# The bliss C++ API

Generated by Doxygen 1.9.4

# Chapter 1

# Outline

This is the C++ API documentation of bliss, produced by running `doxygen` in the source directory.

The algorithms and data structures used in bliss, the graph file format, as well as the compilation process can be found at the `bliss web site`.

The C++ language API is the main API to bliss. It basically consists of the public methods in the classes

- bliss::Graph and
- bliss::Digraph.

For an example of its use, see the source of the bliss executable.

## 1.1 The C language API

The C language API is given in the file bliss_C.h. It is currently only a subset of the C++ API, so consider using the C++ API whenever possible.

# Chapter 2

# The bliss executable

```
/*
Copyright (c) 2003-2021 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/>.
*/
#include <functional>
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <cassert>
#include <bliss/defs.hh>
#include <bliss/timer.hh>
#include <bliss/utils.hh>
#include <bliss/graph.hh>
#include <bliss/digraph.hh>
/* 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 = strrchr(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];
        }
    }
    }
}
```

```
/* Output an error message and exit the whole program with the exit value 1.  */
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);
  auto report_aut = [&](const unsigned int n, const unsigned int* aut) -> void {
    fprintf(stdout, "Generator:  ");
    bliss::print_permutation(stdout, n, aut, 1);
    fprintf(stdout, "\n");
  };

  if(opt_canonize == false)
    {
      /* No canonical labeling, only automorphism group */
      g->find_automorphisms(stats, report_aut);
    }
  else
    {
      /* Canonical labeling and automorphism group */
      const unsigned int* cl = g->canonical_form(stats, report_aut);
      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 simple wrapper class for non-negative 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 Heap

  *A min-heap of unsigned integers.*
- class KQueue

  *A simple implementation of queues with fixed maximum capacity.*
- class Orbit

  *A class for representing orbit information.*
- class Partition

  *A class for refinable, backtrackable ordered partitions.*
- class Stats

  *Statistics returned by the bliss search algorithm.*
- class Timer

  *A simple helper class for measuring elapsed time.*
- class **TreeNode**
- class UintSeqHash

  *A updatable hash for sequences of unsigned ints.*

### Functions

- size_t print_permutation (FILE *const fp, const unsigned int N, const unsigned int *perm, const unsigned int offset)
- size_t 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)

## Variables

- static const char ∗const **version** = "0.77"

  *The version number of bliss.*

### 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 std::vector< unsigned int > & perm )
```

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

#### 7.1.2.2 is_permutation() [2/2]

```
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.

#### 7.1.2.3 print_permutation() [1/2]

```
size_t 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.

#### 7.1.2.4 print_permutation() [2/2]

```
size_t 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.

# Chapter 8

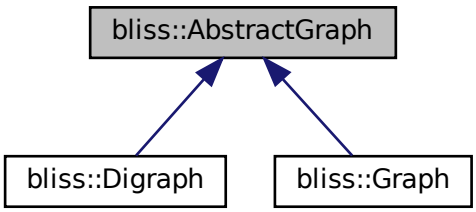
# Class Documentation

## 8.1 bliss::AbstractGraph Class Reference

An abstract base class for different types of graphs.

`#include <abstractgraph.hh>`

Inheritance diagram for bliss::AbstractGraph:




### Classes

- class **CR_CEP**
- struct **PathInfo**

### 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 unsigned int get_color (const unsigned int vertex) const =0
- virtual void change_color (const unsigned int vertex, const unsigned int color)=0

- 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
- virtual AbstractGraph * permute (const std::vector< unsigned int > &perm) const =0
- virtual bool is_automorphism (unsigned int *const perm) const =0
- virtual bool is_automorphism (const std::vector< unsigned int > &perm) const =0
- void find_automorphisms (Stats &stats, const std::function< void(unsigned int n, const unsigned int *aut)> &report=nullptr, const std::function< bool()> &terminate=nullptr)
- const unsigned int * canonical_form (Stats &stats, const std::function< void(unsigned int n, const unsigned int *aut)> &report=nullptr, const std::function< bool()> &terminate=nullptr)
- 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)

### 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,
            const std::function< void(unsigned int n, const unsigned int *aut)> & report =
nullptr,
            const std::function< bool()> & terminate = nullptr )
```

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()).

If the *terminate* function argument is given, it is called in each search tree node: if the function returns true, then the search is terminated and thus (i) not all the automorphisms may have been generated and (ii) the returned labeling may not be canonical. The *terminate* function may be used to limit the time spent in bliss in case the graph is too difficult under the available time constraints. If used, keep the function simple to evaluate so that it does not consume too much time.

#### 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,
            const std::function< void(unsigned int n, const unsigned int *aut)> & report =
nullptr,
            const std::function< bool()> & terminate = nullptr )
```

Find a set of generators for the automorphism group of the graph. The function *report* (if non-null) is called each time a new generator for the automorphism group is found. The first argument *n* for the function is the length of the automorphism (equal to get_nof_vertices()), and the second 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 *report* function; if you want to use the automorphism later, you have to take a copy of it. Do not call any member functions from the *report* function.

The search statistics are copied in *stats*.

If the *terminate* function argument is given, it is called in each search tree node: if the function returns true, then the search is terminated and thus not all the automorphisms may have been generated. The *terminate* function may be used to limit the time spent in bliss in case the graph is too difficult under the available time constraints. If used, keep the function simple to evaluate so that it does not consume too much time.

#### 8.1.2.6 get_color()

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

Get the color of a vertex.

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

#### 8.1.2.7 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.

#### 8.1.2.8 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.

#### 8.1.2.9 is_automorphism() [1/2]

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

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

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

#### 8.1.2.10 is_automorphism() [2/2]

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

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

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

#### 8.1.2.11 permute() [1/2]

```
virtual AbstractGraph * bliss::AbstractGraph::permute (
            const std::vector< unsigned int > & 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.

#### 8.1.2.12 permute() [2/2]

```
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.

#### 8.1.2.13 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 |

#### 8.1.2.14 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 |

#### 8.1.2.15 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 |

#### 8.1.2.16 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 |

#### 8.1.2.17 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 |

#### 8.1.2.18 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.

#### 8.1.2.19 write_dot() [1/2]

```
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.

#### 8.1.2.20 write_dot() [2/2]

```
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.

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

- src/abstractgraph.hh
- src/abstractgraph.cc

## 8.2 bliss::BigNum Class Reference

A simple wrapper class for non-negative big integers (or approximation of them).

`#include <bignum.hh>`

### Public Member Functions

- BigNum ()
- ~BigNum ()
- void assign (unsigned int n)
- void multiply (unsigned int n)
- size_t print (FILE *const fp) const
- const std::vector< unsigned int > & get_factors () const
- std::string to_string () const

### 8.2.1 Detailed Description

A simple wrapper class for non-negative 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, if the compile time flag BLISS_BIGNUM_APPROX is set, a "long double" is used to approximate a big integer. Otherwise, by default, a big integer is represented by a product of integer-sized factors.

### 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.

#### 8.2.2.2 ~BigNum()

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

Destroy the number.

### 8.2.3 Member Function Documentation

#### 8.2.3.1 assign()

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

Set the number to *n*.

#### 8.2.3.2 get_factors()

```
const std::vector< unsigned int > & bliss::BigNum::get_factors ( ) const  [inline]
```

Get a reference to the factors vector.

#### 8.2.3.3 multiply()

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

Multiply the number with *n*.

#### 8.2.3.4 print()

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

Print the number in the file stream *fp*. In the current version, the returned number of characters printed, is incorrect (either -1 or 0).

#### 8.2.3.5 to_string()

```
std::string bliss::BigNum::to_string ( ) const  [inline]
```

Get the tring representation of the number. Unoptimized, uses an elementary school algorithm to multiply the factors.

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

- src/bignum.hh

## 8.3 bliss_graph_struct Struct Reference

The true bliss graph is hiding in this struct.

### 8.3.1 Detailed Description

The true bliss graph is hiding in this struct.

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

- src/bliss_C.cc

## 8.4 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**

  *An approximation (due to possible rounding errors) of the size of the automorphism group.*
- long unsigned int **nof_nodes**

  *The number of nodes in the search tree.*
- long unsigned int **nof_leaf_nodes**

  *The number of leaf nodes in the search tree.*
- long unsigned int **nof_bad_nodes**

  *The number of bad nodes in the search tree.*
- long unsigned int **nof_canupdates**

  *The number of canonical representative updates.*
- long unsigned int **nof_generators**

  *The number of generator permutations.*
- unsigned long int **max_level**

  *The maximal depth of the search tree.*

### 8.4.1 Detailed Description

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

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

- src/bliss_C.h

## 8.5 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

### 8.5.1 Detailed Description

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

### 8.5.2 Member Function Documentation

#### 8.5.2.1 is_in_splitting_queue()

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

Is this cell in splitting queue?

#### 8.5.2.2 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:

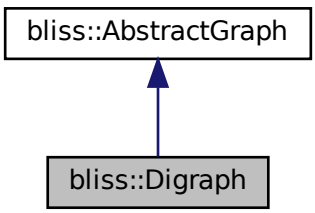
- src/partition.hh

## 8.6 bliss::Digraph Class Reference

The class for directed, vertex colored graphs.

`#include <digraph.hh>`

Inheritance diagram for bliss::Digraph:



## Classes

- class **Vertex**

## 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)
- 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)
- unsigned int get_color (const unsigned int vertex) const
- void change_color (const unsigned int vertex, const unsigned int color)
- Digraph * copy () const
- int cmp (Digraph &other)
- void set_splitting_heuristic (SplittingHeuristic shs)
- Digraph * permute (const unsigned int *const perm) const
- Digraph * permute (const std::vector< unsigned int > &perm) const
- bool is_automorphism (unsigned int *const perm) const
- bool is_automorphism (const std::vector< unsigned int > &perm) const
- void set_verbose_level (const unsigned int level)
- void set_verbose_file (FILE *const fp)
- void set_failure_recording (const bool active)
- void set_component_recursion (const bool active)
- void find_automorphisms (Stats &stats, const std::function< void(unsigned int n, const unsigned int *aut)> &report=nullptr, const std::function< bool()> &terminate=nullptr)
- const unsigned int * canonical_form (Stats &stats, const std::function< void(unsigned int n, const unsigned int *aut)> &report=nullptr, const std::function< bool()> &terminate=nullptr)
- void set_long_prune_activity (const bool active)

## Static Public Member Functions

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

### 8.6.1 Detailed Description

The class for directed, vertex colored graphs.

Multiple edges between vertices are not allowed (copies will be ignored).

## 8.6.2 Member Enumeration Documentation

### 8.6.2.1 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. |

## 8.6.3 Constructor & Destructor Documentation

### 8.6.3.1 Digraph()

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

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

### 8.6.3.2 ~Digraph()

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

Destroy the graph.

## 8.6.4 Member Function Documentation

#### 8.6.4.1 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.

#### 8.6.4.2 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.

#### 8.6.4.3 canonical_form()

```
const unsigned int * bliss::AbstractGraph::canonical_form (
            Stats & stats,
            const std::function< void(unsigned int n, const unsigned int *aut)> & report =
nullptr,
            const std::function< bool()> & terminate = nullptr )  [inherited]
```

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()).

If the *terminate* function argument is given, it is called in each search tree node: if the function returns true, then the search is terminated and thus (i) not all the automorphisms may have been generated and (ii) the returned labeling may not be canonical. The *terminate* function may be used to limit the time spent in bliss in case the graph is too difficult under the available time constraints. If used, keep the function simple to evaluate so that it does not consume too much time.

#### 8.6.4.4 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.

#### 8.6.4.5 cmp()

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

Compare this graph to the *other* graph in a total orger on graphs.

**Returns**

0 if the graphs are equal, -1 if this graph is "smaller than" the other, and 1 if this graph is "greater than" the other.

#### 8.6.4.6 copy()

```
Digraph * bliss::Digraph::copy ( ) const
```

Get a copy of the graph.

#### 8.6.4.7 find_automorphisms()

```
void bliss::AbstractGraph::find_automorphisms (
            Stats & stats,
            const std::function< void(unsigned int n, const unsigned int *aut)> & report =
nullptr,
            const std::function< bool()> & terminate = nullptr )  [inherited]
```

Find a set of generators for the automorphism group of the graph. The function *report* (if non-null) is called each time a new generator for the automorphism group is found. The first argument *n* for the function is the length of the automorphism (equal to get_nof_vertices()), and the second 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 *report* function; if you want to use the automorphism later, you have to take a copy of it. Do not call any member functions from the *report* function.

The search statistics are copied in *stats*.

If the *terminate* function argument is given, it is called in each search tree node: if the function returns true, then the search is terminated and thus not all the automorphisms may have been generated. The *terminate* function may be used to limit the time spent in bliss in case the graph is too difficult under the available time constraints. If used, keep the function simple to evaluate so that it does not consume too much time.

#### 8.6.4.8 get_color()

```
unsigned int bliss::Digraph::get_color (
            const unsigned int vertex ) const  [virtual]
```

Get the color of a vertex.

Implements bliss::AbstractGraph.

#### 8.6.4.9 get_hash()

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

Get a hash value for the graph.

**Returns**

the hash value

Implements bliss::AbstractGraph.

#### 8.6.4.10 get_nof_vertices()

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

Return the number of vertices in the graph.

Implements bliss::AbstractGraph.

#### 8.6.4.11 is_automorphism() [1/2]

```
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.

Implements bliss::AbstractGraph.

#### 8.6.4.12 is_automorphism() [2/2]

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

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

Implements bliss::AbstractGraph.

#### 8.6.4.13 permute() [1/2]

```
Digraph * bliss::Digraph::permute (
            const std::vector< unsigned int > & 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.

#### 8.6.4.14 permute() [2/2]

```
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.

#### 8.6.4.15 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

#### 8.6.4.16 set_component_recursion()

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

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 |

#### 8.6.4.17 set_failure_recording()

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

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 |

#### 8.6.4.18 set_long_prune_activity()

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

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 |

#### 8.6.4.19 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.

#### 8.6.4.20 set_verbose_file()

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

Set the file stream for the verbose output.

**Parameters**

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

#### 8.6.4.21 set_verbose_level()

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

Set the verbose output level for the algorithms.

**Parameters**

| | |
|---|---|
| *level* | the level of verbose output, 0 means no verbose output |

#### 8.6.4.22 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.

#### 8.6.4.23 write_dot() [1/2]

```
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.

#### 8.6.4.24 write_dot() [2/2]

```
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.

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

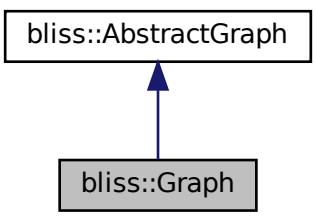
- src/digraph.hh
- src/digraph.cc

## 8.7 bliss::Graph Class Reference

The class for undirected, vertex colored graphs.

```
#include <graph.hh>
```

Inheritance diagram for bliss::Graph:



## Classes

- class **Vertex**

## 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)
- virtual unsigned int get_hash ()
- unsigned int get_nof_vertices () const
- Graph * permute (const unsigned int *const perm) const
- Graph * permute (const std::vector< unsigned int > &perm) const
- bool is_automorphism (unsigned int *const perm) const
- bool is_automorphism (const std::vector< unsigned int > &perm) const
- unsigned int add_vertex (const unsigned int color=0)
- void add_edge (const unsigned int v1, const unsigned int v2)
- unsigned int get_color (const unsigned int vertex) const
- void change_color (const unsigned int vertex, const unsigned int color)
- Graph * copy () const
- int cmp (Graph &other)
- void set_splitting_heuristic (const SplittingHeuristic shs)
- void set_verbose_level (const unsigned int level)
- void set_verbose_file (FILE *const fp)
- void set_failure_recording (const bool active)
- void set_component_recursion (const bool active)
- void find_automorphisms (Stats &stats, const std::function< void(unsigned int n, const unsigned int *aut)> &report=nullptr, const std::function< bool()> &terminate=nullptr)
- const unsigned int * canonical_form (Stats &stats, const std::function< void(unsigned int n, const unsigned int *aut)> &report=nullptr, const std::function< bool()> &terminate=nullptr)
- void set_long_prune_activity (const bool active)

## Static Public Member Functions

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

### 8.7.1 Detailed Description

The class for undirected, vertex colored graphs.

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

## 8.7.2 Member Enumeration Documentation

### 8.7.2.1 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. |

## 8.7.3 Constructor & Destructor Documentation

### 8.7.3.1 Graph()

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

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

### 8.7.3.2 ~Graph()

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

Destroy the graph.

## 8.7.4 Member Function Documentation

#### 8.7.4.1 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.

#### 8.7.4.2 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.

#### 8.7.4.3 canonical_form()

```
const unsigned int * bliss::AbstractGraph::canonical_form (
            Stats & stats,
            const std::function< void(unsigned int n, const unsigned int *aut)> & report =
nullptr,
            const std::function< bool()> & terminate = nullptr )  [inherited]
```

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()).

If the *terminate* function argument is given, it is called in each search tree node: if the function returns true, then the search is terminated and thus (i) not all the automorphisms may have been generated and (ii) the returned labeling may not be canonical. The *terminate* function may be used to limit the time spent in bliss in case the graph is too difficult under the available time constraints. If used, keep the function simple to evaluate so that it does not consume too much time.

#### 8.7.4.4 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.

#### 8.7.4.5 cmp()

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

Compare this graph to the *other* graph in a total orger on graphs.

**Returns**

0 if the graphs are equal, -1 if this graph is "smaller than" the other, and 1 if this graph is "greater than" the other.

#### 8.7.4.6 copy()

```
Graph * bliss::Graph::copy ( ) const
```

Get a copy of the graph.

#### 8.7.4.7 find_automorphisms()

```
void bliss::AbstractGraph::find_automorphisms (
            Stats & stats,
            const std::function< void(unsigned int n, const unsigned int *aut)> & report =
nullptr,
            const std::function< bool()> & terminate = nullptr )  [inherited]
```

Find a set of generators for the automorphism group of the graph. The function *report* (if non-null) is called each time a new generator for the automorphism group is found. The first argument *n* for the function is the length of the automorphism (equal to get_nof_vertices()), and the second 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 *report* function; if you want to use the automorphism later, you have to take a copy of it. Do not call any member functions from the *report* function.

The search statistics are copied in *stats*.

If the *terminate* function argument is given, it is called in each search tree node: if the function returns true, then the search is terminated and thus not all the automorphisms may have been generated. The *terminate* function may be used to limit the time spent in bliss in case the graph is too difficult under the available time constraints. If used, keep the function simple to evaluate so that it does not consume too much time.

#### 8.7.4.8 get_color()

```
unsigned int bliss::Graph::get_color (
            const unsigned int vertex ) const  [virtual]
```

Get the color of a vertex.

Implements bliss::AbstractGraph.

#### 8.7.4.9 get_hash()

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

Get a hash value for the graph.

**Returns**

the hash value

Implements bliss::AbstractGraph.

#### 8.7.4.10 get_nof_vertices()

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

Return the number of vertices in the graph.

Implements bliss::AbstractGraph.

#### 8.7.4.11 is_automorphism() [1/2]

```
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.

Implements bliss::AbstractGraph.

#### 8.7.4.12 is_automorphism() [2/2]

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

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

Implements bliss::AbstractGraph.

#### 8.7.4.13 permute() [1/2]

```
Graph * bliss::Graph::permute (
            const std::vector< unsigned int > & 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.

#### 8.7.4.14 permute() [2/2]

```
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.

#### 8.7.4.15 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

#### 8.7.4.16 set_component_recursion()

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

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 |

#### 8.7.4.17 set_failure_recording()

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

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 |

#### 8.7.4.18 set_long_prune_activity()

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

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 |

#### 8.7.4.19 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.

#### 8.7.4.20 set_verbose_file()

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

Set the file stream for the verbose output.

**Parameters**

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

#### 8.7.4.21 set_verbose_level()

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

Set the verbose output level for the algorithms.

**Parameters**

| | |
|---|---|
| *level* | the level of verbose output, 0 means no verbose output |

#### 8.7.4.22 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.

#### 8.7.4.23 write_dot() [1/2]

```
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.

#### 8.7.4.24 write_dot() [2/2]

```
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.

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

- src/graph.hh
- src/graph.cc

## 8.8 bliss::Heap Class Reference

A min-heap of unsigned integers.

```
#include <heap.hh>
```

### Public Member Functions

- bool is_empty () const
- void clear ()
- void insert (const unsigned int e)
- unsigned int smallest () const
- unsigned int remove ()
- size_t size () const

### 8.8.1 Detailed Description

A min-heap of unsigned integers.

### 8.8.2 Member Function Documentation

#### 8.8.2.1 clear()

```
void bliss::Heap::clear ( )  [inline]
```

Remove all the elements in the heap. Time complexity is O(1).

#### 8.8.2.2 insert()

```
void bliss::Heap::insert (
            const unsigned int e )  [inline]
```

Insert the element *e* in the heap. Time complexity is O(log(N)), where N is the number of elements currently in the heap.

#### 8.8.2.3 is_empty()

```
bool bliss::Heap::is_empty ( ) const  [inline]
```

Is the heap empty? Time complexity is O(1).

#### 8.8.2.4 remove()

```
unsigned int bliss::Heap::remove ( )  [inline]
```

Remove and return the smallest element in the heap. Time complexity is O(log(N)), where N is the number of elements currently in the heap.

#### 8.8.2.5 size()

```
size_t bliss::Heap::size ( ) const  [inline]
```

Get the number of elements in the heap.

#### 8.8.2.6 smallest()

```
unsigned int bliss::Heap::smallest ( ) const  [inline]
```

Return the smallest element in the heap. Time complexity is O(1).

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

- src/heap.hh

## 8.9 bliss::KQueue< Type > Class Template Reference

A simple implementation of queues with fixed maximum capacity.

```
#include <kqueue.hh>
```

## Public Member Functions

- KQueue ()
- void init (const unsigned int N)
- bool is_empty () const
- unsigned int size () const
- void clear ()
- Type front () const
- Type pop_front ()
- void push_front (Type e)
- Type pop_back ()
- void push_back (Type e)

### 8.9.1 Detailed Description

**template<class Type>**
**class bliss::KQueue< Type >**

A simple implementation of queues with fixed maximum capacity.

### 8.9.2 Constructor & Destructor Documentation

#### 8.9.2.1 KQueue()

```
template<class Type >
bliss::KQueue< Type >::KQueue
```

Create a new queue with capacity zero. The function init() should be called next.

### 8.9.3 Member Function Documentation

#### 8.9.3.1 clear()

```
template<class Type >
void bliss::KQueue< Type >::clear
```

Remove all the elements in the queue.

#### 8.9.3.2 front()

```
template<class Type >
Type bliss::KQueue< Type >::front
```

Return (but don't remove) the first element in the queue.

#### 8.9.3.3 init()

```
template<class Type >
void bliss::KQueue< Type >::init (
            const unsigned int N )
```

Initialize the queue to have the capacity to hold at most *N* elements.

#### 8.9.3.4 is_empty()

```
template<class Type >
bool bliss::KQueue< Type >::is_empty
```

Is the queue empty?

#### 8.9.3.5 pop_back()

```
template<class Type >
Type bliss::KQueue< Type >::pop_back ( )
```

Remove and return the last element of the queue.

#### 8.9.3.6 pop_front()

```
template<class Type >
Type bliss::KQueue< Type >::pop_front
```

Remove and return the first element of the queue.

#### 8.9.3.7 push_back()

```
template<class Type >
void bliss::KQueue< Type >::push_back (
            Type e )
```

Push the element *e* in the back of the queue.

#### 8.9.3.8 push_front()

```
template<class Type >
void bliss::KQueue< Type >::push_front (
            Type e )
```

Push the element *e* in the front of the queue.

#### 8.9.3.9 size()

```
template<class Type >
unsigned int bliss::KQueue< Type >::size
```

Return the number of elements in the queue.

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

- src/kqueue.hh

## 8.10 bliss::Orbit Class Reference

A class for representing orbit information.

```
#include <orbit.hh>
```

### Public Member Functions

- Orbit ()
- void init (const unsigned int N)
- void reset ()
- void merge_orbits (unsigned int e1, unsigned int e2)
- bool is_minimal_representative (unsigned int e) const
- unsigned int get_minimal_representative (unsigned int e) const
- unsigned int orbit_size (unsigned int e) const
- unsigned int nof_orbits () const

### 8.10.1 Detailed Description

A class for representing orbit information.

Given a set {0,...,N-1} of N elements, represent equivalence classes (that is, unordered partitions) of the elements. Supports only equivalence class merging, not splitting. Merging two classes requires time O(k), where k is the number of the elements in the smaller of the merged classes. Getting the smallest representative in a class (and thus testing whether two elements belong to the same class) is a constant time operation.

### 8.10.2 Constructor & Destructor Documentation

#### 8.10.2.1 Orbit()

```
bliss::Orbit::Orbit ( )
```

Create a new orbit information object. The init() function must be called next to actually initialize the object.

### 8.10.3 Member Function Documentation

#### 8.10.3.1 get_minimal_representative()

```
unsigned int bliss::Orbit::get_minimal_representative (
            unsigned int e ) const
```

Get the smallest element in the orbit of the element *e*. Time complexity is O(1).

#### 8.10.3.2 init()

```
void bliss::Orbit::init (
            const unsigned int N )
```

Initialize the orbit information to consider sets of *N* elements. It is required that $N > 0$. The orbit information is reset so that each element forms an orbit of its own. Time complexity is O(N).

**See also**

reset()

#### 8.10.3.3 is_minimal_representative()

```
bool bliss::Orbit::is_minimal_representative (
            unsigned int e ) const
```

Is the element *e* the smallest element in its orbit? Time complexity is O(1).

#### 8.10.3.4 merge_orbits()

```
void bliss::Orbit::merge_orbits (
            unsigned int e1,
            unsigned int e2 )
```

Merge the orbits of the elements *e1* and *e2*. Time complexity is O(k), where k is the number of elements in the smaller of the merged orbits.

#### 8.10.3.5 nof_orbits()

```
unsigned int bliss::Orbit::nof_orbits ( ) const  [inline]
```

Get the number of orbits. Time complexity is O(1).

#### 8.10.3.6 orbit_size()

```
unsigned int bliss::Orbit::orbit_size (
            unsigned int e ) const
```

Get the number of elements in the orbit of the element *e*. Time complexity is O(1).

#### 8.10.3.7 reset()

```
void bliss::Orbit::reset ( )
```

Reset the orbits so that each element forms an orbit of its own. Time complexity is O(N).

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

- src/orbit.hh
- src/orbit.cc

## 8.11 bliss::Partition Class Reference

A class for refinable, backtrackable ordered partitions.

```
#include <partition.hh>
```

### Classes

- class Cell

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

### Public Types

- typedef unsigned int BacktrackPoint

### Public Member Functions

- BacktrackPoint set_backtrack_point ()
- void goto_backtrack_point (BacktrackPoint p)
- Cell * individualize (Cell *const cell, const unsigned int element)
- Cell * get_cell (const unsigned int e) const
- void init (const unsigned int N)
- bool is_discrete () const
- size_t print (FILE *const fp, const bool add_newline=true) const
- size_t print_signature (FILE *const fp, const bool add_newline=true) const
- void clear_ivs (Cell *const cell)

### 8.11.1 Detailed Description

A class for refinable, backtrackable ordered partitions.

This is rather a data structure with some helper functions than a proper self-contained class. That is, for efficiency reasons the fields of this class are directly manipulated from bliss::AbstractGraph and its subclasses. Conversely, some methods of this class modify the fields of bliss::AbstractGraph, too.

### 8.11.2 Member Typedef Documentation

#### 8.11.2.1 BacktrackPoint

```
typedef unsigned int bliss::Partition::BacktrackPoint
```

Type for backtracking points.

### 8.11.3 Member Function Documentation

#### 8.11.3.1 clear_ivs()

```
void bliss::Partition::clear_ivs (
            Cell *const cell )
```

Clear the invariant_values of the elements in the Cell *cell*.

#### 8.11.3.2 get_cell()

```
Cell * bliss::Partition::get_cell (
            const unsigned int e ) const  [inline]
```

Get the cell of the element *e*

#### 8.11.3.3 goto_backtrack_point()

```
void bliss::Partition::goto_backtrack_point (
            BacktrackPoint p )
```

Backtrack to the point *p* and remove it.

#### 8.11.3.4 individualize()

```
Partition::Cell * bliss::Partition::individualize (
            Partition::Cell *const cell,
            const unsigned int element )
```

Split the non-unit Cell *cell* = {*element*,e1,e2,...,en} containing the element *element* in two: *cell* = {e1,...,en} and *newcell* = {*element*}.

**Parameters**

| | |
|---|---|
| *cell* | a non-unit Cell |
| *element* | an element in *cell* |

**Returns**

the new unit Cell *newcell*

#### 8.11.3.5 init()

```
void bliss::Partition::init (
            const unsigned int N )
```

Initialize the partition to the unit partition (all elements in one cell) over the $N > 0$ elements {0,...,*N-1}*.

#### 8.11.3.6 is_discrete()

```
bool bliss::Partition::is_discrete ( ) const  [inline]
```

Returns true iff the partition is discrete, meaning that all the elements are in their own cells.

#### 8.11.3.7 print()

```
size_t bliss::Partition::print (
            FILE *const fp,
            const bool add_newline = true ) const
```

Print the partition into the file stream *fp*.

#### 8.11.3.8 print_signature()

```
size_t bliss::Partition::print_signature (
            FILE *const fp,
            const bool add_newline = true ) const
```

Print the partition cell sizes into the file stream *fp*.

#### 8.11.3.9 set_backtrack_point()

```
Partition::BacktrackPoint bliss::Partition::set_backtrack_point ( )
```

Get a new backtrack point for the current partition

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

- src/partition.hh
- src/partition.cc

## 8.12 bliss::Stats Class Reference

Statistics returned by the bliss search algorithm.

`#include <stats.hh>`

### Public Member Functions

- size_t print (FILE *const fp) const
- const BigNum & get_group_size () 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

### 8.12.1 Detailed Description

Statistics returned by the bliss search algorithm.

### 8.12.2 Member Function Documentation

#### 8.12.2.1 get_group_size()

```
const BigNum & bliss::Stats::get_group_size ( ) const  [inline]
```

The size of the automorphism group.

#### 8.12.2.2 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.

#### 8.12.2.3 get_max_level()

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

The maximal depth of the search tree.

#### 8.12.2.4 get_nof_bad_nodes()

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

The number of bad nodes in the search tree.

#### 8.12.2.5 get_nof_canupdates()

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

The number of canonical representative updates.

#### 8.12.2.6 get_nof_generators()

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

The number of generator permutations.

#### 8.12.2.7 get_nof_leaf_nodes()

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

The number of leaf nodes in the search tree.

#### 8.12.2.8 get_nof_nodes()

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

The number of nodes in the search tree.

#### 8.12.2.9 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:

- src/stats.hh

## 8.13 bliss::Timer Class Reference

A simple helper class for measuring elapsed time.

```
#include <timer.hh>
```

### Public Member Functions

- **Timer** ()

  *Create and start a new timer.*
- void **reset** ()

  *Reset the timer.*
- double **get_duration** () const

  *Get the time (in seconds) elapsed since the creation or the last reset() call of the timer.*

### 8.13.1 Detailed Description

A simple helper class for measuring elapsed time.

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

- src/timer.hh

## 8.14 bliss::UintSeqHash Class Reference

A updatable hash for sequences of unsigned ints.

`#include <uintseqhash.hh>`

### Public Member Functions

- void reset ()
- void update (unsigned int n)
- unsigned int get_value () const
- int cmp (const UintSeqHash &other) const
- bool is_lt (const UintSeqHash &other) const
- bool is_le (const UintSeqHash &other) const
- bool is_equal (const UintSeqHash &other) const

### 8.14.1 Detailed Description

A updatable hash for sequences of unsigned ints.

### 8.14.2 Member Function Documentation

#### 8.14.2.1 cmp()

```
int bliss::UintSeqHash::cmp (
            const UintSeqHash & other ) const  [inline]
```

Compare the hash values of this and *other*. Return -1/0/1 if the value of this is smaller/equal/greater than that of *other*.

#### 8.14.2.2 get_value()

```
unsigned int bliss::UintSeqHash::get_value ( ) const  [inline]
```

Get the hash value of the sequence seen so far.

#### 8.14.2.3 is_equal()

```
bool bliss::UintSeqHash::is_equal (
            const UintSeqHash & other ) const  [inline]
```

An abbreviation for cmp(other) == 0

#### 8.14.2.4 is_le()

```
bool bliss::UintSeqHash::is_le (
            const UintSeqHash & other ) const  [inline]
```

An abbreviation for cmp(other) <= 0

#### 8.14.2.5 is_lt()

```
bool bliss::UintSeqHash::is_lt (
            const UintSeqHash & other ) const  [inline]
```

An abbreviation for cmp(other) < 0

#### 8.14.2.6 reset()

```
void bliss::UintSeqHash::reset ( )  [inline]
```

Reset the hash value.

#### 8.14.2.7 update()

```
void bliss::UintSeqHash::update (
            unsigned int n )
```

Add the unsigned int *n* to the sequence.

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

- src/uintseqhash.hh
- src/uintseqhash.cc

# Chapter 9

# File Documentation

## 9.1 abstractgraph.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
27 namespace bliss {
28   class AbstractGraph;
29 }
30
31 #include <cstdio>
32 #include <functional>
33 #include <vector>
34 #include <bliss/stats.hh>
35 #include <bliss/kqueue.hh>
36 #include <bliss/heap.hh>
37 #include <bliss/orbit.hh>
38 #include <bliss/partition.hh>
39 #include <bliss/uintseqhash.hh>
40
41 namespace bliss {
42
43
47 class AbstractGraph
48 {
49   friend class Partition;
50
51 public:
52   AbstractGraph();
53   virtual ~AbstractGraph();
54
59   void set_verbose_level(const unsigned int level);
60
65   void set_verbose_file(FILE * const fp);
66
70   virtual unsigned int add_vertex(const unsigned int color = 0) = 0;
71
78   virtual void add_edge(const unsigned int source, const unsigned int target) = 0;
79
83   virtual unsigned int get_color(const unsigned int vertex) const = 0;
84
88   virtual void change_color(const unsigned int vertex, const unsigned int color) = 0;
89
```

```
90
96   void set_failure_recording(const bool active) {
97     assert(not in_search);
98     opt_use_failure_recording = active;
99   }
100
110   void set_component_recursion(const bool active) {
111     assert(not in_search);
112     opt_use_comprec = active;
113   }
114
115
116
120   virtual unsigned int get_nof_vertices() const = 0;
121
128   virtual AbstractGraph* permute(const unsigned int* const perm) const = 0;
129
136   virtual AbstractGraph* permute(const std::vector<unsigned int>& perm) const = 0;
137
142   virtual bool is_automorphism(unsigned int* const perm) const = 0;
143
148   virtual bool is_automorphism(const std::vector<unsigned int>& perm) const = 0;
149
174   void find_automorphisms(Stats& stats,
175               const std::function<void(unsigned int n, const unsigned int* aut)>& report = nullptr,
176               const std::function<bool()>& terminate = nullptr);
177
203   const unsigned int* canonical_form(Stats& stats,
204                      const std::function<void(unsigned int n, const unsigned int* aut)>& report =
     nullptr,
205                      const std::function<bool()>& terminate = nullptr);
206
216   virtual void write_dimacs(FILE * const fp) = 0;
217
222   virtual void write_dot(FILE * const fp) = 0;
223
229   virtual void write_dot(const char * const file_name) = 0;
230
235   virtual unsigned int get_hash() = 0;
236
247   void set_long_prune_activity(const bool active) {
248     assert(not in_search);
249     opt_use_long_prune = active;
250   }
251
252
253
254 protected:
257   unsigned int verbose_level;
258
261   FILE *verbstr;
262
265   Partition p;
266
271   bool in_search;
272
276   bool opt_use_failure_recording;
277
278   /* The "tree-specific" invariant value for the point when current path
279  * got different from the first path */
280   unsigned int failure_recording_fp_deviation;
281
285   bool opt_use_comprec;
286
287
288   unsigned int refine_current_path_certificate_index;
289   bool refine_compare_certificate;
290   bool refine_equal_to_first;
291   unsigned int refine_first_path_subcertificate_end;
292   int refine_cmp_to_best;
293   unsigned int refine_best_path_subcertificate_end;
294
295   static const unsigned int CERT_SPLIT = 0; //UINT_MAX;
296   static const unsigned int CERT_EDGE  = 1; //UINT_MAX-1;
301   void cert_add(const unsigned int v1,
302         const unsigned int v2,
303         const unsigned int v3);
304
310   void cert_add_redundant(const unsigned int x,
311               const unsigned int y,
312               const unsigned int z);
313
317   bool opt_use_long_prune;
322   static const unsigned int long_prune_options_max_mem = 50;
327   static const unsigned int long_prune_options_max_stored_auts = 100;
328
329   unsigned int long_prune_max_stored_autss;
```

```
330   std::vector<std::vector<bool> *> long_prune_fixed;
331   std::vector<std::vector<bool> *> long_prune_mcrs;
332   std::vector<bool> long_prune_temp;
333   unsigned int long_prune_begin;
334   unsigned int long_prune_end;
338   void long_prune_init();
342   void long_prune_deallocate();
343   void long_prune_add_automorphism(const unsigned int *aut);
344   std::vector<bool>& long_prune_get_fixed(const unsigned int index);
345   std::vector<bool>& long_prune_allocget_fixed(const unsigned int index);
346   std::vector<bool>& long_prune_get_mcrs(const unsigned int index);
347   std::vector<bool>& long_prune_allocget_mcrs(const unsigned int index);
352   void long_prune_swap(const unsigned int i, const unsigned int j);
353
354   /*
355 * Data structures and routines for refining the partition p into equitable
356 */
357   Heap neighbour_heap;
358   virtual bool split_neighbourhood_of_unit_cell(Partition::Cell * const) = 0;
359   virtual bool split_neighbourhood_of_cell(Partition::Cell * const) = 0;
360   void refine_to_equitable();
361   void refine_to_equitable(Partition::Cell * const unit_cell);
362   void refine_to_equitable(Partition::Cell * const unit_cell1,
363                Partition::Cell * const unit_cell2);
364
365
371   bool do_refine_to_equitable();
372
373   unsigned int eqref_max_certificate_index;
377   bool compute_eqref_hash;
378   UintSeqHash eqref_hash;
379
380
385   virtual bool is_equitable() const = 0;
386
387   unsigned int *first_path_labeling;
388   unsigned int *first_path_labeling_inv;
389   Orbit         first_path_orbits;
390   unsigned int *first_path_automorphism;
391
392   unsigned int *best_path_labeling;
393   unsigned int *best_path_labeling_inv;
394   Orbit         best_path_orbits;
395   unsigned int *best_path_automorphism;
396
397   void update_labeling(unsigned int * const lab);
398   void update_labeling_and_its_inverse(unsigned int * const lab,
399                        unsigned int * const lab_inv);
400   void update_orbit_information(Orbit &o, const unsigned int *perm);
401
402   void reset_permutation(unsigned int *perm);
403
404   std::vector<unsigned int> certificate_current_path;
405   std::vector<unsigned int> certificate_first_path;
406   std::vector<unsigned int> certificate_best_path;
407
408   unsigned int certificate_index;
409   virtual void initialize_certificate() = 0;
410
411   /* Remove duplicates from seq.
412 * If m is the largest element in seq, them m < tmp.size() must hold.
413 * All entries in tmp must be false when called.
414 * Under that condition, all entries in tmp are false on exit as well.
415 */
416   static void remove_duplicates(std::vector<unsigned int>& seq, std::vector<bool>& tmp);
417
418   virtual void remove_duplicate_edges() = 0;
419   virtual void make_initial_equitable_partition() = 0;
420   virtual Partition::Cell* find_next_cell_to_be_splitted(Partition::Cell *cell) = 0;
421
422
427   typedef struct {
428     unsigned int splitting_element;
429     unsigned int certificate_index;
430     unsigned int subcertificate_length;
431     UintSeqHash eqref_hash;
432   } PathInfo;
433
434   void search(const bool canonical, Stats &stats,
435           const std::function<void(unsigned int n, const unsigned int* aut)>& report_function = nullptr,
436           const std::function<bool()>& terminate = nullptr);
437
438
439   /*
440 *
441 * Nonuniform component recursion (NUCR)
442 *
```

```
443 */
444
445   /* The currently traversed component */
446   unsigned int cr_level;
447
451   class CR_CEP {
452   public:
454     unsigned int creation_level;
457     unsigned int discrete_cell_limit;
459     unsigned int next_cr_level;
461     unsigned int next_cep_index;
462     bool first_checked;
463     bool best_checked;
464   };
468   std::vector<CR_CEP> cr_cep_stack;
469
481   virtual bool nucr_find_first_component(const unsigned int level) = 0;
482   virtual bool nucr_find_first_component(const unsigned int level,
483                       std::vector<unsigned int>& component,
484                       unsigned int& component_elements,
485                       Partition::Cell*& sh_return) = 0;
490   std::vector<unsigned int> cr_component;
494   unsigned int cr_component_elements;
495
496
497
498
499
500
501 };
502
503 } // namespace bliss
```

## 9.2 bignum.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #if defined(BLISS_USE_GMP)
23 #include <gmp.h>
24 #else
25 #include <vector>
26 #include <string>
27 #endif
28
29 #include <cstdlib>
30 #include <cstdio>
31 #include <bliss/defs.hh>
32
33
34 namespace bliss {
35
48 #if defined(BLISS_USE_GMP)
49
50 class BigNum
51 {
52   mpz_t v;
53 public:
57   BigNum() {mpz_init(v); }
58
62   ~BigNum() {mpz_clear(v); }
63
67   void assign(unsigned int n) {mpz_set_ui(v, n); }
68
72   void multiply(unsigned int n) {mpz_mul_ui(v, v, n); }
```

```
73
77   size_t print(FILE* const fp)const {return mpz_out_str(fp, 10, v); }
78
84   void get(mpz_t& result)const {mpz_set(result, v); }
85 };
86
87 #elif defined(BLISS_BIGNUM_APPROX)
88
89 class BigNum
90 {
91   long double v;
92 public:
96   BigNum():  v(0.0) {}
97
101   void assign(unsigned int n) {v = (long double)n; }
102
106   void multiply(unsigned int n) {v *= (long double)n; }
107
111   size_t print(FILE* const fp)const {return fprintf(fp, "%Lg", v); }
112 };
113
114 #else
115
116
117 class BigNum
118 {
119   /* This is a version that does not actually compute the number
120 * but rather only stores the factor integers.
121 */
122   std::vector<unsigned int> factors;
123 public:
127   BigNum() {
128     factors.push_back(0);
129   }
130
134   ~BigNum() {}
135
139   void assign(unsigned int n) {
140     factors.clear();
141     factors.push_back(n);
142   }
143
147   void multiply(unsigned int n) {
148     factors.push_back(n);
149   }
150
156   size_t print(FILE* const fp)const {
157     assert(not factors.empty());
158     size_t r = 0;
159     /*
160 const char* sep = "";
161 for(int v:  factors) {
162 r += fprintf(fp, "%s%d", sep, v);
163 sep = "*";
164 }
165 */
166     for(char d:  to_string())
167       r += fprintf(fp, "%c", d);
168     return r;
169   }
170
174   const std::vector<unsigned int>& get_factors()const {
175     return factors;
176   }
177
182   std::string to_string()const {
183     // Base 100 result, in reverse order
184     std::vector<unsigned int> result;
185     result.push_back(1);
186     for(unsigned int factor:  factors) {
187       std::vector<unsigned int> summand;
188       unsigned int offset = 0;
189       while(factor != 0) {
190     const unsigned int multiplier = factor % 100;
191     // Multiplication by a "digit"
192     std::vector<unsigned int> product;
193     for(unsigned int i = 0; i < offset; i++)
194       product.push_back(0);
195     unsigned int carry = 0;
196     for(unsigned int digit:  result) {
197       unsigned int v = digit * multiplier + carry;
198       product.push_back(v % 100);
199       carry = v / 100;
200     }
201     if(carry > 0)
202       product.push_back(carry);
203     // Addition
```

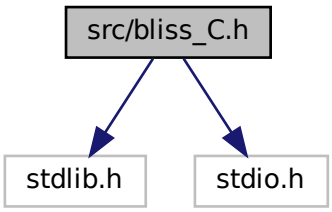
```
204     add(summand, product);
205     // Next "digit" in factor
206     factor = factor / 100;
207     offset++;
208       }
209       result = summand;
210     }
211     return _string(result);
212   }
213
214 protected:
215   static void add(std::vector<unsigned int>& num, const std::vector<unsigned int>& summand) {
216     unsigned int carry = 0;
217     unsigned int i = 0;
218     while(i < num.size() and i < summand.size()) {
219       const unsigned int v = carry + num[i] + summand[i];
220       num[i] = v % 100;
221       carry = v / 100;
222       i++;
223     }
224     while(i < summand.size()) {
225       const unsigned int v = carry + summand[i];
226       num.push_back(v % 100);
227       carry = v / 100;
228       i++;
229     }
230     while(i < num.size()) {
231       const unsigned int v = carry + num[i];
232       num[i] = v % 100;
233       carry = v / 100;
234       i++;
235     }
236     if(carry != 0)
237       num.push_back(carry);
238   }
239
240
241   static std::string _string(const std::vector<unsigned int> n) {
242     const char digits[] = {'0','1','2','3','4','5','6','7','8','9'};
243     std::string r;
244     bool first = true;
245     for(auto it = n.crbegin(); it != n.crend(); it++) {
246       unsigned int digit = *it;
247       unsigned int high = digit / 10;
248       if(not first or high > 0)
249     r.push_back(digits[high]);
250       first = false;
251       r.push_back(digits[digit % 10]);
252     }
253     return r;
254   }
255
256 };
257
258 #endif
259
260 } //namespace bliss
```

## 9.3 src/bliss_C.h File Reference

The bliss C API.

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

Include dependency graph for bliss_C.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)

### 9.3.1 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.

## 9.3.2 Function Documentation

### 9.3.2.1 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).

### 9.3.2.2 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].

### 9.3.2.3 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.

### 9.3.2.4 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.

#### 9.3.2.5 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.

#### 9.3.2.6 bliss_get_nof_vertices()

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

Return the number of vertices in the graph.

#### 9.3.2.7 bliss_hash()

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

Get a hash value for the graph.

#### 9.3.2.8 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.

#### 9.3.2.9 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}.

#### 9.3.2.10 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.

#### 9.3.2.11 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.

#### 9.3.2.12 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.

#### 9.3.2.13 bliss_write_dot()

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

Print the graph in graphviz dot format.

## 9.4 bliss_C.h

Go to the documentation of this file.

```
1 #ifndef BLISS_C_H
2 #define BLISS_C_H
3
4 /*
5 Copyright (c) 2003-2021 Tommi Junttila
6 Released under the GNU Lesser General Public License version 3.
7
8 This file is part of bliss.
9
10 bliss is free software:  you can redistribute it and/or modify
11 it under the terms of the GNU Lesser General Public License as published by
12 the Free Software Foundation, version 3 of the License.
13
14 bliss is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU Lesser General Public License for more details.
18
19 You should have received a copy of the GNU Lesser General Public License
20 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
```

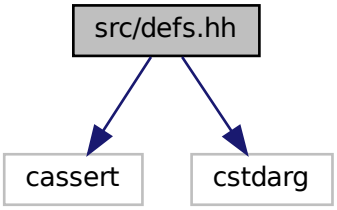
```
21 */
22
32 #include <stdlib.h>
33 #include <stdio.h>
34
35
39 typedef struct bliss_graph_struct BlissGraph;
40
41
46 typedef struct bliss_stats_struct
47 {
52   long double group_size_approx;
54   long unsigned int nof_nodes;
56   long unsigned int nof_leaf_nodes;
58   long unsigned int nof_bad_nodes;
60   long unsigned int nof_canupdates;
62   long unsigned int nof_generators;
64   unsigned long int max_level;
65 } BlissStats;
66
67
73 BlissGraph *bliss_new(const unsigned int N);
74
75
84 BlissGraph *bliss_read_dimacs(FILE *fp);
85
86
93 void bliss_write_dimacs(BlissGraph *graph, FILE *fp);
94
95
102 void bliss_release(BlissGraph *graph);
103
104
108 void bliss_write_dot(BlissGraph *graph, FILE *fp);
109
110
114 unsigned int bliss_get_nof_vertices(BlissGraph *graph);
115
116
122 unsigned int bliss_add_vertex(BlissGraph *graph, unsigned int c);
123
124
132 void bliss_add_edge(BlissGraph *graph, unsigned int v1, unsigned int v2);
133
134
143 int bliss_cmp(BlissGraph *graph1, BlissGraph *graph2);
144
145
149 unsigned int bliss_hash(BlissGraph *graph);
150
151
159 BlissGraph *bliss_permute(BlissGraph *graph, const unsigned int *perm);
160
161
177 void
178 bliss_find_automorphisms(BlissGraph *graph,
179              void (*hook)(void *user_param,
180                           unsigned int N,
181                           const unsigned int *aut),
182              void *hook_user_param,
183              BlissStats *stats);
184
185
197 const unsigned int *
198 bliss_find_canonical_labeling(BlissGraph *graph,
199                   void (*hook)(void *user_param,
200                                unsigned int N,
201                                const unsigned int *aut),
202                   void *hook_user_param,
203                   BlissStats *stats);
204
205 #endif
```

## 9.5 src/defs.hh File Reference

Some common definitions.

```
#include <cassert>
#include <cstdarg>
```

Include dependency graph for defs.hh:




### Namespaces

- namespace bliss

### Variables

- static const char ∗const **bliss::version** = "0.77"

  *The version number of bliss.*

### 9.5.1 Detailed Description

Some common definitions.

## 9.6 defs.hh

Go to the documentation of this file.

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #include <cassert>
23 #include <cstdarg>
24
29 #define BLISS_VERSION "0.77"
30 #define BLISS_VERSION_MAJOR 0
31 #define BLISS_VERSION_MINOR 77
```

```
32
33 namespace bliss {
34
36 static const char * const version = "0.77";
37
38
39
40 #if defined(BLISS_DEBUG)
41 #define BLISS_CONSISTENCY_CHECKS
42 #define BLISS_EXPENSIVE_CONSISTENCY_CHECKS
43 #endif
44
45
46 #if defined(BLISS_CONSISTENCY_CHECKS)
47 /* Force a check that the found automorphisms are valid */
48 #define BLISS_VERIFY_AUTOMORPHISMS
49 #endif
50
51
52 #if defined(BLISS_CONSISTENCY_CHECKS)
53 /* Force a check that the generated partitions are equitable */
54 #define BLISS_VERIFY_EQUITABLEDNESS
55 #endif
56
57 } // namespace bliss
58
59
```

# 9.7 digraph.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #include <bliss/abstractgraph.hh>
23
24 namespace bliss {
25
31 class Digraph :  public AbstractGraph
32 {
33 public:
42   typedef enum {
46     shs_f = 0,
49     shs_fs,
52     shs_fl,
56     shs_fm,
60     shs_fsm,
64     shs_flm
65   } SplittingHeuristic;
66
67 protected:
68   class Vertex {
69   public:
70     Vertex();
71     ~Vertex();
72     void add_edge_to(const unsigned int dest_vertex);
73     void add_edge_from(const unsigned int source_vertex);
74     void remove_duplicate_edges(std::vector<bool>& tmp);
75     void sort_edges();
76     unsigned int color;
77     std::vector<unsigned int> edges_out;
78     std::vector<unsigned int> edges_in;
79     unsigned int nof_edges_in()const {return edges_in.size(); }
80     unsigned int nof_edges_out()const {return edges_out.size(); }
81   };
82   std::vector<Vertex> vertices;
```

```
83   void remove_duplicate_edges();
84
90   static unsigned int vertex_color_invariant(const Digraph* const g,
91                          const unsigned int v);
98   static unsigned int indegree_invariant(const Digraph* const g,
99                       const unsigned int v);
106  static unsigned int outdegree_invariant(const Digraph* const g,
107                       const unsigned int v);
113  static unsigned int selfloop_invariant(const Digraph* const g,
114                      const unsigned int v);
115
120  bool refine_according_to_invariant(unsigned int (*inv)(const Digraph* const g,
121                              const unsigned int v));
122
123  /*
124 * Routines needed when refining the partition p into equitable
125 */
126  bool split_neighbourhood_of_unit_cell(Partition::Cell* const);
127  bool split_neighbourhood_of_cell(Partition::Cell* const);
128
129
133  bool is_equitable() const;
134
135  /* Splitting heuristics, documented in more detail in the cc-file.  */
136  SplittingHeuristic sh;
137  Partition::Cell* find_next_cell_to_be_splitted(Partition::Cell *cell);
138  Partition::Cell* sh_first();
139  Partition::Cell* sh_first_smallest();
140  Partition::Cell* sh_first_largest();
141  Partition::Cell* sh_first_max_neighbours();
142  Partition::Cell* sh_first_smallest_max_neighbours();
143  Partition::Cell* sh_first_largest_max_neighbours();
144
145  /* A data structure used in many functions.
146 * Allocated only once to reduce allocation overhead,
147 * may be used only in one function at a time.
148 */
149  std::vector<Partition::Cell*> _neighbour_cells;
150
151  void make_initial_equitable_partition();
152
153  void initialize_certificate();
154
155  void sort_edges();
156
157  bool nucr_find_first_component(const unsigned int level);
158  bool nucr_find_first_component(const unsigned int level,
159                 std::vector<unsigned int>& component,
160                 unsigned int& component_elements,
161                 Partition::Cell*& sh_return);
162
163 public:
167  Digraph(const unsigned int N = 0);
168
172  ~Digraph();
173
187  static Digraph* read_dimacs(FILE* const fp, FILE* const errstr = stderr);
188
192  void write_dimacs(FILE* const fp);
193
194
198  void write_dot(FILE * const fp);
199
203  void write_dot(const char * const file_name);
204
205
206
210  virtual unsigned int get_hash();
211
215  unsigned int get_nof_vertices()const {return vertices.size(); }
216
220  unsigned int add_vertex(const unsigned int color = 0);
221
228  void add_edge(const unsigned int source, const unsigned int target);
229
233  unsigned int get_color(const unsigned int vertex) const;
234
238  void change_color(const unsigned int vertex, const unsigned int color);
239
243  Digraph* copy() const;
244
251  int cmp(Digraph& other);
252
262  void set_splitting_heuristic(SplittingHeuristic shs) {sh = shs; }
263
267  Digraph* permute(const unsigned int* const perm) const;
268
```

```
272   Digraph* permute(const std::vector<unsigned int>& perm) const;
273
277   bool is_automorphism(unsigned int* const perm) const;
278
282   bool is_automorphism(const std::vector<unsigned int>& perm) const;
283 };
284
285 } // namespace bliss
```

## 9.8 graph.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #include <bliss/abstractgraph.hh>
23
24 namespace bliss {
25
31 class Graph :  public AbstractGraph
32 {
33 public:
42   typedef enum {
46     shs_f = 0,
49     shs_fs,
52     shs_fl,
56     shs_fm,
60     shs_fsm,
64     shs_flm
65   } SplittingHeuristic;
66
67 protected:
68   class Vertex {
69   public:
70     Vertex();
71     ~Vertex();
72     void add_edge(const unsigned int other_vertex);
73     void remove_duplicate_edges(std::vector<bool>& tmp);
74     void sort_edges();
75
76     unsigned int color;
77     std::vector<unsigned int> edges;
78     unsigned int nof_edges()const {return edges.size(); }
79   };
80   std::vector<Vertex> vertices;
81   void sort_edges();
82   void remove_duplicate_edges();
83
89   static unsigned int vertex_color_invariant(const Graph* const g,
90                          const unsigned int v);
91
98   static unsigned int degree_invariant(const Graph* const g,
99                        const unsigned int v);
100
106   static unsigned int selfloop_invariant(const Graph* const g,
107                       const unsigned int v);
108
109
110   bool refine_according_to_invariant(unsigned int (*inv)(const Graph* const g,
111                              const unsigned int v));
112
113   /*
114 * Routines needed when refining the partition p into equitable
115 */
116   bool split_neighbourhood_of_unit_cell(Partition::Cell * const);
117   bool split_neighbourhood_of_cell(Partition::Cell * const);
```

```
118
122   bool is_equitable() const;
123
124   /* Splitting heuristics, documented in more detail in graph.cc */
125   SplittingHeuristic sh;
126   Partition::Cell* find_next_cell_to_be_splitted(Partition::Cell *cell);
127   Partition::Cell* sh_first();
128   Partition::Cell* sh_first_smallest();
129   Partition::Cell* sh_first_largest();
130   Partition::Cell* sh_first_max_neighbours();
131   Partition::Cell* sh_first_smallest_max_neighbours();
132   Partition::Cell* sh_first_largest_max_neighbours();
133
134
135   /* A data structure used in many functions.
136 * Allocated only once to reduce allocation overhead,
137 * may be used only in one function at a time.
138 */
139   std::vector<Partition::Cell*> _neighbour_cells;
140
141   void make_initial_equitable_partition();
142
143   void initialize_certificate();
144
145
146
147   bool nucr_find_first_component(const unsigned int level);
148   bool nucr_find_first_component(const unsigned int level,
149                  std::vector<unsigned int>& component,
150                  unsigned int& component_elements,
151                  Partition::Cell*& sh_return);
152
153
154
155
156 public:
160   Graph(const unsigned int N = 0);
161
165   ~Graph();
166
181   static Graph* read_dimacs(FILE* const fp, FILE* const errstr = stderr);
182
188   void write_dimacs(FILE* const fp);
189
193   void write_dot(FILE* const fp);
194
198   void write_dot(const char* const file_name);
199
200
201
205   virtual unsigned int get_hash();
206
210   unsigned int get_nof_vertices()const {return vertices.size(); }
211
215   Graph* permute(const unsigned int* const perm) const;
219   Graph* permute(const std::vector<unsigned int>& perm) const;
220
224   bool is_automorphism(unsigned int* const perm) const;
225
229   bool is_automorphism(const std::vector<unsigned int>& perm) const;
230
234   unsigned int add_vertex(const unsigned int color = 0);
235
242   void add_edge(const unsigned int v1, const unsigned int v2);
243
247   unsigned int get_color(const unsigned int vertex) const;
248
252   void change_color(const unsigned int vertex, const unsigned int color);
253
257   Graph* copy() const;
258
265   int cmp(Graph& other);
266
276   void set_splitting_heuristic(const SplittingHeuristic shs) {sh = shs; }
277
278
279 };
280
281 } // namespace bliss
```

## 9.9 heap.hh

```
1 #pragma once
```

```
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #include <vector>
23 #include <algorithm>
24
25 namespace bliss {
26
31 class Heap
32 {
33   std::vector<unsigned int> contents;
37   struct {
39     bool operator()(const unsigned int e1, const unsigned int e2) {return e1 > e2; }
40   } gt;
41 public:
42
47   bool is_empty()const {return contents.empty(); }
48
53   void clear() {contents.clear(); }
54
60   void insert(const unsigned int e) {
61     contents.push_back(e);
62     std::push_heap(contents.begin(), contents.end(), gt);
63   }
64
69   unsigned int smallest()const {return contents.front(); }
70
76   unsigned int remove() {
77     const unsigned int result = smallest();
78     std::pop_heap(contents.begin(),contents.end(), gt);
79     contents.pop_back();
80     return result;
81   }
82
86   size_t size()const {return contents.size(); }
87 };
88
89 } // namespace bliss
```

# 9.10 kqueue.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #include <new>
23 #include <cassert>
24
25 namespace bliss {
```

```
26
30 template <class Type>
31 class KQueue
32 {
33 public:
38   KQueue();
39
40   ~KQueue();
41
45   void init(const unsigned int N);
46
48   bool is_empty() const;
49
51   unsigned int size() const;
52
54   void clear();
55
57   Type front() const;
58
60   Type pop_front();
61
63   void push_front(Type e);
64
66   Type pop_back();
67
69   void push_back(Type e);
70 private:
71   Type *entries, *end;
72   Type *head, *tail;
73 };
74
75 template <class Type>
76 KQueue<Type>::KQueue()
77 {
78   entries = nullptr;
79   end = nullptr;
80   head = nullptr;
81   tail = nullptr;
82 }
83
84 template <class Type>
85 KQueue<Type>::~KQueue()
86 {
87   delete[] entries;
88   entries = nullptr;
89   end = nullptr;
90   head = nullptr;
91   tail = nullptr;
92 }
93
94 template <class Type>
95 void KQueue<Type>::init(const unsigned int k)
96 {
97   assert(k > 0);
98   delete[] entries;
99   entries = new Type[k+1];
100   end = entries + k + 1;
101   head = entries;
102   tail = head;
103 }
104
105 template <class Type>
106 void KQueue<Type>::clear()
107 {
108   head = entries;
109   tail = head;
110 }
111
112 template <class Type>
113 bool KQueue<Type>::is_empty()const
114 {
115   return head == tail;
116 }
117
118 template <class Type>
119 unsigned int KQueue<Type>::size()const
120 {
121   if(tail >= head)
122     return(tail - head);
123   return (end - head) + (tail - entries);
124 }
125
126 template <class Type>
127 Type KQueue<Type>::front()const
128 {
129   assert(head != tail);
130   return *head;
```

```
131 }
132
133 template <class Type>
134 Type KQueue<Type>::pop_front()
135 {
136   assert(head != tail);
137   Type *old_head = head;
138   head++;
139   if(head == end)
140     head = entries;
141   return *old_head;
142 }
143
144 template <class Type>
145 void KQueue<Type>::push_front(Type e)
146 {
147   if(head == entries)
148     head = end - 1;
149   else
150     head--;
151   assert(head != tail);
152   *head = e;
153 }
154
155 template <class Type>
156 void KQueue<Type>::push_back(Type e)
157 {
158   *tail = e;
159   tail++;
160   if(tail == end)
161     tail = entries;
162   assert(head != tail);
163 }
164
165 } // namespace bliss
```

## 9.11 orbit.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 namespace bliss {
23
36 class Orbit
37 {
38   class OrbitEntry
39   {
40   public:
41     unsigned int element;
42     OrbitEntry *next;
43     unsigned int size;
44   };
45
46   OrbitEntry *orbits;
47   OrbitEntry **in_orbit;
48   unsigned int nof_elements;
49   unsigned int _nof_orbits;
50   void merge_orbits(OrbitEntry *o1, OrbitEntry *o2);
51
52 public:
58   Orbit();
59   ~Orbit();
60
69   void init(const unsigned int N);
70
```

```
75   void reset();
76
82   void merge_orbits(unsigned int e1, unsigned int e2);
83
88   bool is_minimal_representative(unsigned int e) const;
89
94   unsigned int get_minimal_representative(unsigned int e) const;
95
100   unsigned int orbit_size(unsigned int e) const;
101
106   unsigned int nof_orbits()const {return _nof_orbits; }
107 };
108
109 } // namespace bliss
```

## 9.12 partition.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 namespace bliss {
23   class Partition;
24 }
25
26 #include <vector>
27 #include <climits>
28 #include <bliss/kqueue.hh>
29 #include <bliss/abstractgraph.hh>
30
31
32 namespace bliss {
33
44 class Partition
45 {
46 public:
50   class Cell
51   {
52     friend class Partition;
53   public:
54     unsigned int length;
55     /* Index of the first element of the cell in
56 the Partition::elements array */
57     unsigned int first;
58     unsigned int max_ival;
59     unsigned int max_ival_count;
60   private:
61     bool in_splitting_queue;
62   public:
63     bool in_neighbour_heap;
64     /* Pointer to the next cell, null if this is the last one.  */
65     Cell* next;
66     Cell* prev;
67     Cell* next_nonsingleton;
68     Cell* prev_nonsingleton;
69     unsigned int split_level;
71     bool is_unit()const {return(length == 1); }
73     bool is_in_splitting_queue()const {return(in_splitting_queue); }
74   };
75
76
77 private:
78
83   class RefInfo {
84   public:
85     unsigned int split_cell_first;
```

```
86      int prev_nonsingleton_first;
87      int next_nonsingleton_first;
88    };
92    std::vector<RefInfo> refinement_stack;
93
94    class BacktrackInfo {
95    public:
96      unsigned int refinement_stack_size;
97      unsigned int cr_backtrack_point;
98    };
99
103    std::vector<BacktrackInfo> bt_stack;
104
105 public:
106    AbstractGraph* graph;
107
108    /* Used during equitable partition refinement */
109    KQueue<Cell*> splitting_queue;
110    void  splitting_queue_add(Cell* const cell);
111    Cell* splitting_queue_pop();
112    bool  splitting_queue_is_empty() const;
113    void  splitting_queue_clear();
114
115
117    typedef unsigned int BacktrackPoint;
118
122    BacktrackPoint set_backtrack_point();
123
127    void goto_backtrack_point(BacktrackPoint p);
128
137    Cell* individualize(Cell* const cell,
138                const unsigned int element);
139
140    Cell* aux_split_in_two(Cell* const cell,
141                const unsigned int first_half_size);
142
143
144 private:
145    unsigned int N;
146    Cell* cells;
147    Cell* free_cells;
148    unsigned int discrete_cell_count;
149 public:
150    Cell* first_cell;
151    Cell* first_nonsingleton_cell;
152    unsigned int *elements;
153    /* invariant_values[e] gives the invariant value of the element e */
154    unsigned int *invariant_values;
155    /* element_to_cell_map[e] gives the cell of the element e */
156    Cell **element_to_cell_map;
158    Cell* get_cell(const unsigned int e)const {
159      assert(e < N);
160      return element_to_cell_map[e];
161    }
162    /* in_pos[e] points to the elements array s.t.  *in_pos[e] = e  */
163    unsigned int **in_pos;
164
165    Partition();
166    ~Partition();
167
172    void init(const unsigned int N);
173
178    bool is_discrete()const {return(free_cells == 0); }
179
180    unsigned int nof_discrete_cells()const {return(discrete_cell_count); }
181
185    size_t print(FILE* const fp, const bool add_newline = true) const;
186
190    size_t print_signature(FILE* const fp, const bool add_newline = true) const;
191
192    /*
193  * Splits the Cell \a cell into [cell_1,...,cell_n]
194  * according to the invariant_values of the elements in \a cell.
195  * After splitting, cell_1 == \a cell.
196  * Returns the pointer to the Cell cell_n;
197  * cell_n != cell iff the Cell \a cell was actually splitted.
198  * The flag \a max_ival_info_ok indicates whether the max_ival and
199  * max_ival_count fields of the Cell \a cell have consistent values
200  * when the method is called.
201  * Clears the invariant values of the elements in the Cell \a cell as well as
202  * the max_ival and max_ival_count fields of the Cell \a cell.
203  */
204    Cell *zplit_cell(Cell * const cell, const bool max_ival_info_ok);
205
206    /*
207  * Routines for component recursion
208  */
```

```
209   void cr_init();
210   void cr_free();
211   unsigned int cr_get_level(const unsigned int cell_index) const;
212   unsigned int cr_split_level(const unsigned int level,
213                   const std::vector<unsigned int>& cells);
214
216   void clear_ivs(Cell* const cell);
217
218 private:
219   /*
220 * Component recursion data structures
221 */
222
223   /* Is component recursion support in use?  */
224   bool cr_enabled;
225
226   class CRCell {
227   public:
228     unsigned int level;
229     CRCell* next;
230     CRCell** prev_next_ptr;
231     void detach() {
232       if(next)
233     next->prev_next_ptr = prev_next_ptr;
234       *(prev_next_ptr) = next;
235       level = UINT_MAX;
236       next = nullptr;
237       prev_next_ptr = nullptr;
238     }
239   };
240   CRCell* cr_cells;
241   CRCell** cr_levels;
242   class CR_BTInfo {
243   public:
244     unsigned int created_trail_index;
245     unsigned int splitted_level_trail_index;
246   };
247   std::vector<unsigned int> cr_created_trail;
248   std::vector<unsigned int> cr_splitted_level_trail;
249   std::vector<CR_BTInfo> cr_bt_info;
250   unsigned int cr_max_level;
251   void cr_create_at_level(const unsigned int cell_index, unsigned int level);
252   void cr_create_at_level_trailed(const unsigned int cell_index, unsigned int level);
253   unsigned int cr_get_backtrack_point();
254   void cr_goto_backtrack_point(const unsigned int btpoint);
255
256
257   /*
258 *
259 * Auxiliary routines for sorting and splitting cells
260 *
261 */
262   Cell* sort_and_split_cell1(Cell* cell);
263   Cell* sort_and_split_cell255(Cell* const cell, const unsigned int max_ival);
264   bool shellsort_cell(Cell* cell);
265   Cell* split_cell(Cell* const cell);
266
267   /*
268 * Some auxiliary stuff needed for distribution count sorting.
269 * To make the code thread-safe (modulo the requirement that each graph is
270 * only accessed in one thread at a time), the arrays are owned by
271 * the partition instance, not statically defined.
272 */
273   unsigned int dcs_count[256];
274   unsigned int dcs_start[256];
275   void dcs_cumulate_count(const unsigned int max);
276 };
277
278
279 inline Partition::Cell*
280 Partition::splitting_queue_pop()
281 {
282   assert(!splitting_queue.is_empty());
283   Cell* const cell = splitting_queue.pop_front();
284   assert(cell->in_splitting_queue);
285   cell->in_splitting_queue = false;
286   return cell;
287 }
288
289 inline bool
290 Partition::splitting_queue_is_empty()const
291 {
292   return splitting_queue.is_empty();
293 }
294
295
296 inline unsigned int
```

```
297 Partition::cr_get_level(const unsigned int cell_index)const
298 {
299   assert(cr_enabled);
300   assert(cell_index < N);
301   assert(cr_cells[cell_index].level != UINT_MAX);
302   return(cr_cells[cell_index].level);
303 }
304
305 } // namespace bliss
```

## 9.13 stats.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 #include <cstdio>
23 #include <bliss/abstractgraph.hh>
24 #include <bliss/bignum.hh>
25
26 namespace bliss {
27
31 class Stats
32 {
33   friend class AbstractGraph;
35   BigNum group_size;
38   long double group_size_approx;
40   long unsigned int nof_nodes;
42   long unsigned int nof_leaf_nodes;
44   long unsigned int nof_bad_nodes;
46   long unsigned int nof_canupdates;
48   long unsigned int nof_generators;
50   unsigned long int max_level;
52   void reset()
53   {
54     group_size.assign(1);
55     group_size_approx = 1.0;
56     nof_nodes = 0;
57     nof_leaf_nodes = 0;
58     nof_bad_nodes = 0;
59     nof_canupdates = 0;
60     nof_generators = 0;
61     max_level = 0;
62   }
63 public:
64   Stats() { reset(); }
66   size_t print(FILE* const fp)const
67 {
68     size_t r = 0;
69     r += fprintf(fp, "Nodes:                    %lu\n", nof_nodes);
70     r += fprintf(fp, "Leaf nodes:           %lu\n", nof_leaf_nodes);
71     r += fprintf(fp, "Bad nodes:             %lu\n", nof_bad_nodes);
72     r += fprintf(fp, "Canrep updates:  %lu\n", nof_canupdates);
73     r += fprintf(fp, "Generators:           %lu\n", nof_generators);
74     r += fprintf(fp, "Max level:             %lu\n", max_level);
75     r += fprintf(fp, "|Aut|:                    ")+group_size.print(fp)+fprintf(fp, "\n");
76     fflush(fp);
77     return r;
78   }
80   const BigNum& get_group_size()const {return group_size;}
83   long double get_group_size_approx()const {return group_size_approx;}
85   long unsigned int get_nof_nodes()const {return nof_nodes;}
87   long unsigned int get_nof_leaf_nodes()const {return nof_leaf_nodes;}
89   long unsigned int get_nof_bad_nodes()const {return nof_bad_nodes;}
91   long unsigned int get_nof_canupdates()const {return nof_canupdates;}
93   long unsigned int get_nof_generators()const {return nof_generators;}
```

```
95   unsigned long int get_max_level()const {return max_level;}
96 };
97
98 } // namespace bliss
```

## 9.14 timer.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22
23 #include <chrono>
24
25 namespace bliss {
26
30 class Timer
31 {
32   std::chrono::high_resolution_clock::time_point start_time_;
33 public:
37   Timer() {
38     reset();
39   }
40
44   void reset() {
45     start_time_ = std::chrono::high_resolution_clock::now();
46   }
47
52   double get_duration()const {
53     return std::chrono::duration_cast<std::chrono::duration<double>
      >(std::chrono::high_resolution_clock::now() - start_time_).count();
54   }
55 };
56
57 } // namespace bliss
```

## 9.15 uintseqhash.hh

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
22 namespace bliss {
23
27 class UintSeqHash
```

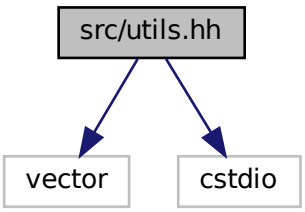
```
28 {
29 protected:
30   unsigned int h;
31 public:
32   UintSeqHash() {h = 0; }
33   UintSeqHash(const UintSeqHash &other) {h = other.h; }
34   UintSeqHash& operator=(const UintSeqHash &other) {h = other.h; return *this; }
35
37   void reset() {h = 0; }
38
40   void update(unsigned int n);
41
43   unsigned int get_value()const {return h; }
44
48   int cmp(const UintSeqHash &other)const {
49     return (h < other.h)?-1:((h == other.h)?0:1);
50   }
52   bool is_lt(const UintSeqHash &other)const {return cmp(other) < 0; }
54   bool is_le(const UintSeqHash &other)const {return cmp(other) <= 0; }
56   bool is_equal(const UintSeqHash &other)const {return cmp(other) == 0; }
57 };
58
59
60 } // namespace bliss
```

## 9.16 src/utils.hh File Reference

Some small utilities.

```
#include <vector>
#include <cstdio>
```

Include dependency graph for utils.hh:




### Namespaces

- namespace bliss

### Functions

- size_t bliss::print_permutation (FILE *const fp, const unsigned int N, const unsigned int *perm, const unsigned int offset)
- size_t 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)

### 9.16.1 Detailed Description

Some small utilities.

## 9.17 utils.hh

Go to the documentation of this file.

```
1 #pragma once
2
3 /*
4 Copyright (c) 2003-2021 Tommi Junttila
5 Released under the GNU Lesser General Public License version 3.
6
7 This file is part of bliss.
8
9 bliss is free software:  you can redistribute it and/or modify
10 it under the terms of the GNU Lesser General Public License as published by
11 the Free Software Foundation, version 3 of the License.
12
13 bliss is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU Lesser General Public License for more details.
17
18 You should have received a copy of the GNU Lesser General Public License
19 along with bliss.    If not, see <http://www.gnu.org/licenses/>.
20 */
21
27 #include <vector>
28 #include <cstdio>
29
30 namespace bliss {
31
38 size_t print_permutation(FILE* fp,
39              const unsigned int N,
40              const unsigned int* perm,
41              const unsigned int offset = 0);
42
49 size_t print_permutation(FILE* fp,
50              const std::vector<unsigned int>& perm,
51              const unsigned int offset = 0);
52
57 bool is_permutation(const unsigned int N, const unsigned int* perm);
58
63 bool is_permutation(const std::vector<unsigned int>& perm);
64
65 } // namespace bliss
```

# Index