

## The bliss C++ API

Generated by Doxygen 1.9.4



<b>1 Outline</b>	<b>1</b>
1.1 The C language API	1
<b>2 The bliss executable</b>	<b>3</b>
<b>3 Namespace Index</b>	<b>7</b>
3.1 Namespace List	7
<b>4 Hierarchical Index</b>	<b>9</b>
4.1 Class Hierarchy	9
<b>5 Class Index</b>	<b>11</b>
5.1 Class List	11
<b>6 File Index</b>	<b>13</b>
6.1 File List	13
<b>7 Namespace Documentation</b>	<b>15</b>
7.1 bliss Namespace Reference	15
7.1.1 Detailed Description	16
7.1.2 Function Documentation	16
7.1.2.1 is_permutation() [1/2]	16
7.1.2.2 is_permutation() [2/2]	16
7.1.2.3 print_permutation() [1/2]	16
7.1.2.4 print_permutation() [2/2]	16
<b>8 Class Documentation</b>	<b>17</b>
8.1 bliss::AbstractGraph Class Reference	17
8.1.1 Detailed Description	18
8.1.2 Member Function Documentation	18
8.1.2.1 add_edge()	18
8.1.2.2 add_vertex()	18
8.1.2.3 canonical_form()	18
8.1.2.4 change_color()	19
8.1.2.5 find_automorphisms()	19
8.1.2.6 get_color()	19
8.1.2.7 get_hash()	19
8.1.2.8 get_nof_vertices()	20
8.1.2.9 is_automorphism() [1/2]	20
8.1.2.10 is_automorphism() [2/2]	20
8.1.2.11 permute() [1/2]	20
8.1.2.12 permute() [2/2]	20
8.1.2.13 set_component_recursion()	21
8.1.2.14 set_failure_recording()	21
8.1.2.15 set_long_prune_activity()	21

8.1.2.16 set_verbose_file()	21
8.1.2.17 set_verbose_level()	22
8.1.2.18 write_dimacs()	22
8.1.2.19 write_dot() [1/2]	22
8.1.2.20 write_dot() [2/2]	23
8.2 bliss::BigNum Class Reference	23
8.2.1 Detailed Description	23
8.2.2 Constructor & Destructor Documentation	24
8.2.2.1 BigNum()	24
8.2.2.2 ~BigNum()	24
8.2.3 Member Function Documentation	24
8.2.3.1 assign()	24
8.2.3.2 get_factors()	24
8.2.3.3 multiply()	24
8.2.3.4 print()	24
8.2.3.5 to_string()	25
8.3 bliss_graph_struct Struct Reference	25
8.3.1 Detailed Description	25
8.4 bliss_stats_struct Struct Reference	25
8.4.1 Detailed Description	26
8.5 bliss::Partition::Cell Class Reference	26
8.5.1 Detailed Description	26
8.5.2 Member Function Documentation	26
8.5.2.1 is_in_splitting_queue()	26
8.5.2.2 is_unit()	26
8.6 bliss::Digraph Class Reference	27
8.6.1 Detailed Description	28
8.6.2 Member Enumeration Documentation	28
8.6.2.1 SplittingHeuristic	28
8.6.3 Constructor & Destructor Documentation	28
8.6.3.1 Digraph()	28
8.6.3.2 ~Digraph()	28
8.6.4 Member Function Documentation	29
8.6.4.1 add_edge()	29
8.6.4.2 add_vertex()	29
8.6.4.3 canonical_form()	29
8.6.4.4 change_color()	30
8.6.4.5 cmp()	30
8.6.4.6 copy()	30
8.6.4.7 find_automorphisms()	30
8.6.4.8 get_color()	31
8.6.4.9 get_hash()	31

8.6.4.10 <code>get_nof_vertices()</code>	31
8.6.4.11 <code>is_automorphism()</code> [1/2]	31
8.6.4.12 <code>is_automorphism()</code> [2/2]	32
8.6.4.13 <code>permute()</code> [1/2]	32
8.6.4.14 <code>permute()</code> [2/2]	32
8.6.4.15 <code>read_dimacs()</code>	32
8.6.4.16 <code>set_component_recursion()</code>	33
8.6.4.17 <code>set_failure_recording()</code>	33
8.6.4.18 <code>set_long_prune_activity()</code>	33
8.6.4.19 <code>set_splitting_heuristic()</code>	34
8.6.4.20 <code>set_verbose_file()</code>	34
8.6.4.21 <code>set_verbose_level()</code>	34
8.6.4.22 <code>write_dimacs()</code>	34
8.6.4.23 <code>write_dot()</code> [1/2]	35
8.6.4.24 <code>write_dot()</code> [2/2]	35
8.7 bliss::Graph Class Reference	35
8.7.1 Detailed Description	36
8.7.2 Member Enumeration Documentation	37
8.7.2.1 <code>SplittingHeuristic</code>	37
8.7.3 Constructor & Destructor Documentation	37
8.7.3.1 <code>Graph()</code>	37
8.7.3.2 <code>~Graph()</code>	37
8.7.4 Member Function Documentation	37
8.7.4.1 <code>add_edge()</code>	38
8.7.4.2 <code>add_vertex()</code>	38
8.7.4.3 <code>canonical_form()</code>	38
8.7.4.4 <code>change_color()</code>	38
8.7.4.5 <code>cmp()</code>	39
8.7.4.6 <code>copy()</code>	39
8.7.4.7 <code>find_automorphisms()</code>	39
8.7.4.8 <code>get_color()</code>	39
8.7.4.9 <code>get_hash()</code>	40
8.7.4.10 <code>get_nof_vertices()</code>	40
8.7.4.11 <code>is_automorphism()</code> [1/2]	40
8.7.4.12 <code>is_automorphism()</code> [2/2]	40
8.7.4.13 <code>permute()</code> [1/2]	41
8.7.4.14 <code>permute()</code> [2/2]	41
8.7.4.15 <code>read_dimacs()</code>	41
8.7.4.16 <code>set_component_recursion()</code>	41
8.7.4.17 <code>set_failure_recording()</code>	42
8.7.4.18 <code>set_long_prune_activity()</code>	42
8.7.4.19 <code>set_splitting_heuristic()</code>	42

8.7.4.20 set_verbose_file()	43
8.7.4.21 set_verbose_level()	43
8.7.4.22 write_dimacs()	43
8.7.4.23 write_dot() [1/2]	43
8.7.4.24 write_dot() [2/2]	44
8.8 bliss::Heap Class Reference	44
8.8.1 Detailed Description	44
8.8.2 Member Function Documentation	44
8.8.2.1 clear()	45
8.8.2.2 insert()	45
8.8.2.3 is_empty()	45
8.8.2.4 remove()	45
8.8.2.5 size()	45
8.8.2.6 smallest()	45
8.9 bliss::KQueue< Type > Class Template Reference	45
8.9.1 Detailed Description	46
8.9.2 Constructor & Destructor Documentation	46
8.9.2.1 KQueue()	46
8.9.3 Member Function Documentation	46
8.9.3.1 clear()	46
8.9.3.2 front()	46
8.9.3.3 init()	47
8.9.3.4 is_empty()	47
8.9.3.5 pop_back()	47
8.9.3.6 pop_front()	47
8.9.3.7 push_back()	47
8.9.3.8 push_front()	47
8.9.3.9 size()	48
8.10 bliss::Orbit Class Reference	48
8.10.1 Detailed Description	48
8.10.2 Constructor & Destructor Documentation	48
8.10.2.1 Orbit()	48
8.10.3 Member Function Documentation	49
8.10.3.1 get_minimal_representative()	49
8.10.3.2 init()	49
8.10.3.3 is_minimal_representative()	49
8.10.3.4 merge_orbits()	49
8.10.3.5 nof_orbits()	49
8.10.3.6 orbit_size()	50
8.10.3.7 reset()	50
8.11 bliss::Partition Class Reference	50
8.11.1 Detailed Description	51

8.11.2 Member Typedef Documentation	51
8.11.2.1 BacktrackPoint	51
8.11.3 Member Function Documentation	51
8.11.3.1 clear_ivs()	51
8.11.3.2 get_cell()	51
8.11.3.3 goto_backtrack_point()	51
8.11.3.4 individualize()	51
8.11.3.5 init()	52
8.11.3.6 is_discrete()	52
8.11.3.7 print()	52
8.11.3.8 print_signature()	52
8.11.3.9 set_backtrack_point()	52
8.12 bliss::Stats Class Reference	53
8.12.1 Detailed Description	53
8.12.2 Member Function Documentation	53
8.12.2.1 get_group_size()	53
8.12.2.2 get_group_size_approx()	53
8.12.2.3 get_max_level()	53
8.12.2.4 get_nof_bad_nodes()	54
8.12.2.5 get_nof_canupdates()	54
8.12.2.6 get_nof_generators()	54
8.12.2.7 get_nof_leaf_nodes()	54
8.12.2.8 get_nof_nodes()	54
8.12.2.9 print()	54
8.13 bliss::Timer Class Reference	54
8.13.1 Detailed Description	55
8.14 bliss::UIntSeqHash Class Reference	55
8.14.1 Detailed Description	55
8.14.2 Member Function Documentation	55
8.14.2.1 cmp()	55
8.14.2.2 get_value()	56
8.14.2.3 is_equal()	56
8.14.2.4 is_le()	56
8.14.2.5 is_lt()	56
8.14.2.6 reset()	56
8.14.2.7 update()	56
<b>9 File Documentation</b>	<b>57</b>
9.1 abstractgraph.hh	57
9.2 bignum.hh	60
9.3 src/bliss_C.h File Reference	62
9.4 bliss_C.h	62

---

9.5 src/defs.hh File Reference . . . . .	64
9.5.1 Detailed Description . . . . .	64
9.6 defs.hh . . . . .	64
9.7 digraph.hh . . . . .	65
9.8 graph.hh . . . . .	67
9.9 heap.hh . . . . .	68
9.10 kqueue.hh . . . . .	69
9.11 orbit.hh . . . . .	71
9.12 partition.hh . . . . .	72
9.13 stats.hh . . . . .	75
9.14 timer.hh . . . . .	76
9.15 uintseqhash.hh . . . . .	76
9.16 src/utls.hh File Reference . . . . .	77
9.16.1 Detailed Description . . . . .	77
9.17 utls.hh . . . . .	77
<b>Index</b>	<b>79</b>



# 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:

<a href="#">bliss</a> . . . . .	15
---------------------------------	----





## Chapter 4

# Hierarchical Index

### 4.1 Class Hierarchy

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

bliss::AbstractGraph . . . . .	17
bliss::Digraph . . . . .	27
bliss::Graph . . . . .	35
bliss::BigNum . . . . .	23
bliss_graph_struct . . . . .	25
bliss_stats_struct . . . . .	25
bliss::Partition::Cell . . . . .	26
bliss::Heap . . . . .	44
bliss::KQueue< Type > . . . . .	45
bliss::KQueue< bliss::Partition::Cell * > . . . . .	45
bliss::Orbit . . . . .	48
bliss::Partition . . . . .	50
bliss::Stats . . . . .	53
bliss::Timer . . . . .	54
bliss::UIntSeqHash . . . . .	55



## Chapter 5

# Class Index

### 5.1 Class List

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

<a href="#">bliss::AbstractGraph</a>	An abstract base class for different types of graphs . . . . .	17
<a href="#">bliss::BigNum</a>	A simple wrapper class for non-negative big integers (or approximation of them) . . . . .	23
<a href="#">bliss_graph_struct</a>	The true bliss graph is hiding in this struct . . . . .	25
<a href="#">bliss_stats_struct</a>	The C API version of the statistics returned by the bliss search algorithm . . . . .	25
<a href="#">bliss::Partition::Cell</a>	Data structure for holding information about a cell in a <a href="#">Partition</a> . . . . .	26
<a href="#">bliss::Digraph</a>	The class for directed, vertex colored graphs . . . . .	27
<a href="#">bliss::Graph</a>	The class for undirected, vertex colored graphs . . . . .	35
<a href="#">bliss::Heap</a>	A min-heap of unsigned integers . . . . .	44
<a href="#">bliss::KQueue&lt; Type &gt;</a>	A simple implementation of queues with fixed maximum capacity . . . . .	45
<a href="#">bliss::Orbit</a>	A class for representing orbit information . . . . .	48
<a href="#">bliss::Partition</a>	A class for refinable, backtrackable ordered partitions . . . . .	50
<a href="#">bliss::Stats</a>	Statistics returned by the bliss search algorithm . . . . .	53
<a href="#">bliss::Timer</a>	A simple helper class for measuring elapsed time . . . . .	54
<a href="#">bliss::UIntSeqHash</a>	A updatable hash for sequences of unsigned ints . . . . .	55



## Chapter 6

# File Index

### 6.1 File List

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

src/ <a href="#">abstractgraph.hh</a> . . . . .	57
src/ <a href="#">bignum.hh</a> . . . . .	60
src/ <a href="#">bliss_C.h</a>	
The bliss C API . . . . .	62
src/ <a href="#">defs.hh</a>	
Some common definitions . . . . .	64
src/ <a href="#">digraph.hh</a> . . . . .	65
src/ <a href="#">graph.hh</a> . . . . .	67
src/ <a href="#">heap.hh</a> . . . . .	68
src/ <a href="#">kqueue.hh</a> . . . . .	69
src/ <a href="#">orbit.hh</a> . . . . .	71
src/ <a href="#">partition.hh</a> . . . . .	72
src/ <a href="#">stats.hh</a> . . . . .	75
src/ <a href="#">timer.hh</a> . . . . .	76
src/ <a href="#">uintseqhash.hh</a> . . . . .	76
src/ <a href="#">utils.hh</a>	
Some small utilities . . . . .	77



## 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  $\{0, \dots, \text{get\_nof\_vertices}() - 1\}$ ) 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  $N=\text{this.get\_nof\_vertices}()$  elements and be a bijection on  $\{0,1,\dots,N-1\}$ , 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  $N=\text{this.get\_nof\_vertices}()$  elements and be a bijection on  $\{0,1,\dots,N-1\}$ , 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

<i>active</i>	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

<i>active</i>	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

<i>active</i>	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

<i>fp</i>	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

<i>level</i>	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

<i>fp</i>	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

<i>file_name</i>	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

<i>fp</i>	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  $\{0, \dots, \text{get\_nof\_vertices}() - 1\}$ ) 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 order 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  $N = \text{this.get\_nof\_vertices}()$  elements and be a bijection on  $\{0, 1, \dots, N-1\}$ , 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  $N = \text{this.get\_nof\_vertices}()$  elements and be a bijection on  $\{0, 1, \dots, N-1\}$ , 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**

<i>fp</i>	the file stream for the graph file
<i>errstr</i>	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**

<i>active</i>	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**

<i>active</i>	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**

<i>active</i>	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

<i>fp</i>	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

<i>level</i>	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

<i>fp</i>	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

<i>file_name</i>	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

<i>fp</i>	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  $\{0, \dots, \text{get\_nof\_vertices}() - 1\}$ ) 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 order 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  $N=\text{this.get\_nof\_vertices}()$  elements and be a bijection on  $\{0,1,\dots,N-1\}$ , 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  $N=\text{this.get\_nof\_vertices}()$  elements and be a bijection on  $\{0,1,\dots,N-1\}$ , 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**

<i>fp</i>	the file stream for the graph file
<i>errstr</i>	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

<i>active</i>	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

<i>active</i>	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

<i>active</i>	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

<i>fp</i>	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

<i>level</i>	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

<i>file_name</i>	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 dotted format.

##### Parameters

<i>fp</i>	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

<i>cell</i>	a non-unit <a href="#">Cell</a>
<i>element</i>	an element in <i>cell</i>

## 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  $\{0, \dots, N-1\}$ .

**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
22 namespace bliss {
23     class AbstractGraph;
24 }
25
26 #include <cstdio>
27 #include <functional>
28 #include <vector>
29 #include <bliss/stats.hh>
30 #include <bliss/kqueue.hh>
31 #include <bliss/heap.hh>
32 #include <bliss/orbit.hh>
33 #include <bliss/partition.hh>
34 #include <bliss/uintseqhash.hh>
35
36 namespace bliss {
37
38     class AbstractGraph
39     {
40     public:
41         friend class Partition;
42
43         AbstractGraph();
44         virtual ~AbstractGraph();
45
46         void set_verbose_level(const unsigned int level);
47
48         void set_verbose_file(FILE * const fp);
49
50         virtual unsigned int add_vertex(const unsigned int color = 0) = 0;
51
52         virtual void add_edge(const unsigned int source, const unsigned int target) = 0;
53
54         virtual unsigned int get_color(const unsigned int vertex) const = 0;
55
56         virtual void change_color(const unsigned int vertex, const unsigned int color) = 0;
57     };
58 }
```

```

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 =
205     nullptr,
206     const std::function<bool()>& terminate = nullptr);
207
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:
255     unsigned int verbose_level;
256
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_mcrcs;
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_mcrcs(const unsigned int index);
347 std::vector<bool>& long_prune_allocget_mcrcs(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, then 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_split(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
448  class CR_CEP {
449  public:
450      unsigned int creation_level;
451      unsigned int discrete_cell_limit;
452      unsigned int next_cr_level;
453      unsigned int next_cep_index;
454      bool first_checked;
455      bool best_checked;
456  };
457  std::vector<CR_CEP> cr_cep_stack;
458
459  virtual bool nucr_find_first_component(const unsigned int level) = 0;
460  virtual bool nucr_find_first_component(const unsigned int level,
461      std::vector<unsigned int>& component,
462      unsigned int& component_elements,
463      Partition::Cell*& sh_return) = 0;
464  std::vector<unsigned int> cr_component;
465  unsigned int cr_component_elements;
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
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
36 #if defined(BLISS_USE_GMP)
37
38 class BigNum
39 {
40 public:
41     BigNum() {mpz_init(v); }
42     ~BigNum() {mpz_clear(v); }
43
44     void assign(unsigned int n) {mpz_set_ui(v, n); }
45     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:

## 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
23 #include <stdlib.h>
24 #include <stdio.h>
25
26
27 typedef struct bliss_graph_struct BlissGraph;
28
29
30 typedef struct bliss_stats_struct
31 {
32     long double group_size_approx;
33     long unsigned int nof_nodes;
34     long unsigned int nof_leaf_nodes;
35     long unsigned int nof_bad_nodes;
36     long unsigned int nof_canupdates;
37     long unsigned int nof_generators;
38     unsigned long int max_level;
39 } BlissStats;
40
41
42 BlissGraph *bliss_new(const unsigned int N);
43
44
45 BlissGraph *bliss_read_dimacs(FILE *fp);
46
47
48 void bliss_write_dimacs(BlissGraph *graph, FILE *fp);
49
50
51 void bliss_release(BlissGraph *graph);
52
53
54 void bliss_write_dot(BlissGraph *graph, FILE *fp);
55
56
57 unsigned int bliss_get_nof_vertices(BlissGraph *graph);
58
59
60 unsigned int bliss_add_vertex(BlissGraph *graph, unsigned int c);
61
62
63 void bliss_add_edge(BlissGraph *graph, unsigned int v1, unsigned int v2);
64
65
66 int bliss_cmp(BlissGraph *graph1, BlissGraph *graph2);
67
68
69 unsigned int bliss_hash(BlissGraph *graph);
70
71
72 BlissGraph *bliss_permute(BlissGraph *graph, const unsigned int *perm);
73
74
75 void
76 bliss_find_automorphisms(BlissGraph *graph,
77     void (*hook)(void *user_param,
78         unsigned int N,
79         const unsigned int *aut),
80     void *hook_user_param,
81     BlissStats *stats);
82
83
84 const unsigned int *
85 bliss_find_canonical_labeling(BlissGraph *graph,
86     void (*hook)(void *user_param,
87         unsigned int N,
88         const unsigned int *aut),
89     void *hook_user_param,
90     BlissStats *stats);
91

```

```
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
25 #define BLISS_VERSION "0.77"
26 #define BLISS_VERSION_MAJOR 0
27 #define BLISS_VERSION_MINOR 77
28
29 namespace bliss {
30
31 static const char * const version = "0.77";
32
33 }
```



```

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
26 class Digraph : public AbstractGraph
27 {
28 public:
29     typedef enum {
30         shs_f = 0,
31         shs_fs,
32         shs_fl,
33         shs_fm,
34         shs_fsm,
35         shs_flm
36     } SplittingHeuristic;
37
38 protected:
39     class Vertex {
40     public:
41         Vertex();
42         ~Vertex();
43         void add_edge_to(const unsigned int dest_vertex);
44         void add_edge_from(const unsigned int source_vertex);
45         void remove_duplicate_edges(std::vector<bool>& tmp);
46         void sort_edges();
47         unsigned int color;
48         std::vector<unsigned int> edges_out;
49         std::vector<unsigned int> edges_in;
50         unsigned int nof_edges_in() const {return edges_in.size(); }
51         unsigned int nof_edges_out() const {return edges_out.size(); }
52     };
53     std::vector<Vertex> vertices;
54     void remove_duplicate_edges();
55
56     static unsigned int vertex_color_invariant(const Digraph* const g,
57                                                const unsigned int v);
58     static unsigned int indegree_invariant(const Digraph* const g,
59                                            const unsigned int v);
60

```

```

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_split(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:
164 Digraph(const unsigned int N = 0);
165
166
172 ~Digraph();
173
174 static Digraph* read_dimacs(FILE* const fp, FILE* const errstr = stderr);
175
176 void write_dimacs(FILE* const fp);
177
178 void write_dot(FILE * const fp);
179
180 void write_dot(const char * const file_name);
181
182
183 virtual unsigned int get_hash();
184
185 unsigned int get_nof_vertices()const {return vertices.size(); }
186
187 unsigned int add_vertex(const unsigned int color = 0);
188
189 void add_edge(const unsigned int source, const unsigned int target);
190
191 unsigned int get_color(const unsigned int vertex) const;
192
193 void change_color(const unsigned int vertex, const unsigned int color);
194
195 Digraph* copy() const;
196
197 int cmp(Digraph& other);
198
199 void set_splitting_heuristic(SplittingHeuristic shs) {sh = shs; }
200
201 Digraph* permute(const unsigned int* const perm) const;
202
203 Digraph* permute(const std::vector<unsigned int>& perm) const;
204
205 bool is_automorphism(unsigned int* const perm) const;
206
207 bool is_automorphism(const std::vector<unsigned int>& perm) const;
208
209 };

```

```

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
26 class Graph : public AbstractGraph
27 {
28 public:
29     typedef enum {
30         shs_f = 0,
31         shs_fs,
32         shs_fl,
33         shs_fm,
34         shs_fsm,
35         shs_flm
36     } SplittingHeuristic;
37
38 protected:
39     class Vertex {
40     public:
41         Vertex();
42         ~Vertex();
43         void add_edge(const unsigned int other_vertex);
44         void remove_duplicate_edges(std::vector<bool>& tmp);
45         void sort_edges();
46
47         unsigned int color;
48         std::vector<unsigned int> edges;
49         unsigned int nof_edges() const {return edges.size(); }
50     };
51     std::vector<Vertex> vertices;
52     void sort_edges();
53     void remove_duplicate_edges();
54
55     static unsigned int vertex_color_invariant(const Graph* const g,
56                                                const unsigned int v);
57
58     static unsigned int degree_invariant(const Graph* const g,
59                                          const unsigned int v);
60
61     static unsigned int selfloop_invariant(const Graph* const g,
62                                           const unsigned int v);
63
64     bool refine_according_to_invariant(unsigned int (*inv)(const Graph* const g,
65                                                           const unsigned int v));
66
67     /*
68     * Routines needed when refining the partition p into equitable
69     */
70     bool split_neighbourhood_of_unit_cell(Partition::Cell * const);
71     bool split_neighbourhood_of_cell(Partition::Cell * const);
72
73     bool is_equitable() const;
74
75     /* Splitting heuristics, documented in more detail in graph.cc */
76     SplittingHeuristic sh;
77     Partition::Cell* find_next_cell_to_be_split(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:
157     Graph(const unsigned int N = 0);
158
159     ~Graph();
160
161
162 static Graph* read_dimacs(FILE* const fp, FILE* const errstr = stderr);
163
164 void write_dimacs(FILE* const fp);
165
166 void write_dot(FILE* const fp);
167
168 void write_dot(const char* const file_name);
169
170
171
172 virtual unsigned int get_hash();
173
174 unsigned int get_nof_vertices() const {return vertices.size(); }
175
176
177 Graph* permute(const unsigned int* const perm) const;
178 Graph* permute(const std::vector<unsigned int>& perm) const;
179
180
181 bool is_automorphism(unsigned int* const perm) const;
182
183 bool is_automorphism(const std::vector<unsigned int>& perm) const;
184
185
186 unsigned int add_vertex(const unsigned int color = 0);
187
188 void add_edge(const unsigned int v1, const unsigned int v2);
189
190 unsigned int get_color(const unsigned int vertex) const;
191
192 void change_color(const unsigned int vertex, const unsigned int color);
193
194 Graph* copy() const;
195
196 int cmp(Graph& other);
197
198 void set_splitting_heuristic(const SplittingHeuristic shs) {sh = shs; }
199
200 };
201
202 } // 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
27 class Heap
28 {
29     std::vector<unsigned int> contents;
30     struct {
31         bool operator()(const unsigned int e1, const unsigned int e2) {return e1 > e2; }
32     } gt;
33 public:
34
35     bool is_empty()const {return contents.empty(); }
36
37     void clear() {contents.clear(); }
38
39     void insert(const unsigned int e) {
40         contents.push_back(e);
41         std::push_heap(contents.begin(), contents.end(), gt);
42     }
43
44     unsigned int smallest()const {return contents.front(); }
45
46     unsigned int remove() {
47         const unsigned int result = smallest();
48         std::pop_heap(contents.begin(), contents.end(), gt);
49         contents.pop_back();
50         return result;
51     }
52
53     size_t size()const {return contents.size(); }
54 };
55
56 } // 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
27 template <class Type>
28 class KQueue
29 {
30 public:
31     KQueue();

```

```

39
40 ~KQueue();
41
42 void init(const unsigned int N);
43
44 bool is_empty() const;
45
46 unsigned int size() const;
47
48 void clear();
49
50 Type front() const;
51
52 Type pop_front();
53
54 void push_front(Type e);
55
56 Type pop_back();
57
58 void push_back(Type e);
59 private:
60 Type *entries, *end;
61 Type *head, *tail;
62 };
63
64 template <class Type>
65 KQueue<Type>::KQueue()
66 {
67     entries = nullptr;
68     end = nullptr;
69     head = nullptr;
70     tail = nullptr;
71 }
72
73 template <class Type>
74 KQueue<Type>::~KQueue()
75 {
76     delete[] entries;
77     entries = nullptr;
78     end = nullptr;
79     head = nullptr;
80     tail = nullptr;
81 }
82
83 template <class Type>
84 void KQueue<Type>::init(const unsigned int k)
85 {
86     assert(k > 0);
87     delete[] entries;
88     entries = new Type[k+1];
89     end = entries + k + 1;
90     head = entries;
91     tail = head;
92 }
93
94 template <class Type>
95 void KQueue<Type>::clear()
96 {
97     head = entries;
98     tail = head;
99 }
100
101 template <class Type>
102 bool KQueue<Type>::is_empty() const
103 {
104     return head == tail;
105 }
106
107 template <class Type>
108 unsigned int KQueue<Type>::size() const
109 {
110     if(tail >= head)
111         return(tail - head);
112     return (end - head) + (tail - entries);
113 }
114
115 template <class Type>
116 Type KQueue<Type>::front() const
117 {
118     assert(head != tail);
119     return *head;
120 }
121
122 template <class Type>
123 Type KQueue<Type>::pop_front()
124 {
125     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
24 class Orbit
25 {
26     class OrbitEntry
27     {
28     public:
29         unsigned int element;
30         OrbitEntry *next;
31         unsigned int size;
32     };
33
34     OrbitEntry *orbits;
35     OrbitEntry **in_orbit;
36     unsigned int nof_elements;
37     unsigned int _nof_orbits;
38     void merge_orbits(OrbitEntry *o1, OrbitEntry *o2);
39
40 public:
41     Orbit();
42     ~Orbit();
43
44     void init(const unsigned int N);
45
46     void reset();
47
48     void merge_orbits(unsigned int e1, unsigned int e2);
49
50     bool is_minimal_representative(unsigned int e) const;
51
52 }

```

```

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
34     class Partition
35     {
36     public:
37         class Cell
38         {
39         friend class Partition;
40         public:
41             unsigned int length;
42             /* Index of the first element of the cell in
43 the Partition::elements array */
44             unsigned int first;
45             unsigned int max_ival;
46             unsigned int max_ival_count;
47         private:
48             bool in_splitting_queue;
49         public:
50             bool in_neighbour_heap;
51             /* Pointer to the next cell, null if this is the last one. */
52             Cell* next;
53             Cell* prev;
54             Cell* next_nonsingleton;
55             Cell* prev_nonsingleton;
56             unsigned int split_level;
57             bool is_unit()const {return (length == 1); }
58             bool is_in_splitting_queue()const {return (in_splitting_queue); }
59         };
60
61     };
62
63 private:
64     class RefInfo {
65     public:
66         unsigned int split_cell_first;
67         int prev_nonsingleton_first;
68         int next_nonsingleton_first;
69     };
70
71     std::vector<RefInfo> refinement_stack;
72
73     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;
157     Cell* get_cell(const unsigned int e) const {
158         assert(e < N);
159         return element_to_cell_map[e];
160     }
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
28 class Stats
29 {
30     friend class AbstractGraph;
31     BigNum group_size;
32     long double group_size_approx;
33     long unsigned int nof_nodes;
34     long unsigned int nof_leaf_nodes;
35     long unsigned int nof_bad_nodes;
36     long unsigned int nof_canupdates;
37     long unsigned int nof_generators;
38     unsigned long int max_level;
39     void reset()
40     {
41         group_size.assign(1);
42         group_size_approx = 1.0;
43         nof_nodes = 0;
44         nof_leaf_nodes = 0;
45         nof_bad_nodes = 0;
46         nof_canupdates = 0;
47         nof_generators = 0;
48         max_level = 0;
49     }
50 public:
51     Stats() { reset(); }
52     size_t print(FILE* const fp) const
53     {
54         size_t r = 0;
55         r += fprintf(fp, "Nodes: %lu\n", nof_nodes);
56         r += fprintf(fp, "Leaf nodes: %lu\n", nof_leaf_nodes);
57         r += fprintf(fp, "Bad nodes: %lu\n", nof_bad_nodes);
58         r += fprintf(fp, "Canrep updates: %lu\n", nof_canupdates);
59         r += fprintf(fp, "Generators: %lu\n", nof_generators);
60         r += fprintf(fp, "Max level: %lu\n", max_level);
61         r += fprintf(fp, "|Aut|: %lu\n", group_size.print(fp)+fprintf(fp, "\n"));
62         fflush(fp);
63         return r;
64     }
65     const BigNum& get_group_size() const {return group_size;}
66     long double get_group_size_approx() const {return group_size_approx;}
67     long unsigned int get_nof_nodes() const {return nof_nodes;}
68     long unsigned int get_nof_leaf_nodes() const {return nof_leaf_nodes;}
69     long unsigned int get_nof_bad_nodes() const {return nof_bad_nodes;}
70     long unsigned int get_nof_canupdates() const {return nof_canupdates;}
71     long unsigned int get_nof_generators() const {return nof_generators;}
72     unsigned long int get_max_level() const {return max_level;}
73 };
74
75 } // 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 #include <chrono>
23
24 namespace bliss {
25 class Timer
26 {
27     std::chrono::high_resolution_clock::time_point start_time_;
28 public:
29     Timer() {
30         reset();
31     }
32
33     void reset() {
34         start_time_ = std::chrono::high_resolution_clock::now();
35     }
36
37     double get_duration() const {
38         return std::chrono::duration_cast<std::chrono::duration<double>>
39             >(std::chrono::high_resolution_clock::now() - start_time_).count();
40     }
41 };
42
43 } // 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 class UIntSeqHash
24 {
25 protected:
26     unsigned int h;
27 public:
28     UIntSeqHash() {h = 0; }
29     UIntSeqHash(const UIntSeqHash &other) {h = other.h; }
30     UIntSeqHash& operator=(const UIntSeqHash &other) {h = other.h; return *this; }
31
32     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 Junntila
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 <cstdio>
24
25 namespace bliss {
26
27 size_t print_permutation(FILE* fp,
28                         const unsigned int N,
29                         const unsigned int* perm,
30                         const unsigned int offset = 0);
31
32 size_t print_permutation(FILE* fp,
33                         const std::vector<unsigned int>& perm,
34                         const unsigned int offset = 0);
35
36 bool is_permutation(const unsigned int N, const unsigned int* perm);
37
38 bool is_permutation(const std::vector<unsigned int>& perm);
39
40 } // namespace bliss
```

# Index

- ~BigNum
  - bliss::BigNum, [24](#)
- ~Digraph
  - bliss::Digraph, [28](#)
- ~Graph
  - bliss::Graph, [37](#)
- add\_edge
  - bliss::AbstractGraph, [18](#)
  - bliss::Digraph, [29](#)
  - bliss::Graph, [37](#)
- add\_vertex
  - bliss::AbstractGraph, [18](#)
  - bliss::Digraph, [29](#)
  - bliss::Graph, [38](#)
- assign
  - bliss::BigNum, [24](#)
- BacktrackPoint
  - bliss::Partition, [51](#)
- BigNum
  - bliss::BigNum, [24](#)
- bliss, [15](#)
  - is\_permutation, [16](#)
  - print\_permutation, [16](#)
- bliss::AbstractGraph, [17](#)
  - add\_edge, [18](#)
  - add\_vertex, [18](#)
  - canonical\_form, [18](#)
  - change\_color, [18](#)
  - find\_automorphisms, [19](#)
  - get\_color, [19](#)
  - get\_hash, [19](#)
  - get\_nof\_vertices, [19](#)
  - is\_automorphism, [20](#)
  - permute, [20](#)
  - set\_component\_recursion, [20](#)
  - set\_failure\_recording, [21](#)
  - set\_long\_prune\_activity, [21](#)
  - set\_verbose\_file, [21](#)
  - set\_verbose\_level, [22](#)
  - write\_dimacs, [22](#)
  - write\_dot, [22](#), [23](#)
- bliss::BigNum, [23](#)
  - ~BigNum, [24](#)
  - assign, [24](#)
  - BigNum, [24](#)
  - get\_factors, [24](#)
  - multiply, [24](#)
  - print, [24](#)
  - to\_string, [24](#)
- bliss::Digraph, [27](#)
  - ~Digraph, [28](#)
  - add\_edge, [29](#)
  - add\_vertex, [29](#)
  - canonical\_form, [29](#)
  - change\_color, [29](#)
  - cmp, [30](#)
  - copy, [30](#)
  - Digraph, [28](#)
  - find\_automorphisms, [30](#)
  - get\_color, [30](#)
  - get\_hash, [31](#)
  - get\_nof\_vertices, [31](#)
  - is\_automorphism, [31](#)
  - permute, [32](#)
  - read\_dimacs, [32](#)
  - set\_component\_recursion, [33](#)
  - set\_failure\_recording, [33](#)
  - set\_long\_prune\_activity, [33](#)
  - set\_splitting\_heuristic, [34](#)
  - set\_verbose\_file, [34](#)
  - set\_verbose\_level, [34](#)
  - shs\_f, [28](#)
  - shs\_fl, [28](#)
  - shs\_flm, [28](#)
  - shs\_fm, [28](#)
  - shs\_fs, [28](#)
  - shs\_fsm, [28](#)
  - SplittingHeuristic, [28](#)
  - write\_dimacs, [34](#)
  - write\_dot, [35](#)
- bliss::Graph, [35](#)
  - ~Graph, [37](#)
  - add\_edge, [37](#)
  - add\_vertex, [38](#)
  - canonical\_form, [38](#)
  - change\_color, [38](#)
  - cmp, [38](#)
  - copy, [39](#)
  - find\_automorphisms, [39](#)
  - get\_color, [39](#)
  - get\_hash, [39](#)
  - get\_nof\_vertices, [40](#)
  - Graph, [37](#)
  - is\_automorphism, [40](#)
  - permute, [40](#), [41](#)
  - read\_dimacs, [41](#)
  - set\_component\_recursion, [41](#)

- set\_failure\_recording, 42
- set\_long\_prune\_activity, 42
- set\_splitting\_heuristic, 42
- set\_verbose\_file, 42
- set\_verbose\_level, 43
- shs\_f, 37
- shs\_fl, 37
- shs\_flm, 37
- shs\_fm, 37
- shs\_fs, 37
- shs\_fsm, 37
- SplittingHeuristic, 37
- write\_dimacs, 43
- write\_dot, 43, 44
- bliss::Heap, 44
  - clear, 44
  - insert, 45
  - is\_empty, 45
  - remove, 45
  - size, 45
  - smallest, 45
- bliss::KQueue< Type >, 45
  - clear, 46
  - front, 46
  - init, 46
  - is\_empty, 47
  - KQueue, 46
  - pop\_back, 47
  - pop\_front, 47
  - push\_back, 47
  - push\_front, 47
  - size, 47
- bliss::Orbit, 48
  - get\_minimal\_representative, 49
  - init, 49
  - is\_minimal\_representative, 49
  - merge\_orbits, 49
  - nof\_orbits, 49
  - Orbit, 48
  - orbit\_size, 49
  - reset, 50
- bliss::Partition, 50
  - BacktrackPoint, 51
  - clear\_ivs, 51
  - get\_cell, 51
  - goto\_backtrack\_point, 51
  - individualize, 51
  - init, 52
  - is\_discrete, 52
  - print, 52
  - print\_signature, 52
  - set\_backtrack\_point, 52
- bliss::Partition::Cell, 26
  - is\_in\_splitting\_queue, 26
  - is\_unit, 26
- bliss::Stats, 53
  - get\_group\_size, 53
  - get\_group\_size\_approx, 53
  - get\_max\_level, 53
  - get\_nof\_bad\_nodes, 53
  - get\_nof\_canupdates, 54
  - get\_nof\_generators, 54
  - get\_nof\_leaf\_nodes, 54
  - get\_nof\_nodes, 54
  - print, 54
- bliss::Timer, 54
- bliss::UIntSeqHash, 55
  - cmp, 55
  - get\_value, 55
  - is\_equal, 56
  - is\_le, 56
  - is\_lt, 56
  - reset, 56
  - update, 56
- bliss\_graph\_struct, 25
- bliss\_stats\_struct, 25
- canonical\_form
  - bliss::AbstractGraph, 18
  - bliss::Digraph, 29
  - bliss::Graph, 38
- change\_color
  - bliss::AbstractGraph, 18
  - bliss::Digraph, 29
  - bliss::Graph, 38
- clear
  - bliss::Heap, 44
  - bliss::KQueue< Type >, 46
- clear\_ivs
  - bliss::Partition, 51
- cmp
  - bliss::Digraph, 30
  - bliss::Graph, 38
  - bliss::UIntSeqHash, 55
- copy
  - bliss::Digraph, 30
  - bliss::Graph, 39
- Digraph
  - bliss::Digraph, 28
- find\_automorphisms
  - bliss::AbstractGraph, 19
  - bliss::Digraph, 30
  - bliss::Graph, 39
- front
  - bliss::KQueue< Type >, 46
- get\_cell
  - bliss::Partition, 51
- get\_color
  - bliss::AbstractGraph, 19
  - bliss::Digraph, 30
  - bliss::Graph, 39
- get\_factors
  - bliss::BigNum, 24
- get\_group\_size



- bliss::Stats, 53
- get\_group\_size\_approx
  - bliss::Stats, 53
- get\_hash
  - bliss::AbstractGraph, 19
  - bliss::Digraph, 31
  - bliss::Graph, 39
- get\_max\_level
  - bliss::Stats, 53
- get\_minimal\_representative
  - bliss::Orbit, 49
- get\_nof\_bad\_nodes
  - bliss::Stats, 53
- get\_nof\_canupdates
  - bliss::Stats, 54
- get\_nof\_generators
  - bliss::Stats, 54
- get\_nof\_leaf\_nodes
  - bliss::Stats, 54
- get\_nof\_nodes
  - bliss::Stats, 54
- get\_nof\_vertices
  - bliss::AbstractGraph, 19
  - bliss::Digraph, 31
  - bliss::Graph, 40
- get\_value
  - bliss::UIntSeqHash, 55
- goto\_backtrack\_point
  - bliss::Partition, 51
- Graph
  - bliss::Graph, 37
- individualize
  - bliss::Partition, 51
- init
  - bliss::KQueue< Type >, 46
  - bliss::Orbit, 49
  - bliss::Partition, 52
- insert
  - bliss::Heap, 45
- is\_automorphism
  - bliss::AbstractGraph, 20
  - bliss::Digraph, 31
  - bliss::Graph, 40
- is\_discrete
  - bliss::Partition, 52
- is\_empty
  - bliss::Heap, 45
  - bliss::KQueue< Type >, 47
- is\_equal
  - bliss::UIntSeqHash, 56
- is\_in\_splitting\_queue
  - bliss::Partition::Cell, 26
- is\_le
  - bliss::UIntSeqHash, 56
- is\_lt
  - bliss::UIntSeqHash, 56
- is\_minimal\_representative
  - bliss::Orbit, 49
- is\_permutation
  - bliss, 16
- is\_unit
  - bliss::Partition::Cell, 26
- KQueue
  - bliss::KQueue< Type >, 46
- merge\_orbits
  - bliss::Orbit, 49
- multiply
  - bliss::BigNum, 24
- nof\_orbits
  - bliss::Orbit, 49
- Orbit
  - bliss::Orbit, 48
- orbit\_size
  - bliss::Orbit, 49
- permute
  - bliss::AbstractGraph, 20
  - bliss::Digraph, 32
  - bliss::Graph, 40, 41
- pop\_back
  - bliss::KQueue< Type >, 47
- pop\_front
  - bliss::KQueue< Type >, 47
- print
  - bliss::BigNum, 24
  - bliss::Partition, 52
  - bliss::Stats, 54
- print\_permutation
  - bliss, 16
- print\_signature
  - bliss::Partition, 52
- push\_back
  - bliss::KQueue< Type >, 47
- push\_front
  - bliss::KQueue< Type >, 47
- read\_dimacs
  - bliss::Digraph, 32
  - bliss::Graph, 41
- remove
  - bliss::Heap, 45
- reset
  - bliss::Orbit, 50
  - bliss::UIntSeqHash, 56
- set\_backtrack\_point
  - bliss::Partition, 52
- set\_component\_recursion
  - bliss::AbstractGraph, 20
  - bliss::Digraph, 33
  - bliss::Graph, 41
- set\_failure\_recording
  - bliss::AbstractGraph, 21
  - bliss::Digraph, 33

- bliss::Graph, [42](#)
- set\_long\_prune\_activity
  - bliss::AbstractGraph, [21](#)
  - bliss::Digraph, [33](#)
  - bliss::Graph, [42](#)
- set\_splitting\_heuristic
  - bliss::Digraph, [34](#)
  - bliss::Graph, [42](#)
- set\_verbose\_file
  - bliss::AbstractGraph, [21](#)
  - bliss::Digraph, [34](#)
  - bliss::Graph, [42](#)
- set\_verbose\_level
  - bliss::AbstractGraph, [22](#)
  - bliss::Digraph, [34](#)
  - bliss::Graph, [43](#)
- shs\_f
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- shs\_fl
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- shs\_flm
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- shs\_fm
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- shs\_fs
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- shs\_fsm
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- size
  - bliss::Heap, [45](#)
  - bliss::KQueue< Type >, [47](#)
- smallest
  - bliss::Heap, [45](#)
- SplittingHeuristic
  - bliss::Digraph, [28](#)
  - bliss::Graph, [37](#)
- src/abstractgraph.hh, [57](#)
- src/bignum.hh, [60](#)
- src/bliss\_C.h, [62](#)
- src/defs.hh, [64](#)
- src/digraph.hh, [65](#)
- src/graph.hh, [67](#)
- src/heap.hh, [68](#)
- src/kqueue.hh, [69](#)
- src/orbit.hh, [71](#)
- src/partition.hh, [72](#)
- src/stats.hh, [75](#)
- src/timer.hh, [76](#)
- src/uintseqhash.hh, [76](#)
- src/utils.hh, [77](#)
- to\_string
  - bliss::BigNum, [24](#)
- update
  - bliss::UIntSeqHash, [56](#)
- write\_dimacs
  - bliss::AbstractGraph, [22](#)
  - bliss::Digraph, [34](#)
  - bliss::Graph, [43](#)
- write\_dot
  - bliss::AbstractGraph, [22](#), [23](#)
  - bliss::Digraph, [35](#)
  - bliss::Graph, [43](#), [44](#)