bliss: Source Code Documentation

0.73 (Debian 0.73-2)

Generated by Doxygen 1.8.13

Contents

Chapter 1

Bliss

1.1 Introduction

This is the source code documentation of bliss, produced by running `doxygen` in the source directory. The algorithms and data structures used in bliss are documented in the papers found at the `bliss web site`.

1.2 Compiling

Compiling bliss in Linux should be easy, just execute

```
make
```

in the bliss source directory. This will produce the executable program `bliss` as well as the library file `libbliss.a` that can be linked in other programs. If you have the `GNU Multiple Precision Arithmetic Library` (GMP) installed in your machine, you can also use

```
make gmp
```

to enable exact computation of automorphism group sizes.

When linking the bliss library `libbliss.a` in other programs, remember to include the standard c++ library (and the GMP library if you compiled bliss to include it). For instance,

```
gcc -o test test.c -lstdc++ -lgmp -lbliss
```

1.3 The C++ language API

The C++ language API is the main API to bliss; all other APIs are just more or less complete variants of it. The C++ API consists basically of the public methods in the classes bliss::AbstractGraph, bliss::Graph, and bliss::Digraph. For an example of its use, see the source of the bliss executable.

1.4 The C language API

The C language API is given in the file bliss_C.h. It is currently more restricted than the C++ API so consider using the C++ API whenever possible.

Chapter 2

bliss executable

```
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <cassert>
#include <bliss/defs.hh>
#include <bliss/graph.hh>
#include <bliss/timer.hh>
#include <bliss/utils.hh>

/*
  Copyright (c) 2003-2015 Tommi Junttila
  Released under the GNU Lesser General Public License version 3.

  This file is part of bliss.

  bliss is free software: you can redistribute it and/or modify
  it under the terms of the GNU Lesser General Public License as published by
  the Free Software Foundation, version 3 of the License.

  bliss 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 Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public License
  along with bliss.  If not, see <http://www.gnu.org/licenses/>.
*/

/* Input file name */
static const char* infilename = 0;

static bool opt_directed = false;
static bool opt_canonize = false;
static const char* opt_output_can_file = 0;
static const char* opt_splitting_heuristics = "fsm";
static bool opt_use_failure_recording = true;
static bool opt_use_component_recursion = true;


/* Verbosity level and target stream */
static unsigned int verbose_level = 1;
static FILE* verbstr = stdout;



#if !defined(BLISS_COMPILED_DATE)
#define BLISS_COMPILED_DATE "compiled " __DATE__
#endif

static void
version(FILE* const fp)
{
  fprintf(fp,
"bliss version %s (" BLISS_COMPILED_DATE ")\n"
"Copyright (C) 2003-2015 Tommi Junttila.\n"
"\n"
"License LGPLv3+: GNU LGPL version 3 or later, <http://gnu.org/licenses/lgpl.html>.\n"
"This program comes with ABSOLUTELY NO WARRANTY. This is free software,\n"
"and you are welcome to redistribute it under certain conditions;\n"
"see COPYING and COPYING.LESSER for details.\n"
      , bliss::version
```

```
          );
}

static void
usage(FILE* const fp, const char* argv0)
{
  const char* program_name;

  program_name = rindex(argv0, '/');

  if(program_name) program_name++;
  else program_name = argv0;
  if(!program_name or *program_name == 0) program_name = "bliss";

  fprintf(fp,
"Usage: %s [options] [<graphfile>]\n"
"  Run bliss on <graphfile>.\n"
"Options:\n"
"  -directed   the input graph is directed\n"
"  -can        compute canonical form\n"
"  -ocan=f     compute canonical form and output it in file f\n"
"  -v=N        set verbose level to N [N >= 0, default: 1]\n"
"  -sh=X       select splitting heuristics, where X is\n"
"                f    first non-singleton cell\n"
"                fl   first largest non-singleton cell\n"
"                fs   first smallest non-singleton cell\n"
"                fm   first maximally non-trivially connected\n"
"                     non-singleton cell\n"
"                flm  first largest maximally non-trivially connected\n"
"                     non-singleton cell\n"
"                fsm  first smallest maximally non-trivially connected\n"
"                     non-singleton cell [default]\n"
"  -fr=X       use failure recording? [X=y/n, default: y]\n"
"  -cr=X       use component recursion? [X=y/n, default: y]\n"
"  -version    print the version number and exit\n"
"  -help       print this help and exit\n"
"\n"
          ,program_name
          );
}



static void
parse_options(const int argc, const char** argv)
{
  unsigned int tmp;
  for(int i = 1; i < argc; i++)
    {
      if(strcmp(argv[i], "-can") == 0)
        opt_canonize = true;
      else if((strncmp(argv[i], "-ocan=", 6) == 0) and (strlen(argv[i]) > 6))
        {
          opt_canonize = true;
          opt_output_can_file = argv[i]+6;
        }
      else if(sscanf(argv[i], "-v=%u", &tmp) == 1)
        verbose_level = tmp;
      else if(strcmp(argv[i], "-directed") == 0)
        opt_directed = true;
      else if(strcmp(argv[i], "-fr=n") == 0)
        opt_use_failure_recording = false;
      else if(strcmp(argv[i], "-fr=y") == 0)
        opt_use_failure_recording = true;
      else if(strcmp(argv[i], "-cr=n") == 0)
        opt_use_component_recursion = false;
      else if(strcmp(argv[i], "-cr=y") == 0)
        opt_use_component_recursion = true;
      else if((strncmp(argv[i], "-sh=", 4) == 0) and (strlen(argv[i]) > 4))
        {
          opt_splitting_heuristics = argv[i]+4;
        }
      else if(strcmp(argv[i], "-version") == 0)
        {
          version(stdout);
          exit(0);
        }
      else if(strcmp(argv[i], "-help") == 0)
        {
          usage(stdout, argv[0]);
          exit(0);
        }
      else if(argv[i][0] == '-')
        {
          fprintf(stderr, "Unknown command line argument `%s'\n", argv[i]);
          usage(stderr, argv[0]);
          exit(1);
```

```
        }
      else
        {
          if(infilename)
            {
              fprintf(stderr, "Too many file arguments\n");
              usage(stderr, argv[0]);
              exit(1);
            }
          else
            {
              infilename = argv[i];
            }
        }
    }
}



static void
report_aut(void* param, const unsigned int n, const unsigned int* aut)
{
  assert(param);
  fprintf((FILE*)param, "Generator: ");
  bliss::print_permutation((FILE*)param, n, aut, 1);
  fprintf((FILE*)param, "\n");
}



/* Output an error message and exit the whole program */
static void
_fatal(const char* fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  vfprintf(stderr, fmt, ap); fprintf(stderr, "\n");
  va_end(ap);
  exit(1);
}



int
main(const int argc, const char** argv)
{
  bliss::Timer timer;
  bliss::AbstractGraph* g = 0;

  parse_options(argc, argv);

  /* Parse splitting heuristics */
  bliss::Digraph::SplittingHeuristic shs_directed =
      bliss::Digraph::shs_fsm;
  bliss::Graph::SplittingHeuristic shs_undirected =
      bliss::Graph::shs_fsm;
  if(opt_directed)
    {
      if(strcmp(opt_splitting_heuristics, "f") == 0)
        shs_directed = bliss::Digraph::shs_f;
      else if(strcmp(opt_splitting_heuristics, "fs") == 0)
        shs_directed = bliss::Digraph::shs_fs;
      else if(strcmp(opt_splitting_heuristics, "fl") == 0)
        shs_directed = bliss::Digraph::shs_fl;
      else if(strcmp(opt_splitting_heuristics, "fm") == 0)
        shs_directed = bliss::Digraph::shs_fm;
      else if(strcmp(opt_splitting_heuristics, "fsm") == 0)
        shs_directed = bliss::Digraph::shs_fsm;
      else if(strcmp(opt_splitting_heuristics, "flm") == 0)
        shs_directed = bliss::Digraph::shs_flm;
      else
        _fatal("Illegal option -sh=%s, aborting", opt_splitting_heuristics);
    }
  else
    {
      if(strcmp(opt_splitting_heuristics, "f") == 0)
        shs_undirected = bliss::Graph::shs_f;
      else if(strcmp(opt_splitting_heuristics, "fs") == 0)
        shs_undirected = bliss::Graph::shs_fs;
      else if(strcmp(opt_splitting_heuristics, "fl") == 0)
        shs_undirected = bliss::Graph::shs_fl;
      else if(strcmp(opt_splitting_heuristics, "fm") == 0)
        shs_undirected = bliss::Graph::shs_fm;
      else if(strcmp(opt_splitting_heuristics, "fsm") == 0)
        shs_undirected = bliss::Graph::shs_fsm;
      else if(strcmp(opt_splitting_heuristics, "flm") == 0)
        shs_undirected = bliss::Graph::shs_flm;
```

```
    else
      _fatal("Illegal option -sh=%s, aborting", opt_splitting_heuristics);
  }

/* Open the input file */
FILE* infile = stdin;
if(infilename)
  {
    infile = fopen(infilename, "r");
    if(!infile)
      _fatal("Cannot not open `%s' for input, aborting", infilename);
  }

/* Read the graph from the file */
if(opt_directed)
  {
    /* Read directed graph in the DIMACS format */
    g = bliss::Digraph::read_dimacs(infile);
  }
else
  {
    /* Read undirected graph in the DIMACS format */
    g = bliss::Graph::read_dimacs(infile);
  }

if(infile != stdin)
  fclose(infile);

if(!g)
  _fatal("Failed to read the graph, aborting");

if(verbose_level >= 2)
  {
    fprintf(verbstr, "Graph read in %.2f seconds\n", timer.get_duration());
    fflush(verbstr);
  }


bliss::Stats stats;

/* Set splitting heuristics and verbose level */
if(opt_directed)
  ((bliss::Digraph*)g)->set_splitting_heuristic(shs_directed);
else
  ((bliss::Graph*)g)->set_splitting_heuristic(shs_undirected);
g->set_verbose_level(verbose_level);
g->set_verbose_file(verbstr);
g->set_failure_recording(opt_use_failure_recording);
g->set_component_recursion(opt_use_component_recursion);


if(opt_canonize == false)
  {
    /* No canonical labeling, only automorphism group */
    g->find_automorphisms(stats, &report_aut, stdout);
  }
else
  {
    /* Canonical labeling and automorphism group */
    const unsigned int* cl = g->canonical_form(stats, &report_aut, stdout);

    fprintf(stdout, "Canonical labeling: ");
    bliss::print_permutation(stdout, g->
    get_nof_vertices(), cl, 1);
    fprintf(stdout, "\n");

    if(opt_output_can_file)
      {
        bliss::AbstractGraph* cf = g->permute(cl);
        FILE* const fp = fopen(opt_output_can_file, "w");
        if(!fp)
          _fatal("Cannot open `%s' for outputting the canonical form, aborting", opt_output_can_file);
        cf->write_dimacs(fp);
        fclose(fp);
        delete cf;
      }
  }

/* Output search statistics */
if(verbose_level > 0 and verbstr)
  stats.print(verbstr);

if(verbose_level > 0)
  {
    fprintf(verbstr, "Total time:\t%.2f seconds\n", timer.get_duration());
    fflush(verbstr);
  }
```

```
  delete g; g = 0;

  return 0;
}
```

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Namespace Documentation

7.1 bliss Namespace Reference

Classes

- class AbstractGraph

  An abstract base class for different types of graphs.
- class BigNum

  A very simple class for big integers (or approximation of them).
- class Digraph

  The class for directed, vertex colored graphs.
- class Graph

  The class for undirected, vertex colored graphs.
- class Stats

  Statistics returned by the bliss search algorithm.

Functions

- void print_permutation (FILE *const fp, const unsigned int N, const unsigned int *perm, const unsigned int offset)
- void print_permutation (FILE *const fp, const std::vector< unsigned int > &perm, const unsigned int offset)
- bool is_permutation (const unsigned int N, const unsigned int *perm)
- bool is_permutation (const std::vector< unsigned int > &perm)

7.1.1 Detailed Description

The namespace bliss contains all the classes and functions of the bliss tool except for the C programming language API.

7.1.2 Function Documentation

7.1.2.1 is_permutation() [1/2]

```
bool bliss::is_permutation (
            const unsigned int N,
            const unsigned int * perm )
```

Check whether *perm* is a valid permutation on {0,...,N-1}. Slow, mainly for debugging and validation purposes.

7.1.2.2 is_permutation() [2/2]

```
bool bliss::is_permutation (
            const std::vector< unsigned int > & perm )
```

Check whether *perm* is a valid permutation on {0,...,N-1}. Slow, mainly for debugging and validation purposes.

7.1.2.3 print_permutation() [1/2]

```
void bliss::print_permutation (
            FILE * fp,
            const unsigned int N,
            const unsigned int * perm,
            const unsigned int offset = 0 )
```

Print the permutation *perm* of {0,...,N-1} in the cycle format in the file stream *fp*. The amount *offset* is added to each element before printing, e.g. the permutation (2 4) is printed as (3 5) when *offset* is 1.

7.1.2.4 print_permutation() [2/2]

```
void bliss::print_permutation (
            FILE * fp,
            const std::vector< unsigned int > & perm,
            const unsigned int offset = 0 )
```

Print the permutation *perm* of {0,...,N-1} in the cycle format in the file stream *fp*. The amount *offset* is added to each element before printing, e.g. the permutation (2 4) is printed as (3 5) when *offset* is 1.

Chapter 8

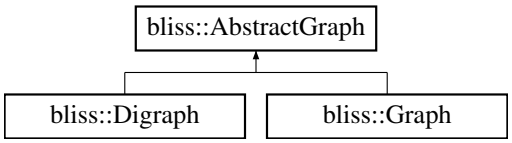
Class Documentation

8.1 bliss::AbstractGraph Class Reference

An abstract base class for different types of graphs.

`#include <graph.hh>`

Inheritance diagram for bliss::AbstractGraph:



Public Member Functions

- void set_verbose_level (const unsigned int level)
- void set_verbose_file (FILE ∗const fp)
- virtual unsigned int add_vertex (const unsigned int color=0)=0
- virtual void add_edge (const unsigned int source, const unsigned int target)=0
- virtual void change_color (const unsigned int vertex, const unsigned int color)=0
- virtual bool is_automorphism (const std::vector< unsigned int > &perm) const
- void set_failure_recording (const bool active)
- void set_component_recursion (const bool active)
- virtual unsigned int get_nof_vertices () const =0
- virtual AbstractGraph ∗ permute (const unsigned int ∗const perm) const =0
- void find_automorphisms (Stats &stats, void(∗hook)(void ∗user_param, unsigned int n, const unsigned int ∗aut), void ∗hook_user_param)
- const unsigned int ∗ canonical_form (Stats &stats, void(∗hook)(void ∗user_param, unsigned int n, const unsigned int ∗aut), void ∗hook_user_param)
- virtual void write_dimacs (FILE ∗const fp)=0
- virtual void write_dot (FILE ∗const fp)=0
- virtual void write_dot (const char ∗const file_name)=0
- virtual unsigned int get_hash ()=0
- void set_long_prune_activity (const bool active)

Protected Attributes

- unsigned int cr_level

8.1.1 Detailed Description

An abstract base class for different types of graphs.

8.1.2 Member Function Documentation

8.1.2.1 add_edge()

```
virtual void bliss::AbstractGraph::add_edge (
            const unsigned int source,
            const unsigned int target )  [pure virtual]
```

Add an edge between vertices *source* and *target*. Duplicate edges between vertices are ignored but try to avoid introducing them in the first place as they are not ignored immediately but will consume memory and computation resources for a while.

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.2 add_vertex()

```
virtual unsigned int bliss::AbstractGraph::add_vertex (
            const unsigned int color = 0 )  [pure virtual]
```

Add a new vertex with color *color* in the graph and return its index.

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.3 canonical_form()

```
const unsigned int * bliss::AbstractGraph::canonical_form (
            Stats & stats,
            void(*)(void *user_param, unsigned int n, const unsigned int *aut) hook,
            void * hook_user_param )
```

Otherwise the same as find_automorphisms() except that a canonical labeling of the graph (a bijection on {0,...,get_nof_vertices()-1}) is returned. The memory allocated for the returned canonical labeling will remain valid only until the next call to a member function with the exception that constant member functions (for example, bliss::Graph::permute()) can be called without invalidating the labeling. To compute the canonical version of an undirected graph, call this function and then bliss::Graph::permute() with the returned canonical labeling. Note that the computed canonical version may depend on the applied version of bliss as well as on some other options (for instance, the splitting heuristic selected with bliss::Graph::set_splitting_heuristic()).

8.1.2.4 change_color()

```
virtual void bliss::AbstractGraph::change_color (
            const unsigned int vertex,
            const unsigned int color )  [pure virtual]
```

Change the color of the vertex *vertex* to *color*.

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.5 find_automorphisms()

```
void bliss::AbstractGraph::find_automorphisms (
            Stats & stats,
            void(*)(void *user_param, unsigned int n, const unsigned int *aut) hook,
            void * hook_user_param )
```

Find a set of generators for the automorphism group of the graph. The function *hook* (if non-null) is called each time a new generator for the automorphism group is found. The first argument *user_param* for the hook is the *hook_user_param* given below, the second argument *n* is the length of the automorphism (equal to get_nof_vertices()) and the third argument *aut* is the automorphism (a bijection on {0,...,get_nof_vertices()-1}). The memory for the automorphism *aut* will be invalidated immediately after the return from the hook function; if you want to use the automorphism later, you have to take a copy of it. Do not call any member functions in the hook. The search statistics are copied in *stats*.

8.1.2.6 get_hash()

```
virtual unsigned int bliss::AbstractGraph::get_hash ( )  [pure virtual]
```

Get a hash value for the graph.

Returns

the hash value

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.7 get_nof_vertices()

```
virtual unsigned int bliss::AbstractGraph::get_nof_vertices ( ) const  [pure virtual]
```

Return the number of vertices in the graph.

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.8 is_automorphism()

```
bool bliss::AbstractGraph::is_automorphism (
            const std::vector< unsigned int > & perm ) const  [virtual]
```

Check whether *perm* is an automorphism of this graph. Unoptimized, mainly for debugging purposes.

Reimplemented in bliss::Digraph, and bliss::Graph.

8.1.2.9 permute()

```
virtual AbstractGraph* bliss::AbstractGraph::permute (
            const unsigned int *const perm ) const  [pure virtual]
```

Return a new graph that is the result of applying the permutation *perm* to this graph. This graph is not modified. *perm* must contain N=this.get_nof_vertices() elements and be a bijection on {0,1,...,N-1}, otherwise the result is undefined or a segfault.

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.10 set_component_recursion()

```
void bliss::AbstractGraph::set_component_recursion (
            const bool active )  [inline]
```

Activate/deactivate component recursion. The choice affects the computed canonical labelings; therefore, if you want to compare whether two graphs are isomorphic by computing and comparing (for equality) their canonical versions, be sure to use the same choice for both graphs. May not be called during the search, i.e. from an automorphism reporting hook function.

Parameters

active	if true, activate component recursion, deactivate otherwise

8.1.2.11 set_failure_recording()

```
void bliss::AbstractGraph::set_failure_recording (
            const bool active )  [inline]
```

Activate/deactivate failure recording. May not be called during the search, i.e. from an automorphism reporting hook function.

Parameters

active	if true, activate failure recording, deactivate otherwise

8.1.2.12 set_long_prune_activity()

```
void bliss::AbstractGraph::set_long_prune_activity (
            const bool active )  [inline]
```

Disable/enable the "long prune" method. The choice affects the computed canonical labelings; therefore, if you want to compare whether two graphs are isomorphic by computing and comparing (for equality) their canonical versions, be sure to use the same choice for both graphs. May not be called during the search, i.e. from an automorphism reporting hook function.

Parameters

active	if true, activate "long prune", deactivate otherwise

8.1.2.13 set_verbose_file()

```
void bliss::AbstractGraph::set_verbose_file (
            FILE *const fp )
```

Set the file stream for the verbose output.

Parameters

fp	the file stream; if null, no verbose output is written

8.1.2.14 set_verbose_level()

```
void bliss::AbstractGraph::set_verbose_level (
            const unsigned int level )
```

Set the verbose output level for the algorithms.

Parameters

level	the level of verbose output, 0 means no verbose output

8.1.2.15 write_dimacs()

```
virtual void bliss::AbstractGraph::write_dimacs (
            FILE *const fp )  [pure virtual]
```

Write the graph to a file in a variant of the DIMACS format. See the bliss website for the definition of the file format. Note that in the DIMACS file the vertices are numbered from 1 to N while in this C++ API they are from 0 to N-1. Thus the vertex n in the file corresponds to the vertex n-1 in the API.

Parameters

fp	the file stream where the graph is written

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.16 write_dot() [1/2]

```
virtual void bliss::AbstractGraph::write_dot (
            FILE *const fp )  [pure virtual]
```

Write the graph to a file in the graphviz dotty format.

Parameters

fp	the file stream where the graph is written

Implemented in bliss::Digraph, and bliss::Graph.

8.1.2.17 write_dot() [2/2]

```
virtual void bliss::AbstractGraph::write_dot (
            const char *const file_name )  [pure virtual]
```

Write the graph in a file in the graphviz dotty format. Do nothing if the file cannot be written.

Parameters

file_name	the name of the file to which the graph is written

Implemented in bliss::Digraph, and bliss::Graph.

8.1.3 Member Data Documentation

8.1.3.1 cr_level

```
unsigned int bliss::AbstractGraph::cr_level  [protected]
```

The currently traversed component

The documentation for this class was generated from the following files:

- graph.hh
- graph.cc

8.2 bliss::BigNum Class Reference

A very simple class for big integers (or approximation of them).

```
#include <bignum.hh>
```

Public Member Functions

- BigNum ()
- void assign (const int n)
- void multiply (const int n)
- size_t print (FILE ∗const fp) const

8.2.1 Detailed Description

A very simple class for big integers (or approximation of them).

If the compile time flag BLISS_USE_GMP is set, then the GNU Multiple Precision Arithmetic library (GMP) is used to obtain arbitrary precision, otherwise "long double" is used to approximate big integers.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 BigNum()

```
bliss::BigNum::BigNum ( )  [inline]
```

Create a new big number and set it to zero.

8.2.3 Member Function Documentation

8.2.3.1 assign()

```
void bliss::BigNum::assign (
            const int n )  [inline]
```

Set the number to *n*.

8.2.3.2 multiply()

```
void bliss::BigNum::multiply (
            const int n )  [inline]
```

Multiply the number with *n*.

8.2.3.3 print()

```
size_t bliss::BigNum::print (
            FILE *const fp ) const  [inline]
```

Print the number in the file stream *fp*.

The documentation for this class was generated from the following file:

- bignum.hh

8.3 bliss_stats_struct Struct Reference

The C API version of the statistics returned by the bliss search algorithm.

```
#include <bliss_C.h>
```

Public Attributes

- long double group_size_approx
- long unsigned int nof_nodes
- long unsigned int nof_leaf_nodes
- long unsigned int nof_bad_nodes
- long unsigned int nof_canupdates
- long unsigned int nof_generators
- unsigned long int max_level

8.3.1 Detailed Description

The C API version of the statistics returned by the bliss search algorithm.

8.3.2 Member Data Documentation

8.3.2.1 group_size_approx

```
long double bliss_stats_struct::group_size_approx
```

An approximation (due to possible rounding errors) of the size of the automorphism group.

8.3.2.2 max_level

```
unsigned long int bliss_stats_struct::max_level
```

The maximal depth of the search tree.

8.3.2.3 nof_bad_nodes

```
long unsigned int bliss_stats_struct::nof_bad_nodes
```

The number of bad nodes in the search tree.

8.3.2.4 nof_canupdates

```
long unsigned int bliss_stats_struct::nof_canupdates
```

The number of canonical representative updates.

8.3.2.5 nof_generators

```
long unsigned int bliss_stats_struct::nof_generators
```

The number of generator permutations.

8.3.2.6 nof_leaf_nodes

```
long unsigned int bliss_stats_struct::nof_leaf_nodes
```

The number of leaf nodes in the search tree.

8.3.2.7 nof_nodes

```
long unsigned int bliss_stats_struct::nof_nodes
```

The number of nodes in the search tree.

The documentation for this struct was generated from the following file:

- bliss_C.h

8.4 bliss::Partition::Cell Class Reference

Data structure for holding information about a cell in a Partition.

`#include <partition.hh>`

Public Member Functions

- bool is_unit () const
- bool is_in_splitting_queue () const

8.4.1 Detailed Description

Data structure for holding information about a cell in a Partition.

8.4.2 Member Function Documentation

8.4.2.1 is_in_splitting_queue()

```
bool bliss::Partition::Cell::is_in_splitting_queue ( ) const  [inline]
```

Is this cell in splitting queue?

8.4.2.2 is_unit()

```
bool bliss::Partition::Cell::is_unit ( ) const  [inline]
```

Is this a unit cell?

The documentation for this class was generated from the following file:

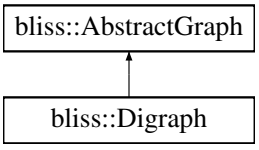
- partition.hh

8.5 bliss::Digraph Class Reference

The class for directed, vertex colored graphs.

`#include <graph.hh>`

Inheritance diagram for bliss::Digraph:



Public Types

- enum SplittingHeuristic {
  shs_f = 0, shs_fs, shs_fl, shs_fm,
  shs_fsm, shs_flm }

Public Member Functions

- Digraph (const unsigned int N=0)
- ~Digraph ()
- void write_dimacs (FILE *const fp)
- void write_dot (FILE *const fp)
- void write_dot (const char *const file_name)
- bool is_automorphism (const std::vector< unsigned int > &perm) const
- virtual unsigned int get_hash ()
- unsigned int get_nof_vertices () const
- unsigned int add_vertex (const unsigned int color=0)
- void add_edge (const unsigned int source, const unsigned int target)
- void change_color (const unsigned int vertex, const unsigned int color)
- int cmp (Digraph &other)
- void set_splitting_heuristic (SplittingHeuristic shs)
- Digraph * permute (const unsigned int *const perm) const

Static Public Member Functions

- static Digraph * read_dimacs (FILE *const fp, FILE *const errstr=stderr)

Additional Inherited Members

8.5.1 Detailed Description

The class for directed, vertex colored graphs.

Multiple edges between vertices are not allowed (i.e., are ignored).

8.5.2 Member Enumeration Documentation

8.5.2.1 SplittingHeuristic

`enum bliss::Digraph::SplittingHeuristic`

The possible splitting heuristics. The selected splitting heuristics affects the computed canonical labelings; therefore, if you want to compare whether two graphs are isomorphic by computing and comparing (for equality) their canonical versions, be sure to use the same splitting heuristics for both graphs.

Enumerator

shs_f	First non-unit cell. Very fast but may result in large search spaces on difficult graphs. Use for large but easy graphs.
shs_fs	First smallest non-unit cell. Fast, should usually produce smaller search spaces than shs_f.
shs_fl	First largest non-unit cell. Fast, should usually produce smaller search spaces than shs_f.
shs_fm	First maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than shs_f, shs_fs, and shs_fl.
shs_fsm	First smallest maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than shs_f, shs_fs, and shs_fl.
shs_flm	First largest maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than shs_f, shs_fs, and shs_fl.

8.5.3 Constructor & Destructor Documentation

8.5.3.1 Digraph()

```
bliss::Digraph::Digraph (
            const unsigned int N = 0 )
```

Create a new directed graph with *N* vertices and no edges.

8.5.3.2 ~Digraph()

```
bliss::Digraph::~Digraph ( )
```

Destroy the graph.

8.5.4 Member Function Documentation

8.5.4.1 add_edge()

```
void bliss::Digraph::add_edge (
            const unsigned int source,
            const unsigned int target )  [virtual]
```

Add an edge from the vertex *source* to the vertex *target*. Duplicate edges are ignored but try to avoid introducing them in the first place as they are not ignored immediately but will consume memory and computation resources for a while.

Implements bliss::AbstractGraph.

8.5.4.2 add_vertex()

```
unsigned int bliss::Digraph::add_vertex (
            const unsigned int color = 0 )  [virtual]
```

Add a new vertex with color 'color' in the graph and return its index.

Implements bliss::AbstractGraph.

8.5.4.3 change_color()

```
void bliss::Digraph::change_color (
            const unsigned int vertex,
            const unsigned int color )  [virtual]
```

Change the color of the vertex 'vertex' to 'color'.

Implements bliss::AbstractGraph.

8.5.4.4 cmp()

```
int bliss::Digraph::cmp (
            Digraph & other )
```

Compare this graph with the graph *other*. Returns 0 if the graphs are equal, and a negative (positive) integer if this graph is "smaller than" ("greater than", resp.) than *other*.

8.5.4.5 get_hash()

```
unsigned int bliss::Digraph::get_hash ( )  [virtual]
```

Get a hash value for the graph.

Returns

the hash value

Implements bliss::AbstractGraph.

8.5.4.6 get_nof_vertices()

```
unsigned int bliss::Digraph::get_nof_vertices ( ) const  [inline], [virtual]
```

Return the number of vertices in the graph.

Implements bliss::AbstractGraph.

8.5.4.7 is_automorphism()

```
bool bliss::Digraph::is_automorphism (
            const std::vector< unsigned int > & perm ) const  [virtual]
```

Check whether *perm* is an automorphism of this graph. Unoptimized, mainly for debugging purposes.

Reimplemented from bliss::AbstractGraph.

8.5.4.8 permute()

```
Digraph * bliss::Digraph::permute (
            const unsigned int *const perm ) const  [virtual]
```

Return a new graph that is the result of applying the permutation *perm* to this graph. This graph is not modified. *perm* must contain N=this.get_nof_vertices() elements and be a bijection on {0,1,...,N-1}, otherwise the result is undefined or a segfault.

Implements bliss::AbstractGraph.

8.5.4.9 read_dimacs()

```
Digraph * bliss::Digraph::read_dimacs (
            FILE *const fp,
            FILE *const errstr = stderr )  [static]
```

Read the graph from the file *fp* in a variant of the DIMACS format. See the bliss website for the definition of the file format. Note that in the DIMACS file the vertices are numbered from 1 to N while in this C++ API they are from 0 to N-1. Thus the vertex n in the file corresponds to the vertex n-1 in the API.

Parameters

fp	the file stream for the graph file
errstr	if non-null, the possible error messages are printed in this file stream

Returns

a new Digraph object or 0 if reading failed for some reason

8.5.4.10 set_splitting_heuristic()

```
void bliss::Digraph::set_splitting_heuristic (
            SplittingHeuristic shs )  [inline]
```

Set the splitting heuristic used by the automorphism and canonical labeling algorithm. The selected splitting heuristics affects the computed canonical labelings; therefore, if you want to compare whether two graphs are isomorphic by computing and comparing (for equality) their canonical versions, be sure to use the same splitting heuristics for both graphs.

8.5.4.11 write_dimacs()

```
void bliss::Digraph::write_dimacs (
            FILE *const fp )  [virtual]
```

Write the graph to a file in a variant of the DIMACS format. See the bliss website for the definition of the file format. Note that in the DIMACS file the vertices are numbered from 1 to N while in this C++ API they are from 0 to N-1. Thus the vertex n in the file corresponds to the vertex n-1 in the API.

Parameters

fp	the file stream where the graph is written

Implements bliss::AbstractGraph.

8.5.4.12 write_dot() [1/2]

```
void bliss::Digraph::write_dot (
            FILE *const fp )  [virtual]
```

Write the graph to a file in the graphviz dotty format.

Parameters

fp	the file stream where the graph is written

Implements bliss::AbstractGraph.

8.5.4.13 write_dot() [2/2]

```
void bliss::Digraph::write_dot (
            const char *const file_name )  [virtual]
```

Write the graph in a file in the graphviz dotty format. Do nothing if the file cannot be written.

Parameters

file_name	the name of the file to which the graph is written

Implements bliss::AbstractGraph.

The documentation for this class was generated from the following files:

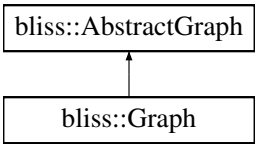
- graph.hh
- graph.cc

8.6 bliss::Graph Class Reference

The class for undirected, vertex colored graphs.

`#include <graph.hh>`

Inheritance diagram for bliss::Graph:



Public Types

- enum SplittingHeuristic {
  shs_f = 0, shs_fs, shs_fl, shs_fm,
  shs_fsm, shs_flm }

Public Member Functions

- Graph (const unsigned int N=0)
- ~Graph ()
- void write_dimacs (FILE *const fp)
- void write_dot (FILE *const fp)
- void write_dot (const char *const file_name)
- bool is_automorphism (const std::vector< unsigned int > &perm) const
- virtual unsigned int get_hash ()
- unsigned int get_nof_vertices () const
- Graph * permute (const unsigned int *const perm) const
- unsigned int add_vertex (const unsigned int color=0)
- void add_edge (const unsigned int v1, const unsigned int v2)
- void change_color (const unsigned int vertex, const unsigned int color)
- int cmp (Graph &other)
- void set_splitting_heuristic (const SplittingHeuristic shs)

Static Public Member Functions

- static Graph * read_dimacs (FILE *const fp, FILE *const errstr=stderr)

Additional Inherited Members

8.6.1 Detailed Description

The class for undirected, vertex colored graphs.

Multiple edges between vertices are not allowed (i.e., are ignored).

8.6.2 Member Enumeration Documentation

8.6.2.1 SplittingHeuristic

```
enum bliss::Graph::SplittingHeuristic
```

The possible splitting heuristics. The selected splitting heuristics affects the computed canonical labelings; therefore, if you want to compare whether two graphs are isomorphic by computing and comparing (for equality) their canonical versions, be sure to use the same splitting heuristics for both graphs.

Enumerator

shs_f	First non-unit cell. Very fast but may result in large search spaces on difficult graphs. Use for large but easy graphs.
shs_fs	First smallest non-unit cell. Fast, should usually produce smaller search spaces than shs_f.
shs_fl	First largest non-unit cell. Fast, should usually produce smaller search spaces than shs_f.
shs_fm	First maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than shs_f, shs_fs, and shs_fl.
shs_fsm	First smallest maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than shs_f, shs_fs, and shs_fl.
shs_flm	First largest maximally non-trivially connected non-unit cell. Not so fast, should usually produce smaller search spaces than shs_f, shs_fs, and shs_fl.

8.6.3 Constructor & Destructor Documentation

8.6.3.1 Graph()

```
bliss::Graph::Graph (
            const unsigned int N = 0 )
```

Create a new graph with *N* vertices and no edges.

8.6.3.2 ~Graph()

```
bliss::Graph::~Graph ( )
```

Destroy the graph.

8.6.4 Member Function Documentation

8.6.4.1 add_edge()

```
void bliss::Graph::add_edge (
            const unsigned int v1,
            const unsigned int v2 )  [virtual]
```

Add an edge between vertices *v1* and *v2*. Duplicate edges between vertices are ignored but try to avoid introducing them in the first place as they are not ignored immediately but will consume memory and computation resources for a while.

Implements bliss::AbstractGraph.

8.6.4.2 add_vertex()

```
unsigned int bliss::Graph::add_vertex (
            const unsigned int color = 0 )  [virtual]
```

Add a new vertex with color *color* in the graph and return its index.

Implements bliss::AbstractGraph.

8.6.4.3 change_color()

```
void bliss::Graph::change_color (
            const unsigned int vertex,
            const unsigned int color )  [virtual]
```

Change the color of the vertex *vertex* to *color*.

Implements bliss::AbstractGraph.

8.6.4.4 cmp()

```
int bliss::Graph::cmp (
            Graph & other )
```

Compare this graph with the graph *other*. Returns 0 if the graphs are equal, and a negative (positive) integer if this graph is "smaller than" ("greater than", resp.) than *other*.

8.6.4.5 get_hash()

```
unsigned int bliss::Graph::get_hash ( )  [virtual]
```

Get a hash value for the graph.

Returns

the hash value

Implements bliss::AbstractGraph.

8.6.4.6 get_nof_vertices()

```
unsigned int bliss::Graph::get_nof_vertices ( ) const  [inline], [virtual]
```

Return the number of vertices in the graph.

Implements bliss::AbstractGraph.

8.6.4.7 is_automorphism()

```
bool bliss::Graph::is_automorphism (
            const std::vector< unsigned int > & perm ) const  [virtual]
```

Check whether *perm* is an automorphism of this graph. Unoptimized, mainly for debugging purposes.

Reimplemented from bliss::AbstractGraph.

8.6.4.8 permute()

```
Graph * bliss::Graph::permute (
            const unsigned int *const perm ) const  [virtual]
```

Return a new graph that is the result of applying the permutation *perm* to this graph. This graph is not modified. *perm* must contain N=this.get_nof_vertices() elements and be a bijection on {0,1,...,N-1}, otherwise the result is undefined or a segfault.

Implements bliss::AbstractGraph.

8.6.4.9 read_dimacs()

```
Graph * bliss::Graph::read_dimacs (
            FILE *const fp,
            FILE *const errstr = stderr )  [static]
```

Read the graph from the file *fp* in a variant of the DIMACS format. See the bliss website for the definition of the file format. Note that in the DIMACS file the vertices are numbered from 1 to N while in this C++ API they are from 0 to N-1. Thus the vertex n in the file corresponds to the vertex n-1 in the API.

Parameters

fp	the file stream for the graph file
errstr	if non-null, the possible error messages are printed in this file stream

Returns

a new Graph object or 0 if reading failed for some reason

8.6.4.10 set_splitting_heuristic()

```
void bliss::Graph::set_splitting_heuristic (
            const SplittingHeuristic shs )  [inline]
```

Set the splitting heuristic used by the automorphism and canonical labeling algorithm. The selected splitting heuristics affects the computed canonical labelings; therefore, if you want to compare whether two graphs are isomorphic by computing and comparing (for equality) their canonical versions, be sure to use the same splitting heuristics for both graphs.

8.6.4.11 write_dimacs()

```
void bliss::Graph::write_dimacs (
            FILE *const fp )  [virtual]
```

Write the graph to a file in a variant of the DIMACS format. See the bliss website for the definition of the file format.

Implements bliss::AbstractGraph.

8.6.4.12 write_dot() [1/2]

```
void bliss::Graph::write_dot (
            FILE *const fp )  [virtual]
```

Write the graph to a file in the graphviz dotty format.

Parameters

fp	the file stream where the graph is written

Implements bliss::AbstractGraph.

8.6.4.13 write_dot() [2/2]

```
void bliss::Graph::write_dot (
            const char *const file_name )  [virtual]
```

Write the graph in a file in the graphviz dotty format. Do nothing if the file cannot be written.

Parameters

file_name	the name of the file to which the graph is written

Implements bliss::AbstractGraph.

The documentation for this class was generated from the following files:

- graph.hh
- graph.cc

8.7 bliss::Stats Class Reference

Statistics returned by the bliss search algorithm.

`#include <graph.hh>`

Public Member Functions

- size_t print (FILE *const fp) const
- long double get_group_size_approx () const
- long unsigned int get_nof_nodes () const
- long unsigned int get_nof_leaf_nodes () const
- long unsigned int get_nof_bad_nodes () const
- long unsigned int get_nof_canupdates () const
- long unsigned int get_nof_generators () const
- unsigned long int get_max_level () const

8.7.1 Detailed Description

Statistics returned by the bliss search algorithm.

8.7.2 Member Function Documentation

8.7.2.1 get_group_size_approx()

```
long double bliss::Stats::get_group_size_approx ( ) const  [inline]
```

An approximation (due to possible overflows/rounding errors) of the size of the automorphism group.

8.7.2.2 get_max_level()

```
unsigned long int bliss::Stats::get_max_level ( ) const  [inline]
```

The maximal depth of the search tree.

8.7.2.3 get_nof_bad_nodes()

```
long unsigned int bliss::Stats::get_nof_bad_nodes ( ) const  [inline]
```

The number of bad nodes in the search tree.

8.7.2.4 get_nof_canupdates()

```
long unsigned int bliss::Stats::get_nof_canupdates ( ) const  [inline]
```

The number of canonical representative updates.

8.7.2.5 get_nof_generators()

```
long unsigned int bliss::Stats::get_nof_generators ( ) const  [inline]
```

The number of generator permutations.

8.7.2.6 get_nof_leaf_nodes()

```
long unsigned int bliss::Stats::get_nof_leaf_nodes ( ) const  [inline]
```

The number of leaf nodes in the search tree.

8.7.2.7 get_nof_nodes()

```
long unsigned int bliss::Stats::get_nof_nodes ( ) const  [inline]
```

The number of nodes in the search tree.

8.7.2.8 print()

```
size_t bliss::Stats::print (
            FILE *const fp ) const  [inline]
```

Print the statistics.

The documentation for this class was generated from the following file:

- graph.hh

Chapter 9

File Documentation

9.1 bliss_C.h File Reference

The bliss C API.

```
#include <stdlib.h>
#include <stdio.h>
```

Classes

- struct bliss_stats_struct

  The C API version of the statistics returned by the bliss search algorithm.

Typedefs

- typedef struct bliss_graph_struct BlissGraph

  The true bliss graph is hiding behind this typedef.
- typedef struct bliss_stats_struct BlissStats

  The C API version of the statistics returned by the bliss search algorithm.

Functions

- BlissGraph * bliss_new (const unsigned int N)
- BlissGraph * bliss_read_dimacs (FILE *fp)
- void bliss_write_dimacs (BlissGraph *graph, FILE *fp)
- void bliss_release (BlissGraph *graph)
- void bliss_write_dot (BlissGraph *graph, FILE *fp)
- unsigned int bliss_get_nof_vertices (BlissGraph *graph)
- unsigned int bliss_add_vertex (BlissGraph *graph, unsigned int c)
- void bliss_add_edge (BlissGraph *graph, unsigned int v1, unsigned int v2)
- int bliss_cmp (BlissGraph *graph1, BlissGraph *graph2)
- unsigned int bliss_hash (BlissGraph *graph)
- BlissGraph * bliss_permute (BlissGraph *graph, const unsigned int *perm)
- void bliss_find_automorphisms (BlissGraph *graph, void(*hook)(void *user_param, unsigned int N, const unsigned int *aut), void *hook_user_param, BlissStats *stats)
- const unsigned int * bliss_find_canonical_labeling (BlissGraph *graph, void(*hook)(void *user_param, unsigned int N, const unsigned int *aut), void *hook_user_param, BlissStats *stats)

9.1.1 Detailed Description

The bliss C API.

This is the C language API to bliss. Note that this C API is only a subset of the C++ API; please consider using the C++ API whenever possible.

9.1.2 Function Documentation

9.1.2.1 bliss_add_edge()

```
void bliss_add_edge (
            BlissGraph * graph,
            unsigned int v1,
            unsigned int v2 )
```

Add a new undirected edge in the graph. *v1* and *v2* are vertex indices returned by bliss_add_vertex(). If duplicate edges are added, they will be ignored (however, they are not necessarily physically ignored immediately but may consume memory for a while so please try to avoid adding duplicate edges whenever possible).

9.1.2.2 bliss_add_vertex()

```
unsigned int bliss_add_vertex (
            BlissGraph * graph,
            unsigned int c )
```

Add a new vertex with color *c* in the graph *graph* and return its index. The vertex indices are always in the range [0,bliss::bliss_get_nof_vertices(*bliss*)-1].

9.1.2.3 bliss_cmp()

```
int bliss_cmp (
            BlissGraph * graph1,
            BlissGraph * graph2 )
```

Compare two graphs according to a total order. Return -1, 0, or 1 if the first graph was smaller than, equal to, or greater than, resp., the other graph. If 0 is returned, then the graphs have the same number vertices, the vertices in them are colored in the same way, and they contain the same edges; that is, the graphs are equal.

9.1.2.4 bliss_find_automorphisms()

```
void bliss_find_automorphisms (
            BlissGraph * graph,
            void(*)(void *user_param, unsigned int N, const unsigned int *aut) hook,
            void * hook_user_param,
            BlissStats * stats )
```

Find a set of generators for the automorphism group of the graph. The hook function *hook* (if non-null) is called each time a new generator for the automorphism group is found. The first argument *user_param* for the hook function is the *hook_user_param* argument, the second argument *N* is the length of the automorphism (equal to bliss::bliss_get_nof_vertices(*graph*)) and the third argument *aut* is the automorphism (a bijection on {0,...,N-1}). The memory for the automorphism *aut* will be invalidated immediately after the return from the hook; if you want to use the automorphism later, you have to take a copy of it. Do not call bliss_* functions in the hook. If *stats* is non-null, then some search statistics are copied there.

9.1.2.5 bliss_find_canonical_labeling()

```
const unsigned int* bliss_find_canonical_labeling (
             BlissGraph * graph,
             void(*)(void *user_param, unsigned int N, const unsigned int *aut) hook,
             void * hook_user_param,
             BlissStats * stats )
```

Otherwise the same as bliss_find_automorphisms() except that a canonical labeling for the graph (a bijection on {0,...,N-1}) is returned. The returned canonical labeling will remain valid only until the next call to a bliss_∗ function with the exception that bliss_permute() can be called without invalidating the labeling. To compute the canonical version of a graph, call this function and then bliss_permute() with the returned canonical labeling. Note that the computed canonical version may depend on the applied version of bliss.

9.1.2.6 bliss_get_nof_vertices()

```
unsigned int bliss_get_nof_vertices (
             BlissGraph * graph )
```

Return the number of vertices in the graph.

9.1.2.7 bliss_hash()

```
unsigned int bliss_hash (
             BlissGraph * graph )
```

Get a hash value for the graph.

9.1.2.8 bliss_new()

```
BlissGraph* bliss_new (
             const unsigned int N )
```

Create a new graph instance with *N* vertices and no edges. *N* can be zero and bliss_add_vertex() called afterwards to add new vertices on-the-fly.

9.1.2.9 bliss_permute()

```
BlissGraph* bliss_permute (
             BlissGraph * graph,
             const unsigned int * perm )
```

Permute the graph with the given permutation *perm*. Returns the permuted graph, the original graph is not modified. The argument *perm* should be an array of N=bliss::bliss_get_nof_vertices(*graph*) elements describing a bijection on {0,...,N-1}.

9.1.2.10 bliss_read_dimacs()

```
BlissGraph* bliss_read_dimacs (
            FILE * fp )
```

Read an undirected graph from a file in the DIMACS format into a new bliss instance. Returns 0 if an error occurred. Note that in the DIMACS file the vertices are numbered from 1 to N while in the bliss C API they are from 0 to N-1. Thus the vertex n in the file corresponds to the vertex n-1 in the API.

9.1.2.11 bliss_release()

```
void bliss_release (
            BlissGraph * graph )
```

Release the graph. Note that the memory pointed by the arguments of hook functions for bliss_find_automorphisms() and bliss_find_canonical_labeling() is deallocated and thus should not be accessed after calling this function.

9.1.2.12 bliss_write_dimacs()

```
void bliss_write_dimacs (
            BlissGraph * graph,
            FILE * fp )
```

Output the graph in the file stream *fp* in the DIMACS format. See the User's Guide for the file format details. Note that in the DIMACS file the vertices are numbered from 1 to N while in bliss they are from 0 to N-1.

9.1.2.13 bliss_write_dot()

```
void bliss_write_dot (
            BlissGraph * graph,
            FILE * fp )
```

Print the graph in graphviz dot format.

9.2 utils.hh File Reference

Some small utilities.

```
#include <cstdio>
```

Namespaces

- bliss

Functions

- void bliss::print_permutation (FILE *const fp, const unsigned int N, const unsigned int *perm, const unsigned int offset)
- void bliss::print_permutation (FILE *const fp, const std::vector< unsigned int > &perm, const unsigned int offset)
- bool bliss::is_permutation (const unsigned int N, const unsigned int *perm)
- bool bliss::is_permutation (const std::vector< unsigned int > &perm)

9.2.1 Detailed Description

Some small utilities.

Index