷ A 1-entry in the matrix $[a_{ij}]$ is declared with a line of the form

  `e` $\langle i \rangle$ $\langle j \rangle$

  where i is a row identifier and j is a column identifier. Each identifier must be declared before it may appear in an entry declaration.

- ▷ A column may be pushed into the solution stack with a line of the form

  `p` $\langle j \rangle$

  where j is a column identifier. No row, column, or entry declarations are permitted after a push.

The character `#` indicates a comment; any input after a comment character is skipped until either a newline or the end of file is encountered.

5.2 Output format

Each solution of the input instance is output by printing the associated solution stack. The contents of the stack are printed as a list of column identifiers, separated by spaces and terminated by a newline.

5.3 An example

The example given in Section 1 can be input to `solve` as follows.

```
r 1
r 2
r 3
r 4
c 1
c 2
c 3
c 4
c 5
e 1 1
e 1 2
e 1 5
e 2 1
e 2 4
e 3 2
e 3 5
e 4 1
e 4 2
e 4 3
```

Further examples can be found in the files `examples/solve-input*`.

Acknowledgments

The authors thank Patric Östergård and Jukka Suomela for comments and useful discussions. Research leading to the development of libexact was supported in part by the Academy of Finland (Grant 117499), the Graduate School in Electronics, Telecommunications and Automation (GETA), and the Foundation of Technology, Helsinki, Finland (Tekniikan Edistämissäätiö).

HIIT is a joint research institution of Helsinki University of Technology (TKK) and the University of Helsinki (UH).

This technical report is the user's guide to libexact, a software library for solving combinatorial exact covering problems.

The Helsinki Institute for Information Technology HIIT (in English)
Tietotekniikan tutkimuslaitos HIIT (in Finnish)

The Helsinki Institute for Information Technology HIIT conducts world-class research on future information technology.

Its research ranges from fundamental methods and technologies to novel applications and their impact on people and society.

HIIT's key competences are in Internet architecture and technologies, mobile and human-centric computing, user-created media, analysis of large sets of data, and probabilistic modeling of complex phenomena.

http://www.hiit.fi

HIIT Technical Reports 2008-1

ISBN 978-951-22-9488-6 (printed)
ISBN 978-951-22-9489-3 (electronic)
ISSN 1458-9478 (electronic)