Contents

1 Introduction

Cliquer is a set of C routines for finding cliques in an arbitrary weighted graph. It can search for maximum cliques, maximum-weight cliques, or cliques whose size or weight is within a given range, optionally limiting the search to maximal cliques. The cliques that are found can either be stored in memory, or a user-defined function can be called for each clique. Cliquer is re-entrant, that is, one may use the clique-searching routines again from the user-defined function. The package also contains a command-line utility `cl`, which can be used to find cliques from graphs in DIMACS-format files.

Cliquer uses an exact branch-and-bound algorithm developed by the second author [10, 11] for the maximum clique and maximum-weight clique problems, and suitably modified versions for other clique searches. The papers cited also contain comparisons with some other common algorithms.

Cliquer has been developed on Linux and it should compile without modification on most modern UNIX systems. Other systems may require minor changes to the source code.

Cliques and important problems related to these are defined in Section 2, where the algorithm used by Cliquer is also briefly discussed. The data structures and clique-searching functions are described in Sections 3 and 4, respectively. Section 5 contains information on how to configure and compile Cliquer. Appendices A, B, and C document functions that are useful for manipulating sets, graphs, and vertex orders (functions of the last type affect the efficiency of the search), respectively. The DIMACS graph file format is described in Appendix D, and example programs are presented in Appendix E.

Cliquer is flexible and easy to use and still competes with the fastest clique programs—it is indeed the fastest for several types of graphs [10, 11]. Bug reports and other feedback should be directed to the corresponding author.

2 Preliminaries and the Algorithm

In this section, central concepts are introduced and the algorithm used by Cliquer is shortly described. For a more extensive treatment, we refer to [10, 11]. Note, however, that the vertex numbering used by Cliquer is the reverse of that described in the references. The algorithm is described here as Cliquer uses it.

This section is not necessary for using Cliquer, but Section 2.4 may be useful for understanding the vertex ordering functions.

2.1 Cliques

We denote an undirected graph by $G = (V, E)$, where V is the set of vertices and E is the set of edges, such that each edge is a set of two vertices in V (which are said to be *adjacent*). The number of vertices, $|V|$, is called the *order* of G. We denote the weight of a vertex $v \in V$ by $w(v)$, and the sum of vertex weights in a set $W \subseteq V$ by $w(W)$.

A *clique* in G is a subset $S \subseteq V$ of vertices, all of which are adjacent to each other. A clique is said to be *maximal* if it is not the subset of any larger clique, and *maximum* if there are no larger cliques in the graph.

In the *maximum clique problem* one wants to find a maximum clique in an arbitrary undirected graph. Since this is an NP-hard problem [6], no polynomial time algorithms are known to exist. Cliquer uses a branch-and-bound algorithm developed by the second author [11], which is based on an algorithm by Carraghan and Pardalos [3]. It adds more efficient pruning methods by storing in memory the maximum clique size of subgraphs that it has discovered. In many cases, it is faster than other known algorithms [11].

The maximum clique problem is computationally equivalent to some other important problems. An *independent set* (also called a *stable set*) of a graph is a set of vertices, none of which are adjacent. By taking the complement of a graph, the *maximum independent set problem* is transformed into the maximum clique problem. For a given independent set of a graph, any edge in the graph is incident to a vertex that is not in this set. Therefore, the *minimum vertex cover problem* is another computationally equivalent problem. For weighted graphs, we get analogous problems. For an extensive survey of the maximum clique problem and related issues, see [1]. This survey also lists a wide variety of applications.

Clique algorithms can be used, for example, to find error-correcting codes of maximum size [9, 12]. Take one vertex for each word in the given space, and let vertices whose mutual distance is at least d, a prescribed parameter, be adjacent. Then maximum cliques in the constructed graph correspond to error-correcting codes of maximum size. An example of finding codes with Cliquer is given in Appendix E. Maximum-weight cliques have applications in the search for codes with prescribed automorphism groups [2, 8, 10]. In searching for combinatorial objects, one may want to use the program *nauty* [7] for isomorphism tests. Cliquer has been designed to respect *nauty*, so both routines can be used in the same program.

2.2 A Maximum Clique Algorithm

We assume some order for the vertices $V = \{v_1, v_2, \ldots, v_n\}$. Let $S_i = \{v_1, v_2, \ldots, v_i\} \subseteq V$. We define the function $c(i)$ to be the size of the maximum clique in the subgraph induced by S_i. Obviously, for every $i = 1, \ldots, n-1$ we have either $c(i+1) = c(i)$ or $c(i+1) = c(i) + 1$. Moreover, $c(i+1) = c(i)+1$ iff there exists a clique in S_{i+1} of size $c(i)+1$ that includes vertex v_{i+1}.

Cliquer calculates the values of $c(i)$ starting from $c(1) = 1$ up, and stores the values found. This enables a pruning strategy not found in old algorithms, such as [3]. Namely, when Cliquer is calculating $c(i+1)$ (that is, searching for a clique of size $c(i)+1$ within S_{i+1}), and it has formed a clique W and is considering adding vertex v_j, it can prune the search if $|W| + c(j) \leq c(i)$. As j is chosen to be the largest index in the set of vertices to be considered, it follows that a clique of size $c(i)+1$ that contains W cannot exist in S_{i+1}. Trivially, if it finds a clique of size $c(i)+1$, it can prune the whole search and start calculating $c(i+2)$.

When searching for *all* maximum cliques, Cliquer first determines the size of the maximum cliques, and then starts the search again at the suitable position.

The order of the vertices has a major impact on the speed of the algorithm. Therefore it is beneficial to use some time in devising the order of the vertices. This ordering is discussed further in Section 2.4.

2.3 A Maximum-Weight Clique Algorithm

The algorithm used with weighted cliques is described in [10], and it is very similar to the unweighted case. The same kind of pruning is used when considering additional vertices, that is, the search can be pruned if $w(W) + C(j) \leq C(i)$, where v_j is the vertex being considered and $C(i)$ is the maximum weight of a clique in S_i.

However, finding a clique of greater weight than $C(i)$ in S_{i+1} is insufficient, as it is not necessarily the maximum-weight clique of S_{i+1}. Therefore, the search is continued until all combinations have been checked, or a clique with weight $C(i) + w(v_{i+1})$ is found.

2.4 Ordering the Vertices

The order of the vertices in the search has a major impact on the speed of the search. Cliquer allows ordering the vertices using a variety of functions, or by defining one's own order.

Note that the vertex numbering used by Cliquer is the reverse of that in [11] and [10], since there S_i is defined as $\{v_i, \ldots, v_n\}$ instead of $\{v_1, \ldots, v_i\}$. It is very much an open research problem to try to find proper vertex orders. The following two heuristics have experimentally been found to be effective [10, 11].

In a vertex coloring of a graph, two adjacent vertices must be assigned different colors. In both the unweighted and the weighted case, the graph is colored one color at a time, adding vertices to a color class as long as possible before creating a new color class. In the unweighted case, the vertex chosen is always the one with the largest degree within the uncolored graph. In the weighted case, the vertex is chosen from the vertices with smallest weight, and has the largest sum of weights adjacent to the vertex in the uncolored graph. The vertices are labelled $v_1, v_2, \ldots, v_n$ in the order that they are chosen during the coloring.

The order of the vertices is defined in `clique_options`, which is passed to the clique searching functions. See Section 4.2 for details.

3 Data Structures

Cliquer defines data structures for sets and graphs, explained in the following sections. In addition, the data type `boolean` (defined as an `int`) and the expressions `TRUE` and `FALSE` are defined for use in boolean variables.

3.1 Sets

A `setelement` is an unsigned integer data type, which is either 16, 32 or 64 bits in length, as set at compile-time (see Section 5.1). `ELEMENTSIZE` is the number of bits in one `setelement`.

A set (by which we always mean a subset of $X = \{0, 1, \ldots, n-1\}$) is represented by an array of `setelements`, which contains one bit for each value the set may hold. The type for sets is `set_t` (equivalent to `setelement *`), and sets should be defined as "`set_t s`" (*not* "`set_t *s`").

Once initialized, `set_t s` contains a representative bit for each value it can contain, which is 1 if the value is in the set, 0 otherwise. Values $\{0, \ldots,$ `ELEMENTSIZE-1`$\}$ are stored in `s[0]`, the next `ELEMENTSIZE` values in `s[1]`, and so forth. Within a `setelement`, the smallest value is stored in the least-significant bit. In addition, the size n of the superset X (giving the total number of 0s and 1s) is stored in `s[-1]`. It is recommended that one uses the macros and functions listed in Appendix A for set manipulation.

For performance reasons, the set handling routines are defined in `set.h` as static functions, so that the compiler can inline them. This may cause warnings of unused functions on some compilers, which may be safely ignored. See Section 5.1 for details.

3.2 Graphs

Graphs in Cliquer are handled with the data type `graph_t *`. The vertices are numbered $\{0, 1, \ldots, n-1\}$, where n is the number of vertices. The structure `graph_t` contains an array of n sets, each of which tells what vertices are adjacent to that vertex, and an array of n `int`s containing the vertex weights. The adjacency matrix is required to be symmetric (that is, the graph must not be directed) and anti-reflexive, and all vertex weights must be positive. The structure contains the following members:

`int n` Number of vertices in the graph.

`set_t *edges` A list of `n` sets, where `edges[i]` contains the vertices that are adjacent to vertex `i`.

`int *weights` A list of `n` `int`s which contain the vertex weights.

It is recommended that a program should only use the type `graph_t *`, and use the functions described in Appendix B for graph manipulation.

3.3 Limitations

The data structures used in Cliquer are dynamically allocated, and do not impose restrictions on graph or set sizes. The only limitations are made by available memory and the integer data type size. Specifically, the total weight of a graph must be less than the maximum value that can be stored in an `int`, and the maximum size of a set must fit into one `setelement`. As most modern systems have at least 32-bit integers, this should not be a limitation.

4 Clique Searching

This section contains the core of this user's manual, the clique-searching functions and their options.

4.1 Clique-Searching Functions

Cliquer includes six functions that search for cliques, all of which begin with "`clique_`". Cliques are returned as sets (of type `set_t`) of the vertices forming the clique. The size of the superset X always equals the number of vertices in the graph. The search functions are as follow:

`int clique_max_weight(g,opts)`
: Returns the largest weight that any clique in graph `g` has. Note that using this function is no faster than using `clique_find_single(g, 0,0,FALSE,opts)` to actually find a maximum-weight clique.

`set_t clique_find_single(g,min_weight,max_weight,maximal,opts)`
: Returns a single clique in graph `g` fulfilling the weight requirements given (see below for details). If such a clique does not exist in the graph, the function returns `NULL`. Note that the clique storage methods in `opts` are *not* used.

`int clique_find_all(g,min_weight,max_weight,maximal,opts)`
: Searches for all cliques in graph `g` fulfilling the weight requirements given (see below for details). The cliques are stored as defined in `opts`. The return value is the number of cliques found.

`int clique_unweighted_max_weight(g,opts)`
`set_t clique_unweighted_find_single(g,min,max,maximal,opts)`
`int clique_unweighted_find_all(g,min,max,maximal,opts)`
: These functions are identical to the three above, except that they assume that all vertex weights are 1. This is useful if one has a weighted graph and wishes to find cliques based on size instead of weight. The first three functions automatically use these functions for unweighted graphs.

The arguments `min_weight`, `max_weight`, and `maximal` define what kind of cliques are searched for. The structure `opts` contains information about how the vertices are to be ordered during the search, how the cliques are stored, and how the progress of the routine is reported. All arguments are treated read-only, though `opts` may contain pointers to areas that are modified. The meanings of the parameters are as follows:

`graph_t *g` The graph in which cliques are searched for.

`int min_weight` Minimum weight of cliques to search for. If `min_weight`=0, then the functions search for maximum-weight cliques.

`int max_weight` Maximum weight of cliques to search for. If `max_weight`=0, then no upper limit is used. If `min_weight`=0, then also `max_weight` *must* be 0.

If `max_weight`> 0, then it is required that `min_weight` ≤ `max_weight`. See Section 5.4 for details.

`boolean maximal` If `TRUE`, requires the cliques to be maximal.

`clique_options *opts` Details how to store the cliques, how to order the vertices during the search, and how to report the progress of the routine (see Section 4.2 below for details). If `opts`=`NULL`, then the default options in `clique_default_options` are used.

4.2 Cliquer Options

The `clique_options` structure contains information that does not affect what kinds of cliques are searched for, but affects the speed of the algorithm, how results are stored, and how progress is reported. Note that when using different vertex orderings, `clique_find_single` may find different cliques fulfilling the weight requirements. The default options are defined in the global variable `clique_default_options`, which can also be modified. The structure contains the following fields:

`int *(*reorder_function)(graph_t *, boolean)`
`int *reorder_map`

These variables define the order of the vertices used in the search. This may greatly affect the speed of the search. Either one of these variables *must* be `NULL`. If both are `NULL`, no reordering will be carried out (that is, the order follows that of the created graph). See Section 2.4 for details on vertex orders.

If `reorder_function` is non-NULL, it is called with the graph as an argument to get the order of vertices to use in the search. The function definition should be `int *function(graph_t *g,boolean weighted)`, where `g` is the graph and `weighted` tells whether a weighted or unweighted search is being done. The function should return an array of `g->n` `int`s allocated with `malloc()`, which contains each of the values {0, 1, ..., `g->n-1`} exactly once. Cliquer has several ordering functions predefined, which are documented in Appendix C.

Alternatively, the vertex order can be given in `reorder_map`. In this case, the array is *not* freed.

The default is to have `reorder_function` as `reorder_by_greedy_coloring` and `reorder_map` as `NULL`.

`boolean (*time_function)( ... )`
`FILE *output`

If non-NULL, this function is called at every base-level recursion. The function definition should be

```
boolean function(int level,int i,int n,int max,
                 double cputime,double realtime,
                 clique_options *opts),
```

where `level` is the re-entrance level (increased by one every time a clique-searching function is called, and decreased when it returns; 1 for the first clique-searching call), `i` is the level of the current recursion, `n` is the total number of recursion levels (the size of the graph), `max` is the weight of the heaviest clique found so far (but see later remarks), `cputime` is the CPU time used by this program in the recursion so far, `realtime` is the total amount of time the recursion has taken so far, and `opts` is the option structure. The values of `cputime` and `realtime` should be approximately the same if there are no other time-consuming processes being run on the computer. The function should return `TRUE` to continue the search, or `FALSE` to abort.

The definition of `max` given above has the following exception. If searching for more than a single clique in a weight interval, `max` stops growing when it has reached `min_weight-1`. Also note that when searching for all maximum cliques, the search will first process the whole graph (to find the size of the maximum clique) and then continue the search for all such graphs from an earlier point; this affects the value of i accordingly.

Cliquer defines two functions that can be used as progress indicating functions. The function `clique_print_time` prints a line indicating the progress if over 0.1 seconds have elapsed from the previous time a line has been printed or if one of the other arguments has changed. It indents the line with two spaces for every re-entrance. The function `clique_print_time_always` works in the same way, except that it prints the line on every call. The time printed by these functions is the real time spent in the algorithm. They print to the file stream `output`, or `stdout` if it is `NULL`.

The default value for `time_function` is `clique_print_time` and `NULL` for `output`.

`boolean (*user_function)(set_t,graph_t *,clique_options *)`
`void *user_data`

When searching for multiple cliques, `user_function` is called for every clique found, if non-NULL. The function definition should be `boolean function(set_t s,graph_t *g,clique_options *opts)`, where `s` is the clique, `g` is the graph, and `opts` is the option structure used. The function should return `TRUE` to continue the search or `FALSE` to abort and return to the caller. Note that there is no way of telling from the return values of the clique-searching functions whether the search was completed or aborted in a user-defined function; if distinction is necessary, a user-defined global variable can be used.

Cliquer is re-entrant, so it is safe to use the clique searching functions from `user_function`. However, `clique_default_options` is the same for all instances, so one may need to define one's own options structure.

The variable `user_data` is ignored by Cliquer, and can be used to pass data to `user_function`.

`set_t *clique_list`
`int clique_list_length`

When searching for multiple cliques, the cliques found are stored in `clique_list`, if non-NULL. This should be an array of at least `clique_list_length` unallocated sets of type `set_t`. At most `clique_list_length` cliques are stored, after that the search continues, but the cliques are not stored.

Both `user_function` and `clique_list` can be defined at the same time. If neither is defined, the only result of `clique_find_all` is the number of cliques in the graph. Note that if either `user_function` or `time_function` returns `FALSE`, the search is aborted. In this case, `clique_find_all` returns the number of cliques found so far, `clique_find_single` returns `NULL`, and `clique_max_weight` returns 0. The functions `clique_print_time` and `clique_print_time_always` always return `TRUE`.

5 Compiling Cliquer

5.1 Configuration

Cliquer is configured in two files: `Makefile` and `cliquerconf.h`. The user should in all cases read `Makefile` for configuration options. The configuration options in `cliquerconf.h` have reasonable defaults, and one should be able to compile Cliquer without modifications.

5.1.1 Makefile

The makefile contains mainly compilation options. The user must define the compiler to be used by setting the `CC` variable, and the compilation flags in `CFLAGS`. One may also leave `CFLAGS` blank, but in this case no code optimization will be done.

The variable `LONGOPTS` is added to the compilation flags when compiling the `cl` program. It should be set to `-DENABLE_LONG_OPTIONS` if long command line options are desired (for example, "`cl --help`"). Otherwise only one-character options will be recognized ("`cl -h`"). Use of long options requires the `getopt_long()` function, which is a GNU extension. If compilation stops with errors about long options, comment out this variable.

The default options are suitable for compiling with GNU C, with long options enabled.

5.1.2 cliquerconf.h

The file `cliquerconf.h` contains configuration options, which are used in all programs using Cliquer. If some option is not defined in `cliquerconf.h`, the default is used. The file contains the following options:

`setelement`
`ELEMENTSIZE`
: A `setelement` is the basic unsigned integer data type used in sets. It is often fastest to be as long an integer as can fit in the general registers of the CPU. `ELEMENTSIZE` is the number of bits in one `setelement`. It must be 16, 32 or 64, otherwise some modifications to the source code are necessary. One must either define both in `cliquerconf.h`, or neither.

  The default is to use "`unsigned long int`" as `setelement`, and try to determine its size from `ULONG_MAX` defined in `limits.h`. If using the default, it is recommended to run "`make test`" to check successful detection.

`INLINE`
: Many compilers can inline simple functions to make faster code. This option is added in the declaration of several simple functions to instruct the compiler to inline them. If function inlining is not desired, or the compiler does not support it, define it empty.

  The default is to use "`inline`", which is recognized by most modern compilers.

`UNUSED_FUNCTION` For performance reasons, the set handling functions are defined in the file `set.h` as static functions. This may cause spurious warnings about unused functions when compiling. Some compilers, such as GNU C, allow the user to add an "attribute" to the function constraining these warnings.

The default is to use "`__attribute__((unused))`" when compiling with GNU C, or blank otherwise.

`ASSERT(cond)` Defining this blank disables all assertions. This is discouraged, because it allows bugs to go unnoticed easier. See Section 5.4 for details.

5.2 Compiling the Command-Line Utility

After configuration, the command-line utility program `cl` can be compiled by simply typing "`make all`". With the program `cl` one can search for cliques from the command line by providing the graph from a file or standard input. One can use all the features in Cliquer by different command-line options. Type "`cl -h`" for information on the available options. It is useful for simple clique searching and for testing Cliquer.

Additionally, "`make test`" compiles and executes a series of unit tests, that is, tests most of the features in Cliquer with a variety of graphs. Running it is recommended to make sure that compilation was successful and configuration options are correctly set. If any of the tests returns an error, check configuration options and try again.

5.3 Writing Your Own Program

All programs using Cliquer should include `cliquer.h`. This in turn includes the files `set.h`, `graph.h`, `reorder.h`, `misc.h`, and `cliquerconf.h`. The programs should be linked together with `cliquer.o`, `graph.o`, and `reorder.o`. The easiest way to do this is by

```
cc -o basic basic.c cliquer.c graph.c reorder.c
```

where `basic` is replaced by the name of the program. Adding compiler-specific optimization flags will make the resulting program faster.

When using Cliquer a lot, it is easiest to make an entry for the program in `Makefile`. The lines

```
basic: basic.o cliquer.o graph.o reorder.o
	$(CC) $(LDFLAGS) -o $@ basic.o cliquer.o graph.o reorder.o
```

with `basic` replaced by the program name should be enough for most needs. Note that the second line must start with a tab, not eight spaces. One can then compile the program by typing "`make basic`" (where `basic` is replaced by the program name).

5.4 Assertions

Cliquer defines the macro `ASSERT(cond)`, which verifies that the specified condition is true. If `cond` evaluates to `FALSE`, an error message containing the file name, the line number, and the condition of the assertion is printed, and the program execution is terminated. Assertions can be used to check the validness of function arguments and internal variables. For instance, one can check the internal consistency of a graph by

```
ASSERT(graph_test(g,NULL));
```

This is recommended after creating or modifying a graph. Aborting the program execution is justified by the fact that if an assertion fails, it most certainly is the result of a bug in the program. Changing `NULL` in the above example to, for instance, `stderr` would also write a line stating the validness and graph parameters to `stderr`.

Cliquer uses assertions mainly in the clique searching functions. Most set and graph functions do not use them for performance reasons. Even though disabling assertions is possible from `cliquerconf.h`, this is discouraged, as it allows bugs to go unnoticed easier.

Note that the clique searching functions assert that `min_weight` $\leq$ `max_weight` if `max_weight` > 0, even though there exists the "correct" answer that no such cliques exist. This is because asking for cliques with a minimum weight that is larger than the maximum weight is in most cases due to a bug in the code (for example, specifying `min_weight` and `max_weight` in the wrong order). On the other hand, testing for complete graph validness with `graph_test()` is *not* performed automatically, since the check is an $O(n^2)$ operation.

Appendices

A Set-Handling Functions

The following routines are defined in `set.h` for set manipulation:

`set_t set_new(int size)` Returns a set which can contain the values $\{0, 1, \ldots,$ `size-1`$\}$. It can be freed using `set_free()` (*not* `free()`). The value of `size` must be greater than zero.

`void set_free(set_t s)` Frees the memory associated with the set `s`.

`set_t set_resize(set_t s,int size)` Resizes the set `s` to a subset of $\{0, 1, \ldots,$ `size-1`$\}$. If the set contains elements with a value greater than or equal to `size`, they are removed from the set. The value `size` must be greater than zero. The return value is the new set (the old set should not be used anymore).

`SET_ADD_ELEMENT(s,i)`
`SET_DEL_ELEMENT(s,i)`
`SET_CONTAINS(s,i)`
`SET_CONTAINS_FAST(s,i)`
Macros that add, remove and test for element `i` in the set `s`. `SET_CONTAINS(s,i)` works for all `i` ≥ 0 (returning `FALSE` if `i` is greater than the set size), while the others assume that $0 \leq$ `i` $\leq$ `SET_MAX_SIZE(s)-1`. Apart from the allowed range, `SET_CONTAINS_FAST` is equivalent to `SET_CONTAINS`.

`SET_MAX_SIZE(s)`
`SET_ARRAY_LENGTH(s)`
Macros that return the superset size and the `setelement` array length of the set `s`, respectively.

`int set_size(set_t s)` Returns the number of elements that the set `s` contains.

`void set_empty(set_t s)` Removes all elements from the set `s`.

`set_t set_duplicate(set_t s)` Returns a duplicate of the set `s`.

`set_t set_copy(set_t dest,set_t src)` Makes the set `dest` contain the same elements as `src`. If `dest` is `NULL`, this performs the equivalent of `set_duplicate(src)`. If `dest` is smaller than `src`, `dest` is resized to the size of src. Return value is either dest or the set allocated in its stead; use as dest=set_copy(dest,src) to ensure correct behavior.

`int set_return_next(set_t s, int n)` Returns the smallest element of the set s which is greater than n, or -1 if such an element does not exist. One can iterate though all elements in s with

```
int i=-1;
while ((i=set_return_next(s,i)) >= 0) {
    /* i is in set s. */
}
```

`set_t set_intersection(set_t res,set_t a,set_t b)`
`set_t set_union(set_t res,set_t a,set_t b)`
Stores the intersection or union of the sets a and b in the set res, which is resized (or created if NULL) to be at least the size of the larger source operand. Return value is res or the set allocated in its stead. It is *not* allowed that res be either a or b.

`void set_print(set_t s)` Prints size and contents of the set s to stdout. Mainly useful for debugging or simple output.

B Graph-Handling Functions

The following routines are defined in graph.h for graph manipulation:

`graph_t *graph_new(int n)`
Creates a new graph with n vertices. There are no edges in the graph and all vertex weights are set to 1. The value of n must be greater than zero.

`void graph_free(graph_t *g)`
Frees the memory used by the graph g.

`GRAPH_ADD_EDGE(g,i,j)`
`GRAPH_DEL_EDGE(g,i,j)`
`GRAPH_IS_EDGE(g,i,j)`
`GRAPH_IS_EDGE_FAST(g,i,j)`
Macros that add, remove and check for an edge between vertices i and j in the graph g. GRAPH_IS_EDGE(g,i,j) works for all i, j ≥ 0 (returning FALSE if i or j exceeds the order of the graph), while the others assume that $0 \leq$ i, j $\leq$ g->n-1. The order of the parameters i

and `j` is insignificant. Apart from the allowed range, `GRAPH_IS_EDGE_FAST` is equivalent to `GRAPH_IS_EDGE`.

`void graph_resize(graph_t *g,int size)`

Resizes the graph `g` to contain `size` vertices. If `size` $<$ `g->n`, then the vertices {`size`, ..., `g->n-1`} will be removed from the graph. The value of `size` must be greater than zero.

`void graph_crop(graph_t *g)`

Removes the highest valued isolated vertices from the graph `g`, so that the highest valued vertex is not isolated.

`graph_t *graph_read_dimacs_file(char *file)`
`graph_t *graph_read_dimacs(FILE *fp)`

Reads a DIMACS-format graph file [4, 5] from the file stream `fp` or from the file `file`. Automatically detects whether the file is in ASCII or binary format. Returns a newly-allocated graph if successful, otherwise prints an error message to `stderr` and returns `NULL`.

The file format is described in Appendix D. The vertex weights are read from the 'n' lines of the preamble. The 'd', 'v', and 'x' lines are silently ignored. All other unknown lines produce a warning message and are ignored.

`boolean graph_write_dimacs_ascii(g,comment,fp)`
`boolean graph_write_dimacs_ascii_file(g,comment,file)`
`boolean graph_write_dimacs_binary(g,comment,fp)`
`boolean graph_write_dimacs_binary_file(g,comment,file)`

Types: `graph *g, char *comment, FILE *fp, char *file`
These functions write the graph `g` in DIMACS ASCII or binary format to the file stream `fp` or the file `file`. If `comment` is non-NULL, then it is added to the file as a comment. `comment` may not contain newlines.

`int graph_edge_count(graph_t *g)`

Returns the number of edges in the graph `g`.

`int graph_vertex_degree(graph_t *g, int v)`

Returns the degree (the number of adjacent vertices) of vertex `v` in the graph `g`.

`boolean graph_test(graph_t *g,FILE *output)`

Returns `TRUE` iff the graph `g` is a valid graph (symmetric, anti-reflexive, positive weights, total weight less than `INT_MAX`). If `output` is non-NULL, prints a message noting errors or validness to file descriptor `output`.

It is recommended to add for example `ASSERT(graph_test(g,NULL))` after creating or modifying a graph to make sure it is internally correct. See Section 5.4 for details on `ASSERT`.

`int graph_test_regular(graph_t *g)`
: Returns the degree of the regular graph `g`, or -1 if `g` is not regular. Does *not* perform the graph consistency tests done by `graph_test`.

`boolean graph_weighted(graph_t *g)`
: Returns `FALSE` iff all vertex weights in graph `g` are the same (not necessarily 1). To check that all weights are equal to 1, use `!graph_weighted(g) && g->weights[0]==1`.

`void graph_print(graph_t *g)`
: Prints the graph `g` to `stdout` in a simple format. Useful mainly in debugging.

C Ordering Functions

Cliquer defines the following functions that can be used as `reorder_function` in the `clique_options` structure. Each take as arguments the graph and a boolean value which is `TRUE` if a weighted search is being done, `FALSE` otherwise. They return a newly-allocated array of `g->n` `int`s defining the order of the vertices. They do not modify the graph.

`reorder_by_ident` No reordering (identity mapping).

`reorder_by_reverse` Orders vertices in reverse order.

`reorder_by_degree` Orders vertices in order of ascending degree.

`reorder_by_random` Orders vertices randomly. Uses the random number generator `rand()` and seeds the value from the current time.

`reorder_by_weighted_greedy_coloring` Orders vertices as defined in Section 2.4 in the weighted case.

`reorder_by_unweighted_greedy_coloring` Orders vertices as defined in Section 2.4 in the unweighted case.

`reorder_by_greedy_coloring` Either of the previous two, depending on whether a weighted or unweighted search is being performed.

`reorder_by_default` The default ordering function, currently `reorder_by_greedy_coloring`.

Additionally, the following functions are defined to allow for more complex orderings.

`void reorder_set(set_t s,int *order)` Orders the elements in the set `s` according to the mapping `i` $\mapsto$ `order[i]`, $0 \leq$ `i` $\leq$ `SET_MAX_SIZE(s)-1`.

`void reorder_graph(graph_t s,int *order)` Orders the vertices of the graph `g` according to the mapping `i` $\mapsto$ `order[i]`, $0 \leq$ `i` $\leq$ `g->n-1`.

`int *reorder_duplicate(int *order,int n)` Duplicates the mapping `order` of size `n`.

`void reorder_invert(int *order,int n)` Inverts the mapping `order`, so that `new[old[i]] == i` for all $0 \leq$ `i` $\leq$ `n-1`.

`boolean reorder_is_bijection(int *order,int n)` Returns `TRUE` if the mapping `order` is a bijection in $\{0, 1, \ldots,$ `n-1`$\}$.

For example, the following code orders the vertices of the graph `g` first randomly, and after that with the default ordering function:

```
int *order;
set_t s;

order=reorder_by_random(g,FALSE);
reorder_graph(g,order);
reorder_invert(order,g->n);
s=clique_find_single(g,0,0,FALSE,NULL);
reorder_set(s,order);
set_print(s);
```

D DIMACS Graph File Format

The DIMACS file format is a common format for describing graphs. The graphs can either be in human-readable ASCII form [4] or in binary form [5]. The binary form takes less space for graphs with an edge density greater than approximately 1.2 %. The formats are described shortly here.

D.1 ASCII Format

The ASCII files consist of textual lines with fields that are separated by at least one blank space. The first field of each line consists of one character, and describes the line type. The vertices in the file are numbered $\{1, 2, \ldots, n\}$. Cliquer automatically changes the numbering to $\{0, 1, \ldots, n-1\}$ by decreasing the values by one when reading the files, and increasing by one when writing. The lines recognized by Cliquer are as follows:

`c Comment line.`
: Lines beginning with '`c`' are comments and are ignored.

`p FORMAT NODES EDGES`
: Each file contains one '`p`' line, which describes the dimensions of the graph. `FORMAT` is for consistency with older formats, and should contain the word "`edge`". The number of vertices and edges in the graph are given in the fields `NODES` and `EDGES`, respectively. Cliquer ignores the `FORMAT` and `EDGES` fields when reading a graph, but they must be present.

`n ID VALUE`
: Assigns the vertex `ID` weight `VALUE`. Vertices that have no corresponding '`n`' line will have the default weight of 1.

`e W V`
: Specifies that there is an edge between vertices `W` and `V`. The line is *not* repeated as "`e V W`".

`d, v, x`
: The '`d`', '`v`', and '`x`' lines define parameters that were used to generate the graph. Refer to [4] for details. These lines are ignored by Cliquer.

When reading a graph, if a line starts with a one-character field that is not mentioned above, a warning message is printed to `stderr` and the line is ignored.

D.2 Binary Format

The binary format files consist of three parts: the first line, a textual preamble, and a binary block. The first line contains an integer describing the length of the preamble, in characters. Next, the preamble contains the same lines as in the ASCII format, except for the '`e`' lines. (Cliquer also accepts '`e`' lines in the preamble, and adds extra edges correspondingly.) Finally, the binary block contains the lower triangular part of the adjacency matrix of the graph in binary format. There are $\lceil i/8 \rceil$ bytes corresponding to vertex v_i, where $i \in \{1, 2, \ldots, n\}$. The bits are used in a "left-to-right" manner, so that the first vertex is in the most significant bit.

Note that although not specified in [5], Cliquer numbers the vertices $\{1, 2, \ldots, n\}$ in the preamble also in the binary case. This is significant especially for weighted graphs, when we need to list the weights.

E Example Programs

The following program takes the name of a DIMACS graph file on the command line, reads it, searches for a single maximum-weight clique, and then prints it.

The program can be compiled as is explained in Section 5.3. An example run might look like the following (`rand-600-0.3.b` in this example contains a random graph with 600 vertices and edge density 0.3):

The next example program finds all binary codes with prescribed length, size, and minimum distance.

Using this program, we may count the number of binary perfect codes of length 7 and minimum distance 3.

```
$ ./hamming 7 16 3
Number of codes: 240
```

Note that for this and other combinatorial problems, the graph has a large automorphism group. This group can be utilized to speed up the search significantly, which is essential when searching for larger codes [9, 12].

Acknowledgment

This research was supported in part by the Academy of Finland under Grant 100500. The authors would like to thank Harri Haanpää and Petteri Kaski for many useful comments and suggestions.

References

[1] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, The maximum clique problem, in: D.-Z. Du and P. M. Pardalos (Eds.), Handbook of Combinatorial Optimization, Supplement Volume A, Kluwer, Dordrecht, 1999, pp. 1–74.

[2] A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith, A new table of constant weight codes, *IEEE Trans. Inform. Theory* **36** (1990), 1334–1380.

[3] R. Carraghan and P. M. Pardalos, An exact algorithm for the maximum clique problem, *Oper. Res. Lett.* **9** (1990), 375–382.

[4] DIMACS, Clique and coloring problems graph format, `ftp://dimacs.rutgers.edu/pub/challenge/graph/doc/ccformat.dvi`, 26.8.2002.

[5] DIMACS, DIMACS format for storing undirected graphs in binary files, `ftp://dimacs.rutgers.edu/pub/challenge/graph/translators/binformat/README.binformat`, 26.8.2002.

[6] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, New York, 1979.

[7] B.D. McKay, *nauty* user's guide (version 1.5), Computer Science Department, Australian National University, Tech. Rep. TR-CS-90-02, 1990.

[8] K. J. Nurmela, M. K. Kaikkonen, and P. R. J. Östergård, New constant weight codes from linear permutation groups, *IEEE Trans. Inform. Theory* **43** (1997), 1623–1630.

[9] P. R. J. Östergård, Classification of binary/ternary one-error-correcting codes, *Discrete Math.* **223** (2000), 253–262.

[10] P. R. J. Östergård, A new algorithm for the maximum-weight clique problem, *Nordic J. Comput.* **8** (2001), 424–436.

[11] P. R. J. Östergård, A fast algorithm for the maximum clique problem, *Discrete Appl. Math.* **120** (2002), 195–205.

[12] P. R. J. Östergård, T. Baicheva, and E. Kolev, Optimal binary one-error-correcting codes of length 10 have 72 codewords, *IEEE Trans. Inform. Theory* **45** (1999), 1229–1231.