

The bliss C++ API

0.77 (Debian 0.77-3)

Generated by Doxygen 1.9.8

1 Outline	1
1.1 The C language API	1
2 The bliss executable	3
3 Namespace Index	7
3.1 Namespace List	7
4 Hierarchical Index	9
4.1 Class Hierarchy	9
5 Class Index	11
5.1 Class List	11
6 File Index	13
6.1 File List	13
7 Namespace Documentation	15
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
8 Class Documentation	17
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()	19
8.1.2.4 change_color()	19
8.1.2.5 find_automorphisms()	19
8.1.2.6 get_color()	20
8.1.2.7 get_hash()	20
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]	21
8.1.2.12 permute() [2/2]	21
8.1.2.13 set_component_recursion()	21
8.1.2.14 set_failure_recording()	21
8.1.2.15 set_long_prune_activity()	22

8.1.2.16 set_verbose_file()	22
8.1.2.17 set_verbose_level()	22
8.1.2.18 write_dimacs()	22
8.1.2.19 write_dot() [1/2]	23
8.1.2.20 write_dot() [2/2]	23
8.2 bliss::BigNum Class Reference	23
8.2.1 Detailed Description	24
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()	25
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()	29
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]	31
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>	32
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>	33
8.6.4.20 <code>set_verbose_file()</code>	33
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]	34
8.6.4.24 <code>write_dot()</code> [2/2]	34
8.7 bliss::Graph Class Reference	35
8.7.1 Detailed Description	36
8.7.2 Member Enumeration Documentation	36
8.7.2.1 <code>SplittingHeuristic</code>	36
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>	37
8.7.4.2 <code>add_vertex()</code>	37
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>	38
8.7.4.6 <code>copy()</code>	38
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>	39
8.7.4.10 <code>get_nof_vertices()</code>	39
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]	40
8.7.4.14 <code>permute()</code> [2/2]	40
8.7.4.15 <code>read_dimacs()</code>	40
8.7.4.16 <code>set_component_recursion()</code>	41
8.7.4.17 <code>set_failure_recording()</code>	41
8.7.4.18 <code>set_long_prune_activity()</code>	41
8.7.4.19 <code>set_splitting_heuristic()</code>	42

8.7.4.20 <code>set_verbose_file()</code>	42
8.7.4.21 <code>set_verbose_level()</code>	42
8.7.4.22 <code>write_dimacs()</code>	42
8.7.4.23 <code>write_dot()</code> [1/2]	42
8.7.4.24 <code>write_dot()</code> [2/2]	43
8.8 <code>bliss::Heap</code> Class Reference	43
8.8.1 Detailed Description	43
8.8.2 Member Function Documentation	44
8.8.2.1 <code>clear()</code>	44
8.8.2.2 <code>insert()</code>	44
8.8.2.3 <code>is_empty()</code>	44
8.8.2.4 <code>remove()</code>	44
8.8.2.5 <code>size()</code>	44
8.8.2.6 <code>smallest()</code>	44
8.9 <code>bliss::KQueue< Type ></code> Class Template Reference	45
8.9.1 Detailed Description	45
8.9.2 Constructor & Destructor Documentation	45
8.9.2.1 <code>KQueue()</code>	45
8.9.3 Member Function Documentation	45
8.9.3.1 <code>clear()</code>	45
8.9.3.2 <code>front()</code>	46
8.9.3.3 <code>init()</code>	46
8.9.3.4 <code>is_empty()</code>	46
8.9.3.5 <code>pop_back()</code>	46
8.9.3.6 <code>pop_front()</code>	46
8.9.3.7 <code>push_back()</code>	46
8.9.3.8 <code>push_front()</code>	47
8.9.3.9 <code>size()</code>	47
8.10 <code>bliss::Orbit</code> Class Reference	47
8.10.1 Detailed Description	47
8.10.2 Constructor & Destructor Documentation	48
8.10.2.1 <code>Orbit()</code>	48
8.10.3 Member Function Documentation	48
8.10.3.1 <code>get_minimal_representative()</code>	48
8.10.3.2 <code>init()</code>	48
8.10.3.3 <code>is_minimal_representative()</code>	48
8.10.3.4 <code>merge_orbits()</code>	48
8.10.3.5 <code>nof_orbits()</code>	49
8.10.3.6 <code>orbit_size()</code>	49
8.10.3.7 <code>reset()</code>	49
8.11 <code>bliss::Partition</code> Class Reference	49
8.11.1 Detailed Description	50

8.11.2 Member Typedef Documentation	50
8.11.2.1 BacktrackPoint	50
8.11.3 Member Function Documentation	50
8.11.3.1 clear_ivs()	50
8.11.3.2 get_cell()	50
8.11.3.3 goto_backtrack_point()	50
8.11.3.4 individualize()	50
8.11.3.5 init()	51
8.11.3.6 is_discrete()	51
8.11.3.7 print()	51
8.11.3.8 print_signature()	51
8.11.3.9 set_backtrack_point()	51
8.12 bliss::Stats Class Reference	52
8.12.1 Detailed Description	52
8.12.2 Member Function Documentation	52
8.12.2.1 get_group_size()	52
8.12.2.2 get_group_size_approx()	52
8.12.2.3 get_max_level()	52
8.12.2.4 get_nof_bad_nodes()	52
8.12.2.5 get_nof_canupdates()	53
8.12.2.6 get_nof_generators()	53
8.12.2.7 get_nof_leaf_nodes()	53
8.12.2.8 get_nof_nodes()	53
8.12.2.9 print()	53
8.13 bliss::Timer Class Reference	53
8.13.1 Detailed Description	54
8.14 bliss::UIntSeqHash Class Reference	54
8.14.1 Detailed Description	54
8.14.2 Member Function Documentation	54
8.14.2.1 cmp()	54
8.14.2.2 get_value()	54
8.14.2.3 is_equal()	55
8.14.2.4 is_le()	55
8.14.2.5 is_lt()	55
8.14.2.6 reset()	55
8.14.2.7 update()	55
9 File Documentation	57
9.1 abstractgraph.hh	57
9.2 bignum.hh	60
9.3 src/bliss_C.h File Reference	62
9.3.1 Detailed Description	63

9.3.2 Function Documentation	64
9.3.2.1 bliss_add_edge()	64
9.3.2.2 bliss_add_vertex()	64
9.3.2.3 bliss_cmp()	64
9.3.2.4 bliss_find_automorphisms()	64
9.3.2.5 bliss_find_canonical_labeling()	65
9.3.2.6 bliss_get_nof_vertices()	65
9.3.2.7 bliss_hash()	65
9.3.2.8 bliss_new()	65
9.3.2.9 bliss_permute()	65
9.3.2.10 bliss_read_dimacs()	66
9.3.2.11 bliss_release()	66
9.3.2.12 bliss_write_dimacs()	66
9.3.2.13 bliss_write_dot()	66
9.4 bliss_C.h	66
9.5 src/defs.hh File Reference	67
9.5.1 Detailed Description	68
9.6 defs.hh	68
9.7 digraph.hh	69
9.8 graph.hh	71
9.9 heap.hh	72
9.10 kqueue.hh	73
9.11 orbit.hh	75
9.12 partition.hh	76
9.13 stats.hh	79
9.14 timer.hh	80
9.15 uintseqhash.hh	80
9.16 src/utils.hh File Reference	81
9.16.1 Detailed Description	82
9.17 utils.hh	82
Index	83

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:

bliss	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	43
bliss::KQueue< Type >	45
bliss::KQueue< bliss::Partition::Cell * >	45
bliss::Orbit	47
bliss::Partition	49
bliss::Stats	52
bliss::Timer	53
bliss::UIntSeqHash	54

Chapter 5

Class Index

5.1 Class List

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

bliss::AbstractGraph	An abstract base class for different types of graphs	17
bliss::BigNum	A simple wrapper class for non-negative big integers (or approximation of them)	23
bliss_graph_struct	The true bliss graph is hiding in this struct	25
bliss_stats_struct	The C API version of the statistics returned by the bliss search algorithm	25
bliss::Partition::Cell	Data structure for holding information about a cell in a Partition	26
bliss::Digraph	The class for directed, vertex colored graphs	27
bliss::Graph	The class for undirected, vertex colored graphs	35
bliss::Heap	A min-heap of unsigned integers	43
bliss::KQueue< Type >	A simple implementation of queues with fixed maximum capacity	45
bliss::Orbit	A class for representing orbit information	47
bliss::Partition	A class for refinable, backtrackable ordered partitions	49
bliss::Stats	Statistics returned by the bliss search algorithm	52
bliss::Timer	A simple helper class for measuring elapsed time	53
bliss::UIntSeqHash	A updatable hash for sequences of unsigned ints	54

Chapter 6

File Index

6.1 File List

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

src/ abstractgraph.hh	57
src/ bignum.hh	60
src/ bliss_C.h	
The bliss C API	62
src/ defs.hh	
Some common definitions	67
src/ digraph.hh	69
src/ graph.hh	71
src/ heap.hh	72
src/ kqueue.hh	73
src/ orbit.hh	75
src/ partition.hh	76
src/ stats.hh	79
src/ timer.hh	80
src/ uintseqhash.hh	80
src/ utils.hh	
Some small utilities	81

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

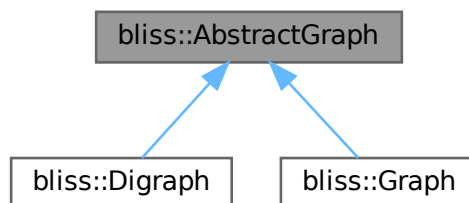
Class Documentation

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 $\{0, \dots, \text{get_nof_vertices}() - 1\}$). 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 string 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:

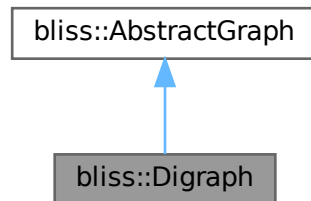
- [src/partition.hh](#)

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

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

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 doty 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:

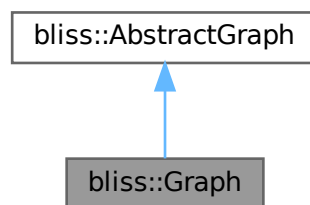
- src/digraph.hh
- src/digraph.cc

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 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/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 ( ) const
```

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 ( ) const
```

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 ( ) const
```

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 Cell
<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

```
00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 namespace bliss {
00023     class AbstractGraph;
00024 }
00025
00026 #include <cstdio>
00027 #include <functional>
00028 #include <vector>
00029 #include <bliss/stats.hh>
00030 #include <bliss/kqueue.hh>
00031 #include <bliss/heap.hh>
00032 #include <bliss/orbit.hh>
00033 #include <bliss/partition.hh>
00034 #include <bliss/uintseqhash.hh>
00035
00036 namespace bliss {
00037
00038     class AbstractGraph
00039     {
00040     public:
00041         AbstractGraph();
00042         virtual ~AbstractGraph();
00043
00044         void set_verbose_level(const unsigned int level);
00045
00046         void set_verbose_file(FILE * const fp);
00047
00048         virtual unsigned int add_vertex(const unsigned int color = 0) = 0;
00049
00050         virtual void add_edge(const unsigned int source, const unsigned int target) = 0;
00051
00052         virtual unsigned int get_color(const unsigned int vertex) const = 0;
00053
00054         virtual void change_color(const unsigned int vertex, const unsigned int color) = 0;
00055     };
00056 }
```

```

00090
00096 void set_failure_recording(const bool active) {
00097     assert(not in_search);
00098     opt_use_failure_recording = active;
00099 }
00100
00110 void set_component_recursion(const bool active) {
00111     assert(not in_search);
00112     opt_use_comprec = active;
00113 }
00114
00115
00116
00120 virtual unsigned int get_nof_vertices() const = 0;
00121
00128 virtual AbstractGraph* permute(const unsigned int* const perm) const = 0;
00129
00136 virtual AbstractGraph* permute(const std::vector<unsigned int>& perm) const = 0;
00137
00142 virtual bool is_automorphism(unsigned int* const perm) const = 0;
00143
00148 virtual bool is_automorphism(const std::vector<unsigned int>& perm) const = 0;
00149
00174 void find_automorphisms(Stats& stats,
00175     const std::function<void(unsigned int n, const unsigned int* aut)>& report = nullptr,
00176     const std::function<bool()>& terminate = nullptr);
00177
00203 const unsigned int* canonical_form(Stats& stats,
00204     const std::function<void(unsigned int n, const unsigned int* aut)>& report =
00205     nullptr,
00206     const std::function<bool()>& terminate = nullptr);
00207
00216 virtual void write_dimacs(FILE * const fp) = 0;
00217
00222 virtual void write_dot(FILE * const fp) = 0;
00223
00229 virtual void write_dot(const char * const file_name) = 0;
00230
00235 virtual unsigned int get_hash() = 0;
00236
00247 void set_long_prune_activity(const bool active) {
00248     assert(not in_search);
00249     opt_use_long_prune = active;
00250 }
00251
00252
00253
00254 protected:
00257 unsigned int verbose_level;
00258
00261 FILE *verbstr;
00262
00265 Partition p;
00266
00271 bool in_search;
00272
00276 bool opt_use_failure_recording;
00277
00278 /* The "tree-specific" invariant value for the point when current path
00279  * got different from the first path */
00280 unsigned int failure_recording_fp_deviation;
00281
00285 bool opt_use_comprec;
00286
00287
00288 unsigned int refine_current_path_certificate_index;
00289 bool refine_compare_certificate;
00290 bool refine_equal_to_first;
00291 unsigned int refine_first_path_subcertificate_end;
00292 int refine_cmp_to_best;
00293 unsigned int refine_best_path_subcertificate_end;
00294
00295 static const unsigned int CERT_SPLIT = 0; //UINT_MAX;
00296 static const unsigned int CERT_EDGE = 1; //UINT_MAX-1;
00301 void cert_add(const unsigned int v1,
00302     const unsigned int v2,
00303     const unsigned int v3);
00304
00310 void cert_add_redundant(const unsigned int x,
00311     const unsigned int y,
00312     const unsigned int z);
00313
00317 bool opt_use_long_prune;
00322 static const unsigned int long_prune_options_max_mem = 50;
00327 static const unsigned int long_prune_options_max_stored_auts = 100;
00328
00329 unsigned int long_prune_max_stored_autss;

```

```

00330     std::vector<std::vector<bool>*> long_prune_fixed;
00331     std::vector<std::vector<bool>*> long_prune_mcrcs;
00332     std::vector<bool> long_prune_temp;
00333     unsigned int long_prune_begin;
00334     unsigned int long_prune_end;
00338     void long_prune_init();
00342     void long_prune_deallocate();
00343     void long_prune_add_automorphism(const unsigned int *aut);
00344     std::vector<bool>& long_prune_get_fixed(const unsigned int index);
00345     std::vector<bool>& long_prune_allocget_fixed(const unsigned int index);
00346     std::vector<bool>& long_prune_get_mcrcs(const unsigned int index);
00347     std::vector<bool>& long_prune_allocget_mcrcs(const unsigned int index);
00352     void long_prune_swap(const unsigned int i, const unsigned int j);
00353
00354     /*
00355      * Data structures and routines for refining the partition p into equitable
00356      */
00357     Heap neighbour_heap;
00358     virtual bool split_neighbourhood_of_unit_cell(Partition::Cell * const) = 0;
00359     virtual bool split_neighbourhood_of_cell(Partition::Cell * const) = 0;
00360     void refine_to_equitable();
00361     void refine_to_equitable(Partition::Cell * const unit_cell);
00362     void refine_to_equitable(Partition::Cell * const unit_cell1,
00363                             Partition::Cell * const unit_cell2);
00364
00365     bool do_refine_to_equitable();
00371
00372     unsigned int eqref_max_certificate_index;
00377     bool compute_eqref_hash;
00378     UIntSeqHash eqref_hash;
00379
00380
00385     virtual bool is_equitable() const = 0;
00386
00387     unsigned int *first_path_labeling;
00388     unsigned int *first_path_labeling_inv;
00389     Orbit first_path_orbits;
00390     unsigned int *first_path_automorphism;
00391
00392     unsigned int *best_path_labeling;
00393     unsigned int *best_path_labeling_inv;
00394     Orbit best_path_orbits;
00395     unsigned int *best_path_automorphism;
00396
00397     void update_labeling(unsigned int * const lab);
00398     void update_labeling_and_its_inverse(unsigned int * const lab,
00399                                       unsigned int * const lab_inv);
00400     void update_orbit_information(Orbit &o, const unsigned int *perm);
00401
00402     void reset_permutation(unsigned int *perm);
00403
00404     std::vector<unsigned int> certificate_current_path;
00405     std::vector<unsigned int> certificate_first_path;
00406     std::vector<unsigned int> certificate_best_path;
00407
00408     unsigned int certificate_index;
00409     virtual void initialize_certificate() = 0;
00410
00411     /* Remove duplicates from seq.
00412      * If m is the largest element in seq, then m < tmp.size() must hold.
00413      * All entries in tmp must be false when called.
00414      * Under that condition, all entries in tmp are false on exit as well.
00415      */
00416     static void remove_duplicates(std::vector<unsigned int>& seq, std::vector<bool>& tmp);
00417
00418     virtual void remove_duplicate_edges() = 0;
00419     virtual void make_initial_equitable_partition() = 0;
00420     virtual Partition::Cell* find_next_cell_to_be_split(Partition::Cell *cell) = 0;
00421
00422
00427     typedef struct {
00428         unsigned int splitting_element;
00429         unsigned int certificate_index;
00430         unsigned int subcertificate_length;
00431         UIntSeqHash eqref_hash;
00432     } PathInfo;
00433
00434     void search(const bool canonical, Stats &stats,
00435               const std::function<void(unsigned int n, const unsigned int* aut)>& report_function =
00436 nullptr,
00437               const std::function<bool()>& terminate = nullptr);
00438
00439     /*
00440      *
00441      * Nonuniform component recursion (NUCR)

```

```

00442     *
00443     */
00444
00445     /* The currently traversed component */
00446     unsigned int cr_level;
00447
00451     class CR_CEP {
00452     public:
00454         unsigned int creation_level;
00457         unsigned int discrete_cell_limit;
00459         unsigned int next_cr_level;
00461         unsigned int next_cep_index;
00462         bool first_checked;
00463         bool best_checked;
00464     };
00468     std::vector<CR_CEP> cr_cep_stack;
00469
00481     virtual bool nucr_find_first_component(const unsigned int level) = 0;
00482     virtual bool nucr_find_first_component(const unsigned int level,
00483                                           std::vector<unsigned int>& component,
00484                                           unsigned int& component_elements,
00485                                           Partition::Cell*& sh_return) = 0;
00490     std::vector<unsigned int> cr_component;
00494     unsigned int cr_component_elements;
00495
00496
00497
00498
00499
00500
00501 };
00502
00503 } // namespace bliss

```

9.2 bignum.hh

```

00001 #pragma once
00002
00003 /*
00004     Copyright (c) 2003-2021 Tommi Junttila
00005     Released under the GNU Lesser General Public License version 3.
00006
00007     This file is part of bliss.
00008
00009     bliss is free software: you can redistribute it and/or modify
00010     it under the terms of the GNU Lesser General Public License as published by
00011     the Free Software Foundation, version 3 of the License.
00012
00013     bliss is distributed in the hope that it will be useful,
00014     but WITHOUT ANY WARRANTY; without even the implied warranty of
00015     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016     GNU Lesser General Public License for more details.
00017
00018     You should have received a copy of the GNU Lesser General Public License
00019     along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #if defined(BLISS_USE_GMP)
00023 #include <gmp.h>
00024 #else
00025 #include <vector>
00026 #include <string>
00027 #endif
00028
00029 #include <cstdlib>
00030 #include <cstdio>
00031 #include <bliss/defs.hh>
00032
00033
00034 namespace bliss {
00035
00048 #if defined(BLISS_USE_GMP)
00049
00050 class BigNum
00051 {
00052     mpz_t v;
00053     public:
00057     BigNum() {mpz_init(v); }
00058
00062     ~BigNum() {mpz_clear(v); }
00063
00067     void assign(unsigned int n) {mpz_set_ui(v, n); }
00068

```

```

00072 void multiply(unsigned int n) {mpz_mul_ui(v, v, n); }
00073
00077 size_t print(FILE* const fp) const {return mpz_out_str(fp, 10, v); }
00078
00084 void get(mpz_t& result) const {mpz_set(result, v); }
00085 };
00086
00087 #elif defined(BLISS_BIGNUM_APPROX)
00088
00089 class BigNum
00090 {
00091     long double v;
00092 public:
00096     BigNum(): v(0.0) {}
00097
00101 void assign(unsigned int n) {v = (long double)n; }
00102
00106 void multiply(unsigned int n) {v *= (long double)n; }
00107
00111 size_t print(FILE* const fp) const {return fprintf(fp, "%Lg", v); }
00112 };
00113
00114 #else
00115
00116 class BigNum
00117 {
00118     /* This is a version that does not actually compute the number
00120      * but rather only stores the factor integers.
00121      */
00122     std::vector<unsigned int> factors;
00123 public:
00127     BigNum() {
00128         factors.push_back(0);
00129     }
00130
00134     ~BigNum() {}
00135
00139 void assign(unsigned int n) {
00140     factors.clear();
00141     factors.push_back(n);
00142 }
00143
00147 void multiply(unsigned int n) {
00148     factors.push_back(n);
00149 }
00150
00156 size_t print(FILE* const fp) const {
00157     assert(not factors.empty());
00158     size_t r = 0;
00159     /*
00160     const char* sep = "";
00161     for(int v: factors) {
00162         r += fprintf(fp, "%s%d", sep, v);
00163         sep = "*";
00164     }
00165     */
00166     for(char d: to_string())
00167         r += fprintf(fp, "%c", d);
00168     return r;
00169 }
00170
00174 const std::vector<unsigned int>& get_factors() const {
00175     return factors;
00176 }
00177
00182 std::string to_string() const {
00183     // Base 100 result, in reverse order
00184     std::vector<unsigned int> result;
00185     result.push_back(1);
00186     for(unsigned int factor: factors) {
00187         std::vector<unsigned int> summand;
00188         unsigned int offset = 0;
00189         while(factor != 0) {
00190             const unsigned int multiplier = factor % 100;
00191             // Multiplication by a "digit"
00192             std::vector<unsigned int> product;
00193             for(unsigned int i = 0; i < offset; i++)
00194                 product.push_back(0);
00195             unsigned int carry = 0;
00196             for(unsigned int digit: result) {
00197                 unsigned int v = digit * multiplier + carry;
00198                 product.push_back(v % 100);
00199                 carry = v / 100;
00200             }
00201             if(carry > 0)
00202                 product.push_back(carry);

```

```

00203     // Addition
00204     add(summand, product);
00205     // Next "digit" in factor
00206     factor = factor / 100;
00207     offset++;
00208     }
00209     result = summand;
00210     }
00211     return _string(result);
00212 }
00213
00214 protected:
00215 static void add(std::vector<unsigned int>& num, const std::vector<unsigned int>& summand) {
00216     unsigned int carry = 0;
00217     unsigned int i = 0;
00218     while(i < num.size() and i < summand.size()) {
00219         const unsigned int v = carry + num[i] + summand[i];
00220         num[i] = v % 100;
00221         carry = v / 100;
00222         i++;
00223     }
00224     while(i < summand.size()) {
00225         const unsigned int v = carry + summand[i];
00226         num.push_back(v % 100);
00227         carry = v / 100;
00228         i++;
00229     }
00230     while(i < num.size()) {
00231         const unsigned int v = carry + num[i];
00232         num[i] = v % 100;
00233         carry = v / 100;
00234         i++;
00235     }
00236     if(carry != 0)
00237         num.push_back(carry);
00238 }
00239
00240
00241 static std::string _string(const std::vector<unsigned int> n) {
00242     const char digits[] = {'0','1','2','3','4','5','6','7','8','9'};
00243     std::string r;
00244     bool first = true;
00245     for(auto it = n.crbegin(); it != n.crend(); it++) {
00246         unsigned int digit = *it;
00247         unsigned int high = digit / 10;
00248         if(not first or high > 0)
00249             r.push_back(digits[high]);
00250         first = false;
00251         r.push_back(digits[digit % 10]);
00252     }
00253     return r;
00254 }
00255 };
00256 };
00257
00258 #endif
00259
00260 } //namespace bliss

```

9.3 src/bliss_C.h File Reference

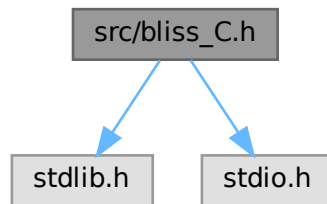
The bliss C API.

```

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

```

Include dependency graph for bliss_C.h:



Classes

- struct [bliss_stats_struct](#)

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

Typedefs

- typedef struct [bliss_graph_struct](#) **BlissGraph**

The true bliss graph is hiding behind this typedef.

- typedef struct [bliss_stats_struct](#) **BlissStats**

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

Functions

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

9.3.1 Detailed Description

The bliss C API.

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

9.3.2 Function Documentation

9.3.2.1 bliss_add_edge()

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

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

9.3.2.2 bliss_add_vertex()

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

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

9.3.2.3 bliss_cmp()

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

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

9.3.2.4 bliss_find_automorphisms()

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

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

9.3.2.5 bliss_find_canonical_labeling()

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

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

9.3.2.6 bliss_get_nof_vertices()

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

Return the number of vertices in the graph.

9.3.2.7 bliss_hash()

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

Get a hash value for the graph.

9.3.2.8 bliss_new()

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

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

9.3.2.9 bliss_permute()

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

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

9.3.2.10 bliss_read_dimacs()

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

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

9.3.2.11 bliss_release()

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

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

9.3.2.12 bliss_write_dimacs()

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

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

9.3.2.13 bliss_write_dot()

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

Print the graph in graphviz dot format.

9.4 bliss_C.h

[Go to the documentation of this file.](#)

```
00001 #ifndef BLISS_C_H
00002 #define BLISS_C_H
00003
00004 /*
00005  Copyright (c) 2003-2021 Tommi Junttila
00006  Released under the GNU Lesser General Public License version 3.
00007
00008  This file is part of bliss.
00009
00010  bliss is free software: you can redistribute it and/or modify
00011  it under the terms of the GNU Lesser General Public License as published by
00012  the Free Software Foundation, version 3 of the License.
00013
00014  bliss is distributed in the hope that it will be useful,
00015  but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017  GNU Lesser General Public License for more details.
00018
00019  You should have received a copy of the GNU Lesser General Public License
00020  along with bliss. If not, see <http://www.gnu.org/licenses/>.
```

```

00021 */
00022
00032 #include <stdlib.h>
00033 #include <stdio.h>
00034
00035
00039 typedef struct bliss_graph_struct BlissGraph;
00040
00041
00046 typedef struct bliss_stats_struct
00047 {
00052     long double group_size_approx;
00054     long unsigned int nof_nodes;
00056     long unsigned int nof_leaf_nodes;
00058     long unsigned int nof_bad_nodes;
00060     long unsigned int nof_canupdates;
00062     long unsigned int nof_generators;
00064     unsigned long int max_level;
00065 } BlissStats;
00066
00067
00073 BlissGraph *bliss_new(const unsigned int N);
00074
00075
00084 BlissGraph *bliss_read_dimacs(FILE *fp);
00085
00086
00093 void bliss_write_dimacs(BlissGraph *graph, FILE *fp);
00094
00095
00102 void bliss_release(BlissGraph *graph);
00103
00104
00108 void bliss_write_dot(BlissGraph *graph, FILE *fp);
00109
00110
00114 unsigned int bliss_get_nof_vertices(BlissGraph *graph);
00115
00116
00122 unsigned int bliss_add_vertex(BlissGraph *graph, unsigned int c);
00123
00124
00132 void bliss_add_edge(BlissGraph *graph, unsigned int v1, unsigned int v2);
00133
00134
00143 int bliss_cmp(BlissGraph *graph1, BlissGraph *graph2);
00144
00145
00149 unsigned int bliss_hash(BlissGraph *graph);
00150
00151
00159 BlissGraph *bliss_permute(BlissGraph *graph, const unsigned int *perm);
00160
00161
00177 void
00178 bliss_find_automorphisms(BlissGraph *graph,
00179                          void (*hook)(void *user_param,
00180                                         unsigned int N,
00181                                         const unsigned int *aut),
00182                          void *hook_user_param,
00183                          BlissStats *stats);
00184
00185
00197 const unsigned int *
00198 bliss_find_canonical_labeling(BlissGraph *graph,
00199                               void (*hook)(void *user_param,
00200                                              unsigned int N,
00201                                              const unsigned int *aut),
00202                               void *hook_user_param,
00203                               BlissStats *stats);
00204
00205 #endif

```

9.5 src/defs.hh File Reference

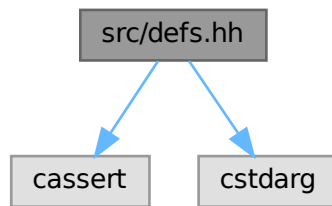
Some common definitions.

```

#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.](#)

```

00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #include <cassert>
00023 #include <cstdarg>
00024
00029 #define BLISS_VERSION "0.77"
00030 #define BLISS_VERSION_MAJOR 0
00031 #define BLISS_VERSION_MINOR 77
00032

```

```

00033 namespace bliss {
00034
00036 static const char * const version = "0.77";
00037
00038
00039
00040 #if defined(BLISS_DEBUG)
00041 #define BLISS_CONSISTENCY_CHECKS
00042 #define BLISS_EXPENSIVE_CONSISTENCY_CHECKS
00043 #endif
00044
00045
00046 #if defined(BLISS_CONSISTENCY_CHECKS)
00047 /* Force a check that the found automorphisms are valid */
00048 #define BLISS_VERIFY_AUTOMORPHISMS
00049 #endif
00050
00051
00052 #if defined(BLISS_CONSISTENCY_CHECKS)
00053 /* Force a check that the generated partitions are equitable */
00054 #define BLISS_VERIFY_EQUITABLEDNESS
00055 #endif
00056
00057 } // namespace bliss
00058
00059

```

9.7 digraph.hh

```

00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #include <bliss/abstractgraph.hh>
00023
00024 namespace bliss {
00025
00031 class Digraph : public AbstractGraph
00032 {
00033 public:
00042     typedef enum {
00046         shs_f = 0,
00049         shs_fs,
00052         shs_fl,
00056         shs_fm,
00060         shs_fsm,
00064         shs_flm
00065     } SplittingHeuristic;
00066
00067 protected:
00068     class Vertex {
00069     public:
00070         Vertex();
00071         ~Vertex();
00072         void add_edge_to(const unsigned int dest_vertex);
00073         void add_edge_from(const unsigned int source_vertex);
00074         void remove_duplicate_edges(std::vector<bool>& tmp);
00075         void sort_edges();
00076         unsigned int color;
00077         std::vector<unsigned int> edges_out;
00078         std::vector<unsigned int> edges_in;
00079         unsigned int nof_edges_in() const {return edges_in.size(); }
00080         unsigned int nof_edges_out() const {return edges_out.size(); }
00081     };
00082     std::vector<Vertex> vertices;
00083     void remove_duplicate_edges();

```

```

00084
00090 static unsigned int vertex_color_invariant(const Digraph* const g,
00091                                           const unsigned int v);
00098 static unsigned int indegree_invariant(const Digraph* const g,
00099                                       const unsigned int v);
00106 static unsigned int outdegree_invariant(const Digraph* const g,
00107                                       const unsigned int v);
00113 static unsigned int selfloop_invariant(const Digraph* const g,
00114                                       const unsigned int v);
00115
00120 bool refine_according_to_invariant(unsigned int (*inv)(const Digraph* const g,
00121                                                       const unsigned int v));
00122
00123 /*
00124  * Routines needed when refining the partition p into equitable
00125  */
00126 bool split_neighbourhood_of_unit_cell(Partition::Cell* const);
00127 bool split_neighbourhood_of_cell(Partition::Cell* const);
00128
00129
00133 bool is_equitable() const;
00134
00135 /* Splitting heuristics, documented in more detail in the cc-file. */
00136 SplittingHeuristic sh;
00137 Partition::Cell* find_next_cell_to_be_split(Partition::Cell *cell);
00138 Partition::Cell* sh_first();
00139 Partition::Cell* sh_first_smallest();
00140 Partition::Cell* sh_first_largest();
00141 Partition::Cell* sh_first_max_neighbours();
00142 Partition::Cell* sh_first_smallest_max_neighbours();
00143 Partition::Cell* sh_first_largest_max_neighbours();
00144
00145 /* A data structure used in many functions.
00146  * Allocated only once to reduce allocation overhead,
00147  * may be used only in one function at a time.
00148  */
00149 std::vector<Partition::Cell> _neighbour_cells;
00150
00151 void make_initial_equitable_partition();
00152
00153 void initialize_certificate();
00154
00155 void sort_edges();
00156
00157 bool nucr_find_first_component(const unsigned int level);
00158 bool nucr_find_first_component(const unsigned int level,
00159                               std::vector<unsigned int>& component,
00160                               unsigned int& component_elements,
00161                               Partition::Cell*& sh_return);
00162
00163 public:
00164 Digraph(const unsigned int N = 0);
00165
00172 ~Digraph();
00173
00187 static Digraph* read_dimacs(FILE* const fp, FILE* const errstr = stderr);
00188
00192 void write_dimacs(FILE* const fp);
00193
00194
00198 void write_dot(FILE * const fp);
00199
00203 void write_dot(const char * const file_name);
00204
00205
00206
00210 virtual unsigned int get_hash();
00211
00215 unsigned int get_nof_vertices() const {return vertices.size(); }
00216
00220 unsigned int add_vertex(const unsigned int color = 0);
00221
00228 void add_edge(const unsigned int source, const unsigned int target);
00229
00233 unsigned int get_color(const unsigned int vertex) const;
00234
00238 void change_color(const unsigned int vertex, const unsigned int color);
00239
00243 Digraph* copy() const;
00244
00251 int cmp(Digraph& other);
00252
00262 void set_splitting_heuristic(SplittingHeuristic shs) {sh = shs; }
00263
00267 Digraph* permute(const unsigned int* const perm) const;
00268
00272 Digraph* permute(const std::vector<unsigned int>& perm) const;

```

```

00273
00277     bool is_automorphism(unsigned int* const perm) const;
00278
00282     bool is_automorphism(const std::vector<unsigned int>& perm) const;
00283 };
00284
00285 } // namespace bliss

```

9.8 graph.hh

```

00001 #pragma once
00002
00003 /*
00004     Copyright (c) 2003–2021 Tommi Junttila
00005     Released under the GNU Lesser General Public License version 3.
00006
00007     This file is part of bliss.
00008
00009     bliss is free software: you can redistribute it and/or modify
00010     it under the terms of the GNU Lesser General Public License as published by
00011     the Free Software Foundation, version 3 of the License.
00012
00013     bliss is distributed in the hope that it will be useful,
00014     but WITHOUT ANY WARRANTY; without even the implied warranty of
00015     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016     GNU Lesser General Public License for more details.
00017
00018     You should have received a copy of the GNU Lesser General Public License
00019     along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #include <bliss/abstractgraph.hh>
00023
00024 namespace bliss {
00025
00031 class Graph : public AbstractGraph
00032 {
00033 public:
00042     typedef enum {
00046         shs_f = 0,
00049         shs_fs,
00052         shs_fl,
00056         shs_fm,
00060         shs_fsm,
00064         shs_flm
00065     } SplittingHeuristic;
00066
00067 protected:
00068     class Vertex {
00069     public:
00070         Vertex();
00071         ~Vertex();
00072         void add_edge(const unsigned int other_vertex);
00073         void remove_duplicate_edges(std::vector<bool>& tmp);
00074         void sort_edges();
00075
00076         unsigned int color;
00077         std::vector<unsigned int> edges;
00078         unsigned int nof_edges() const {return edges.size(); }
00079     };
00080     std::vector<Vertex> vertices;
00081     void sort_edges();
00082     void remove_duplicate_edges();
00083
00089     static unsigned int vertex_color_invariant(const Graph* const g,
00090                                                 const unsigned int v);
00091
00098     static unsigned int degree_invariant(const Graph* const g,
00099                                           const unsigned int v);
00100
00106     static unsigned int selfloop_invariant(const Graph* const g,
00107                                             const unsigned int v);
00108
00109
00110     bool refine_according_to_invariant(unsigned int (*inv)(const Graph* const g,
00111                                                            const unsigned int v));
00112
00113     /*
00114     * Routines needed when refining the partition p into equitable
00115     */
00116     bool split_neighbourhood_of_unit_cell(Partition::Cell * const);
00117     bool split_neighbourhood_of_cell(Partition::Cell * const);
00118

```

```

00122 bool is_equitable() const;
00123
00124 /* Splitting heuristics, documented in more detail in graph.cc */
00125 SplittingHeuristic sh;
00126 Partition::Cell* find_next_cell_to_be_split(Partition::Cell *cell);
00127 Partition::Cell* sh_first();
00128 Partition::Cell* sh_first_smallest();
00129 Partition::Cell* sh_first_largest();
00130 Partition::Cell* sh_first_max_neighbours();
00131 Partition::Cell* sh_first_smallest_max_neighbours();
00132 Partition::Cell* sh_first_largest_max_neighbours();
00133
00134
00135 /* A data structure used in many functions.
00136  * Allocated only once to reduce allocation overhead,
00137  * may be used only in one function at a time.
00138  */
00139 std::vector<Partition::Cell*> _neighbour_cells;
00140
00141 void make_initial_equitable_partition();
00142
00143 void initialize_certificate();
00144
00145
00146
00147 bool nucr_find_first_component(const unsigned int level);
00148 bool nucr_find_first_component(const unsigned int level,
00149                               std::vector<unsigned int>& component,
00150                               unsigned int& component_elements,
00151                               Partition::Cell*& sh_return);
00152
00153
00154
00155
00156 public:
00160 Graph(const unsigned int N = 0);
00161
00162 ~Graph();
00163
00164
00165 static Graph* read_dimacs(FILE* const fp, FILE* const errstr = stderr);
00166
00167 void write_dimacs(FILE* const fp);
00168
00169 void write_dot(FILE* const fp);
00170
00171 void write_dot(const char* const file_name);
00172
00173
00174
00175 virtual unsigned int get_hash();
00176
00177 unsigned int get_nof_vertices() const {return vertices.size(); }
00178
00179 Graph* permute(const unsigned int* const perm) const;
00180 Graph* permute(const std::vector<unsigned int>& perm) const;
00181
00182 bool is_automorphism(unsigned int* const perm) const;
00183
00184 bool is_automorphism(const std::vector<unsigned int>& perm) const;
00185
00186 unsigned int add_vertex(const unsigned int color = 0);
00187
00188 void add_edge(const unsigned int v1, const unsigned int v2);
00189
00190 unsigned int get_color(const unsigned int vertex) const;
00191
00192 void change_color(const unsigned int vertex, const unsigned int color);
00193
00194 Graph* copy() const;
00195
00196 int cmp(Graph& other);
00197
00198 void set_splitting_heuristic(const SplittingHeuristic shs) {sh = shs; }
00199
00200 };
00201
00202 } // namespace bliss

```

9.9 heap.hh

```

00001 #pragma once
00002

```

```

00003  /*
00004   Copyright (c) 2003-2021 Tommi Junttila
00005   Released under the GNU Lesser General Public License version 3.
00006
00007   This file is part of bliss.
00008
00009   bliss is free software: you can redistribute it and/or modify
00010   it under the terms of the GNU Lesser General Public License as published by
00011   the Free Software Foundation, version 3 of the License.
00012
00013   bliss is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016   GNU Lesser General Public License for more details.
00017
00018   You should have received a copy of the GNU Lesser General Public License
00019   along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020  */
00021
00022  #include <vector>
00023  #include <algorithm>
00024
00025  namespace bliss {
00026
00031  class Heap
00032  {
00033      std::vector<unsigned int> contents;
00037      struct {
00039          bool operator()(const unsigned int e1, const unsigned int e2) {return e1 > e2; }
00040      } gt;
00041  public:
00042
00047      bool is_empty() const {return contents.empty(); }
00048
00053      void clear() {contents.clear(); }
00054
00060      void insert(const unsigned int e) {
00061          contents.push_back(e);
00062          std::push_heap(contents.begin(), contents.end(), gt);
00063      }
00064
00069      unsigned int smallest() const {return contents.front(); }
00070
00076      unsigned int remove() {
00077          const unsigned int result = smallest();
00078          std::pop_heap(contents.begin(), contents.end(), gt);
00079          contents.pop_back();
00080          return result;
00081      }
00082
00086      size_t size() const {return contents.size(); }
00087  };
00088
00089  } // namespace bliss

```

9.10 kqueue.hh

```

00001  #pragma once
00002
00003  /*
00004   Copyright (c) 2003-2021 Tommi Junttila
00005   Released under the GNU Lesser General Public License version 3.
00006
00007   This file is part of bliss.
00008
00009   bliss is free software: you can redistribute it and/or modify
00010   it under the terms of the GNU Lesser General Public License as published by
00011   the Free Software Foundation, version 3 of the License.
00012
00013   bliss is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016   GNU Lesser General Public License for more details.
00017
00018   You should have received a copy of the GNU Lesser General Public License
00019   along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020  */
00021
00022  #include <new>
00023  #include <cassert>
00024
00025  namespace bliss {
00026

```

```

00030 template <class Type>
00031 class KQueue
00032 {
00033 public:
00034     KQueue();
00035     ~KQueue();
00036     void init(const unsigned int N);
00037     bool is_empty() const;
00038     unsigned int size() const;
00039     void clear();
00040     Type front() const;
00041     Type pop_front();
00042     void push_front(Type e);
00043     Type pop_back();
00044     void push_back(Type e);
00045 private:
00046     Type *entries, *end;
00047     Type *head, *tail;
00048 };
00049
00050 template <class Type>
00051 KQueue<Type>::KQueue()
00052 {
00053     entries = nullptr;
00054     end = nullptr;
00055     head = nullptr;
00056     tail = nullptr;
00057 }
00058
00059 template <class Type>
00060 KQueue<Type>::~KQueue()
00061 {
00062     delete[] entries;
00063     entries = nullptr;
00064     end = nullptr;
00065     head = nullptr;
00066     tail = nullptr;
00067 }
00068
00069 template <class Type>
00070 void KQueue<Type>::init(const unsigned int k)
00071 {
00072     assert(k > 0);
00073     delete[] entries;
00074     entries = new Type[k+1];
00075     end = entries + k + 1;
00076     head = entries;
00077     tail = head;
00078 }
00079
00080 template <class Type>
00081 void KQueue<Type>::clear()
00082 {
00083     head = entries;
00084     tail = head;
00085 }
00086
00087 template <class Type>
00088 bool KQueue<Type>::is_empty() const
00089 {
00090     return head == tail;
00091 }
00092
00093 template <class Type>
00094 unsigned int KQueue<Type>::size() const
00095 {
00096     if(tail >= head)
00097         return(tail - head);
00098     return (end - head) + (tail - entries);
00099 }
00100
00101 template <class Type>
00102 Type KQueue<Type>::front() const
00103 {
00104     assert(head != tail);
00105     return *head;
00106 }

```

```

00132
00133 template <class Type>
00134 Type KQueue<Type>::pop_front()
00135 {
00136     assert(head != tail);
00137     Type *old_head = head;
00138     head++;
00139     if(head == end)
00140         head = entries;
00141     return *old_head;
00142 }
00143
00144 template <class Type>
00145 void KQueue<Type>::push_front(Type e)
00146 {
00147     if(head == entries)
00148         head = end - 1;
00149     else
00150         head--;
00151     assert(head != tail);
00152     *head = e;
00153 }
00154
00155 template <class Type>
00156 void KQueue<Type>::push_back(Type e)
00157 {
00158     *tail = e;
00159     tail++;
00160     if(tail == end)
00161         tail = entries;
00162     assert(head != tail);
00163 }
00164
00165 } // namespace bliss

```

9.11 orbit.hh

```

00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 namespace bliss {
00023
00024 class Orbit
00025 {
00026     class OrbitEntry
00027     {
00028     public:
00029         unsigned int element;
00030         OrbitEntry *next;
00031         unsigned int size;
00032     };
00033
00034     OrbitEntry *orbits;
00035     OrbitEntry **in_orbit;
00036     unsigned int nof_elements;
00037     unsigned int _nof_orbits;
00038     void merge_orbits(OrbitEntry *o1, OrbitEntry *o2);
00039
00040 public:
00041     Orbit();
00042     ~Orbit();
00043
00044     void init(const unsigned int N);
00045     void reset();

```

```

00076
00082 void merge_orbits(unsigned int e1, unsigned int e2);
00083
00088 bool is_minimal_representative(unsigned int e) const;
00089
00094 unsigned int get_minimal_representative(unsigned int e) const;
00095
00100 unsigned int orbit_size(unsigned int e) const;
00101
00106 unsigned int nof_orbits() const {return _nof_orbits; }
00107 };
00108
00109 } // namespace bliss

```

9.12 partition.hh

```

00001 #pragma once
00002
00003 /*
00004 Copyright (c) 2003-2021 Tommi Junttila
00005 Released under the GNU Lesser General Public License version 3.
00006
00007 This file is part of bliss.
00008
00009 bliss is free software: you can redistribute it and/or modify
00010 it under the terms of the GNU Lesser General Public License as published by
00011 the Free Software Foundation, version 3 of the License.
00012
00013 bliss is distributed in the hope that it will be useful,
00014 but WITHOUT ANY WARRANTY; without even the implied warranty of
00015 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016 GNU Lesser General Public License for more details.
00017
00018 You should have received a copy of the GNU Lesser General Public License
00019 along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 namespace bliss {
00023     class Partition;
00024 }
00025
00026 #include <vector>
00027 #include <climits>
00028 #include <bliss/kqueue.hh>
00029 #include <bliss/abstractgraph.hh>
00030
00031
00032 namespace bliss {
00033
00044 class Partition
00045 {
00046 public:
00050     class Cell
00051     {
00052     friend class Partition;
00053     public:
00054         unsigned int length;
00055         /* Index of the first element of the cell in
00056            the Partition::elements array */
00057         unsigned int first;
00058         unsigned int max_ival;
00059         unsigned int max_ival_count;
00060     private:
00061         bool in_splitting_queue;
00062     public:
00063         bool in_neighbour_heap;
00064         /* Pointer to the next cell, null if this is the last one. */
00065         Cell* next;
00066         Cell* prev;
00067         Cell* next_nonsingleton;
00068         Cell* prev_nonsingleton;
00069         unsigned int split_level;
00070         bool is_unit() const {return length == 1; }
00071         bool is_in_splitting_queue() const {return in_splitting_queue; }
00072     };
00073
00074 };
00075
00076
00077 private:
00078
00083     class RefInfo {
00084     public:
00085         unsigned int split_cell_first;
00086         int prev_nonsingleton_first;

```

```

00087     int next_nonsingleton_first;
00088 };
00092     std::vector<RefInfo> refinement_stack;
00093
00094     class BacktrackInfo {
00095     public:
00096         unsigned int refinement_stack_size;
00097         unsigned int cr_backtrack_point;
00098     };
00099
00103     std::vector<BacktrackInfo> bt_stack;
00104
00105 public:
00106     AbstractGraph* graph;
00107
00108     /* Used during equitable partition refinement */
00109     KQueue<Cell*> splitting_queue;
00110     void splitting_queue_add(Cell* const cell);
00111     Cell* splitting_queue_pop();
00112     bool splitting_queue_is_empty() const;
00113     void splitting_queue_clear();
00114
00115     typedef unsigned int BacktrackPoint;
00117     BacktrackPoint set_backtrack_point();
00123
00127     void goto_backtrack_point(BacktrackPoint p);
00128
00137     Cell* individualize(Cell* const cell,
00138         const unsigned int element);
00139
00140     Cell* aux_split_in_two(Cell* const cell,
00141         const unsigned int first_half_size);
00142
00143 private:
00145     unsigned int N;
00146     Cell* cells;
00147     Cell* free_cells;
00148     unsigned int discrete_cell_count;
00149 public:
00150     Cell* first_cell;
00151     Cell* first_nonsingleton_cell;
00152     unsigned int *elements;
00153     /* invariant_values[e] gives the invariant value of the element e */
00154     unsigned int *invariant_values;
00155     /* element_to_cell_map[e] gives the cell of the element e */
00156     Cell **element_to_cell_map;
00158     Cell* get_cell(const unsigned int e) const {
00159         assert(e < N);
00160         return element_to_cell_map[e];
00161     }
00162     /* in_pos[e] points to the elements array s.t. *in_pos[e] = e */
00163     unsigned int **in_pos;
00164
00165     Partition();
00166     ~Partition();
00167
00172     void init(const unsigned int N);
00173
00178     bool is_discrete() const {return(free_cells == 0); }
00179
00180     unsigned int nof_discrete_cells() const {return(discrete_cell_count); }
00181
00185     size_t print(FILE* const fp, const bool add_newline = true) const;
00186
00190     size_t print_signature(FILE* const fp, const bool add_newline = true) const;
00191
00192     /*
00193     * Splits the Cell \a cell into [cell_1,...,cell_n]
00194     * according to the invariant_values of the elements in \a cell.
00195     * After splitting, cell_1 == \a cell.
00196     * Returns the pointer to the Cell cell_n;
00197     * cell_n != cell iff the Cell \a cell was actually splitted.
00198     * The flag \a max_ival_info_ok indicates whether the max_ival and
00199     * max_ival_count fields of the Cell \a cell have consistent values
00200     * when the method is called.
00201     * Clears the invariant values of the elements in the Cell \a cell as well as
00202     * the max_ival and max_ival_count fields of the Cell \a cell.
00203     */
00204     Cell *zsplit_cell(Cell * const cell, const bool max_ival_info_ok);
00205
00206     /*
00207     * Routines for component recursion
00208     */
00209     void cr_init();

```

```

00210 void cr_free();
00211 unsigned int cr_get_level(const unsigned int cell_index) const;
00212 unsigned int cr_split_level(const unsigned int level,
00213                             const std::vector<unsigned int>& cells);
00214
00216 void clear_ivs(Cell* const cell);
00217
00218 private:
00219 /*
00220  * Component recursion data structures
00221  */
00222
00223 /* Is component recursion support in use? */
00224 bool cr_enabled;
00225
00226 class CRCell {
00227 public:
00228     unsigned int level;
00229     CRCell* next;
00230     CRCell** prev_next_ptr;
00231     void detach() {
00232         if(next)
00233             next->prev_next_ptr = prev_next_ptr;
00234         *(prev_next_ptr) = next;
00235         level = UINT_MAX;
00236         next = nullptr;
00237         prev_next_ptr = nullptr;
00238     }
00239 };
00240 CRCell* cr_cells;
00241 CRCell** cr_levels;
00242 class CR_BTInfo {
00243 public:
00244     unsigned int created_trail_index;
00245     unsigned int splitted_level_trail_index;
00246 };
00247 std::vector<unsigned int> cr_created_trail;
00248 std::vector<unsigned int> cr_splitted_level_trail;
00249 std::vector<CR_BTInfo> cr_bt_info;
00250 unsigned int cr_max_level;
00251 void cr_create_at_level(const unsigned int cell_index, unsigned int level);
00252 void cr_create_at_level_trailed(const unsigned int cell_index, unsigned int level);
00253 unsigned int cr_get_backtrack_point();
00254 void cr_goto_backtrack_point(const unsigned int btpoint);
00255
00256
00257 /*
00258  *
00259  * Auxiliary routines for sorting and splitting cells
00260  *
00261  */
00262 Cell* sort_and_split_cell1(Cell* cell);
00263 Cell* sort_and_split_cell255(Cell* const cell, const unsigned int max_ival);
00264 bool shellsort_cell(Cell* cell);
00265 Cell* split_cell(Cell* const cell);
00266
00267 /*
00268  * Some auxiliary stuff needed for distribution count sorting.
00269  * To make the code thread-safe (modulo the requirement that each graph is
00270  * only accessed in one thread at a time), the arrays are owned by
00271  * the partition instance, not statically defined.
00272  */
00273 unsigned int dcs_count[256];
00274 unsigned int dcs_start[256];
00275 void dcs_cumulate_count(const unsigned int max);
00276 };
00277
00278
00279 inline Partition::Cell*
00280 Partition::splitting_queue_pop()
00281 {
00282     assert(!splitting_queue.is_empty());
00283     Cell* const cell = splitting_queue.pop_front();
00284     assert(cell->in_splitting_queue);
00285     cell->in_splitting_queue = false;
00286     return cell;
00287 }
00288
00289 inline bool
00290 Partition::splitting_queue_is_empty() const
00291 {
00292     return splitting_queue.is_empty();
00293 }
00294
00295
00296 inline unsigned int
00297 Partition::cr_get_level(const unsigned int cell_index) const

```

```

00298 {
00299     assert(cr_enabled);
00300     assert(cell_index < N);
00301     assert(cr_cells[cell_index].level != UINT_MAX);
00302     return(cr_cells[cell_index].level);
00303 }
00304
00305 } // namespace bliss

```

9.13 stats.hh

```

00001 #pragma once
00002
00003 /*
00004     Copyright (c) 2003-2021 Tommi Junttila
00005     Released under the GNU Lesser General Public License version 3.
00006
00007     This file is part of bliss.
00008
00009     bliss is free software: you can redistribute it and/or modify
00010     it under the terms of the GNU Lesser General Public License as published by
00011     the Free Software Foundation, version 3 of the License.
00012
00013     bliss is distributed in the hope that it will be useful,
00014     but WITHOUT ANY WARRANTY; without even the implied warranty of
00015     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016     GNU Lesser General Public License for more details.
00017
00018     You should have received a copy of the GNU Lesser General Public License
00019     along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #include <cstdio>
00023 #include <bliss/abstractgraph.hh>
00024 #include <bliss/bignum.hh>
00025
00026 namespace bliss {
00027
00031 class Stats
00032 {
00033     friend class AbstractGraph;
00034     Bignum group_size;
00035     long double group_size_approx;
00036     long unsigned int nof_nodes;
00037     long unsigned int nof_leaf_nodes;
00038     long unsigned int nof_bad_nodes;
00039     long unsigned int nof_canupdates;
00040     long unsigned int nof_generators;
00041     unsigned long int max_level;
00042     void reset()
00043     {
00044         group_size.assign(1);
00045         group_size_approx = 1.0;
00046         nof_nodes = 0;
00047         nof_leaf_nodes = 0;
00048         nof_bad_nodes = 0;
00049         nof_canupdates = 0;
00050         nof_generators = 0;
00051         max_level = 0;
00052     }
00053 public:
00054     Stats() { reset(); }
00055     size_t print(FILE* const fp) const
00056     {
00057         size_t r = 0;
00058         r += fprintf(fp, "Nodes: %lu\n", nof_nodes);
00059         r += fprintf(fp, "Leaf nodes: %lu\n", nof_leaf_nodes);
00060         r += fprintf(fp, "Bad nodes: %lu\n", nof_bad_nodes);
00061         r += fprintf(fp, "Canrep updates: %lu\n", nof_canupdates);
00062         r += fprintf(fp, "Generators: %lu\n", nof_generators);
00063         r += fprintf(fp, "Max level: %lu\n", max_level);
00064         r += fprintf(fp, "|Aut|: ") + group_size.print(fp) + fprintf(fp, "\n");
00065         fflush(fp);
00066         return r;
00067     }
00068     const Bignum& get_group_size() const {return group_size;}
00069     long double get_group_size_approx() const {return group_size_approx;}
00070     long unsigned int get_nof_nodes() const {return nof_nodes;}
00071     long unsigned int get_nof_leaf_nodes() const {return nof_leaf_nodes;}
00072     long unsigned int get_nof_bad_nodes() const {return nof_bad_nodes;}
00073     long unsigned int get_nof_canupdates() const {return nof_canupdates;}
00074     long unsigned int get_nof_generators() const {return nof_generators;}
00075     unsigned long int get_max_level() const {return max_level;}

```

```

00096 };
00097
00098 } // namespace bliss

```

9.14 timer.hh

```

00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #include <chrono>
00023
00024 namespace bliss {
00025
00026 class Timer
00027 {
00028     std::chrono::high_resolution_clock::time_point start_time_;
00029 public:
00030     Timer() {
00031         reset();
00032     }
00033
00034     void reset() {
00035         start_time_ = std::chrono::high_resolution_clock::now();
00036     }
00037
00038     double get_duration() const {
00039         return std::chrono::duration_cast<std::chrono::duration<double>
00040 >(std::chrono::high_resolution_clock::now() - start_time_).count();
00041     }
00042 };
00043
00044 } // namespace bliss

```

9.15 uintseqhash.hh

```

00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 namespace bliss {
00023
00024 class UIntSeqHash
00025 {

```

```

00029 protected:
00030     unsigned int h;
00031 public:
00032     UIntSeqHash() {h = 0; }
00033     UIntSeqHash(const UIntSeqHash &other) {h = other.h; }
00034     UIntSeqHash& operator=(const UIntSeqHash &other) {h = other.h; return *this; }
00035
00037     void reset() {h = 0; }
00038
00040     void update(unsigned int n);
00041
00043     unsigned int get_value() const {return h; }
00044
00048     int cmp(const UIntSeqHash &other) const {
00049         return (h < other.h)?-1:((h == other.h)?0:1);
00050     }
00052     bool is_lt(const UIntSeqHash &other) const {return cmp(other) < 0; }
00054     bool is_le(const UIntSeqHash &other) const {return cmp(other) <= 0; }
00056     bool is_equal(const UIntSeqHash &other) const {return cmp(other) == 0; }
00057 };
00058
00059
00060 } // namespace bliss

```

9.16 src/utls.hh File Reference

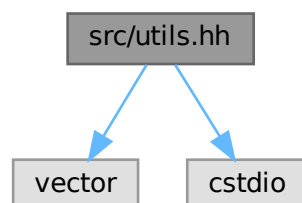
Some small utilities.

```

#include <vector>
#include <cstdio>

```

Include dependency graph for utls.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.](#)

```
00001 #pragma once
00002
00003 /*
00004  Copyright (c) 2003-2021 Tommi Junttila
00005  Released under the GNU Lesser General Public License version 3.
00006
00007  This file is part of bliss.
00008
00009  bliss is free software: you can redistribute it and/or modify
00010  it under the terms of the GNU Lesser General Public License as published by
00011  the Free Software Foundation, version 3 of the License.
00012
00013  bliss is distributed in the hope that it will be useful,
00014  but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016  GNU Lesser General Public License for more details.
00017
00018  You should have received a copy of the GNU Lesser General Public License
00019  along with bliss. If not, see <http://www.gnu.org/licenses/>.
00020 */
00021
00022 #include <vector>
00023 #include <cstdio>
00024
00025 namespace bliss {
00026
00027 size_t print_permutation(FILE* fp,
00028                          const unsigned int N,
00029                          const unsigned int* perm,
00030                          const unsigned int offset = 0);
00031
00032 size_t print_permutation(FILE* fp,
00033                          const std::vector<unsigned int>& perm,
00034                          const unsigned int offset = 0);
00035
00036 bool is_permutation(const unsigned int N, const unsigned int* perm);
00037
00038 bool is_permutation(const std::vector<unsigned int>& perm);
00039
00040 } // namespace bliss
```

Index

- ~BigNum
 - bliss::BigNum, [24](#)
- ~Digraph
 - bliss::Digraph, [29](#)
- ~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, [37](#)
- assign
 - bliss::BigNum, [24](#)
- BacktrackPoint
 - bliss::Partition, [50](#)
- 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, [19](#)
 - find_automorphisms, [19](#)
 - get_color, [19](#)
 - get_hash, [20](#)
 - get_nof_vertices, [20](#)
 - is_automorphism, [20](#)
 - permute, [20](#), [21](#)
 - set_component_recursion, [21](#)
 - set_failure_recording, [21](#)
 - set_long_prune_activity, [21](#)
 - set_verbose_file, [22](#)
 - set_verbose_level, [22](#)
 - write_dimacs, [22](#)
 - write_dot, [23](#)
- bliss::BigNum, [23](#)
 - ~BigNum, [24](#)
 - assign, [24](#)
 - BigNum, [24](#)
 - get_factors, [24](#)
 - multiply, [24](#)
 - print, [24](#)
 - to_string, [25](#)
- bliss::Digraph, [27](#)
 - ~Digraph, [29](#)
 - 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, [31](#), [32](#)
 - read_dimacs, [32](#)
 - set_component_recursion, [32](#)
 - set_failure_recording, [33](#)
 - set_long_prune_activity, [33](#)
 - set_splitting_heuristic, [33](#)
 - set_verbose_file, [33](#)
 - 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, [34](#)
- bliss::Graph, [35](#)
 - ~Graph, [37](#)
 - add_edge, [37](#)
 - add_vertex, [37](#)
 - canonical_form, [37](#)
 - change_color, [38](#)
 - cmp, [38](#)
 - copy, [38](#)
 - find_automorphisms, [38](#)
 - get_color, [39](#)
 - get_hash, [39](#)
 - get_nof_vertices, [39](#)
 - Graph, [37](#)
 - is_automorphism, [39](#), [40](#)
 - permute, [40](#)
 - read_dimacs, [40](#)
 - set_component_recursion, [41](#)

- set_failure_recording, 41
- set_long_prune_activity, 41
- set_splitting_heuristic, 42
- set_verbose_file, 42
- set_verbose_level, 42
- shs_f, 37
- shs_fl, 37
- shs_flm, 37
- shs_fm, 37
- shs_fs, 37
- shs_fsm, 37
- SplittingHeuristic, 36
- write_dimacs, 42
- write_dot, 42, 43
- bliss::Heap, 43
 - clear, 44
 - insert, 44
 - is_empty, 44
 - remove, 44
 - size, 44
 - smallest, 44
- bliss::KQueue< Type >, 45
 - clear, 45
 - front, 45
 - init, 46
 - is_empty, 46
 - KQueue, 45
 - pop_back, 46
 - pop_front, 46
 - push_back, 46
 - push_front, 46
 - size, 47
- bliss::Orbit, 47
 - get_minimal_representative, 48
 - init, 48
 - is_minimal_representative, 48
 - merge_orbits, 48
 - nof_orbits, 48
 - Orbit, 48
 - orbit_size, 49
 - reset, 49
- bliss::Partition, 49
 - BacktrackPoint, 50
 - clear_ivs, 50
 - get_cell, 50
 - goto_backtrack_point, 50
 - individualize, 50
 - init, 51
 - is_discrete, 51
 - print, 51
 - print_signature, 51
 - set_backtrack_point, 51
- bliss::Partition::Cell, 26
 - is_in_splitting_queue, 26
 - is_unit, 26
- bliss::Stats, 52
 - get_group_size, 52
 - get_group_size_approx, 52
 - get_max_level, 52
 - get_nof_bad_nodes, 52
 - get_nof_canupdates, 52
 - get_nof_generators, 53
 - get_nof_leaf_nodes, 53
 - get_nof_nodes, 53
 - print, 53
- bliss::Timer, 53
- bliss::UIntSeqHash, 54
 - cmp, 54
 - get_value, 54
 - is_equal, 54
 - is_le, 55
 - is_lt, 55
 - reset, 55
 - update, 55
- bliss_add_edge
 - bliss_C.h, 64
- bliss_add_vertex
 - bliss_C.h, 64
- bliss_C.h
 - bliss_add_edge, 64
 - bliss_add_vertex, 64
 - bliss_cmp, 64
 - bliss_find_automorphisms, 64
 - bliss_find_canonical_labeling, 64
 - bliss_get_nof_vertices, 65
 - bliss_hash, 65
 - bliss_new, 65
 - bliss_permute, 65
 - bliss_read_dimacs, 65
 - bliss_release, 66
 - bliss_write_dimacs, 66
 - bliss_write_dot, 66
- bliss_cmp
 - bliss_C.h, 64
- bliss_find_automorphisms
 - bliss_C.h, 64
- bliss_find_canonical_labeling
 - bliss_C.h, 64
- bliss_get_nof_vertices
 - bliss_C.h, 65
- bliss_graph_struct, 25
- bliss_hash
 - bliss_C.h, 65
- bliss_new
 - bliss_C.h, 65
- bliss_permute
 - bliss_C.h, 65
- bliss_read_dimacs
 - bliss_C.h, 65
- bliss_release
 - bliss_C.h, 66
- bliss_stats_struct, 25
- bliss_write_dimacs
 - bliss_C.h, 66
- bliss_write_dot
 - bliss_C.h, 66

- canonical_form
 - bliss::AbstractGraph, 18
 - bliss::Digraph, 29
 - bliss::Graph, 37
- change_color
 - bliss::AbstractGraph, 19
 - bliss::Digraph, 29
 - bliss::Graph, 38
- clear
 - bliss::Heap, 44
 - bliss::KQueue< Type >, 45
- clear_ivs
 - bliss::Partition, 50
- cmp
 - bliss::Digraph, 30
 - bliss::Graph, 38
 - bliss::UIntSeqHash, 54
- copy
 - bliss::Digraph, 30
 - bliss::Graph, 38
- Digraph
 - bliss::Digraph, 28
- find_automorphisms
 - bliss::AbstractGraph, 19
 - bliss::Digraph, 30
 - bliss::Graph, 38
- front
 - bliss::KQueue< Type >, 45
- get_cell
 - bliss::Partition, 50
- get_color
 - bliss::AbstractGraph, 19
 - bliss::Digraph, 30
 - bliss::Graph, 39
- get_factors
 - bliss::BigNum, 24
- get_group_size
 - bliss::Stats, 52
- get_group_size_approx
 - bliss::Stats, 52
- get_hash
 - bliss::AbstractGraph, 20
 - bliss::Digraph, 31
 - bliss::Graph, 39
- get_max_level
 - bliss::Stats, 52
- get_minimal_representative
 - bliss::Orbit, 48
- get_nof_bad_nodes
 - bliss::Stats, 52
- get_nof_canupdates
 - bliss::Stats, 52
- get_nof_generators
 - bliss::Stats, 53
- get_nof_leaf_nodes
 - bliss::Stats, 53
- get_nof_nodes
 - bliss::Stats, 53
- get_nof_vertices
 - bliss::AbstractGraph, 20
 - bliss::Digraph, 31
 - bliss::Graph, 39
- get_value
 - bliss::UIntSeqHash, 54
- goto_backtrack_point
 - bliss::Partition, 50
- Graph
 - bliss::Graph, 37
- individualize
 - bliss::Partition, 50
- init
 - bliss::KQueue< Type >, 46
 - bliss::Orbit, 48
 - bliss::Partition, 51
- insert
 - bliss::Heap, 44
- is_automorphism
 - bliss::AbstractGraph, 20
 - bliss::Digraph, 31
 - bliss::Graph, 39, 40
- is_discrete
 - bliss::Partition, 51
- is_empty
 - bliss::Heap, 44
 - bliss::KQueue< Type >, 46
- is_equal
 - bliss::UIntSeqHash, 54
- is_in_splitting_queue
 - bliss::Partition::Cell, 26
- is_le
 - bliss::UIntSeqHash, 55
- is_lt
 - bliss::UIntSeqHash, 55
- is_minimal_representative
 - bliss::Orbit, 48
- is_permutation
 - bliss, 16
- is_unit
 - bliss::Partition::Cell, 26
- KQueue
 - bliss::KQueue< Type >, 45
- merge_orbits
 - bliss::Orbit, 48
- multiply
 - bliss::BigNum, 24
- nof_orbits
 - bliss::Orbit, 48
- Orbit
 - bliss::Orbit, 48
- orbit_size

- bliss::Orbit, 49
- Outline, 1
- permute
 - bliss::AbstractGraph, 20, 21
 - bliss::Digraph, 31, 32
 - bliss::Graph, 40
- pop_back
 - bliss::KQueue< Type >, 46
- pop_front
 - bliss::KQueue< Type >, 46
- print
 - bliss::BigNum, 24
 - bliss::Partition, 51
 - bliss::Stats, 53
- print_permutation
 - bliss, 16
- print_signature
 - bliss::Partition, 51
- push_back
 - bliss::KQueue< Type >, 46
- push_front
 - bliss::KQueue< Type >, 46
- read_dimacs
 - bliss::Digraph, 32
 - bliss::Graph, 40
- remove
 - bliss::Heap, 44
- reset
 - bliss::Orbit, 49
 - bliss::UIntSeqHash, 55
- set_backtrack_point
 - bliss::Partition, 51
- set_component_recursion
 - bliss::AbstractGraph, 21
 - bliss::Digraph, 32
 - bliss::Graph, 41
- set_failure_recording
 - bliss::AbstractGraph, 21
 - bliss::Digraph, 33
 - bliss::Graph, 41
- set_long_prune_activity
 - bliss::AbstractGraph, 21
 - bliss::Digraph, 33
 - bliss::Graph, 41
- set_splitting_heuristic
 - bliss::Digraph, 33
 - bliss::Graph, 42
- set_verbose_file
 - bliss::AbstractGraph, 22
 - bliss::Digraph, 33
 - bliss::Graph, 42
- set_verbose_level
 - bliss::AbstractGraph, 22
 - bliss::Digraph, 34
 - bliss::Graph, 42
- 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, 44
 - bliss::KQueue< Type >, 47
- smallest
 - bliss::Heap, 44
- SplittingHeuristic
 - bliss::Digraph, 28
 - bliss::Graph, 36
- src/abstractgraph.hh, 57
- src/bignum.hh, 60
- src/bliss_C.h, 62, 66
- src/defs.hh, 67, 68
- src/digraph.hh, 69
- src/graph.hh, 71
- src/heap.hh, 72
- src/kqueue.hh, 73
- src/orbit.hh, 75
- src/partition.hh, 76
- src/stats.hh, 79
- src/timer.hh, 80
- src/uintseqhash.hh, 80
- src/utils.hh, 81, 82
- The bliss executable, 3
- to_string
 - bliss::BigNum, 25
- update
 - bliss::UIntSeqHash, 55
- write_dimacs
 - bliss::AbstractGraph, 22
 - bliss::Digraph, 34
 - bliss::Graph, 42
- write_dot
 - bliss::AbstractGraph, 23
 - bliss::Digraph, 34
 - bliss::Graph, 42, 43