Logic Programming with Max-Clique and its Application to Graph Coloring (Tool Description)*

Michael Codish[1], Michael Frank[2], Amit Metodi[3], and Morad Muslimany[4]

1 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
`mcodish@cs.bgu.ac.il`

2 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
`frankm@cs.bgu.ac.il`

3 **Cadence Design Systems, Israel**
`ametodi@cadence.com`

4 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
`moradm@cs.bgu.ac.il`

Abstract

This paper presents `pl-cliquer`, a Prolog interface to the `cliquer` tool for the maximum clique problem. Using `pl-cliquer` facilitates a programming style that allows logic programs to integrate with other tools such as: Boolean satisfiability solvers, finite domain constraint solvers, and graph isomorphism tools. We illustrate this programming style to solve the Graph Coloring problem, applying a symmetry break that derives from finding a maximum clique in the input graph. We present an experimentation of the resulting Graph Coloring solver on two benchmarks, one from the graph coloring community and the other from the examination timetabling community. The implementation of `pl-cliquer` consists of two components: A lightweight C interface, connecting `cliquer`'s C library and Prolog, and a Prolog module which loads the library. The complete tool is available as a SWI-Prolog module.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Logic Programming, Constraints, Maximum Clique

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.5

1 Introduction

The maximum clique problem, which is about finding the largest set of pairwise adjacent vertices in a graph, has been studied extensively both in theory [18, 10, 16, 2, 3, 14, 15, 5, 33] and in practice [29, 28, 19, 6, 7]. Several tools have been developed recently, which tackle the maximum clique problem [28, 21, 20, 29]. One of these tools is `cliquer` [27], which uses an exact branch-and-bound algorithm in the search for maximum cliques. `cliquer` consists of a collection of C routines, which can be compiled to either a standalone executable or a shared library.

* Supported by the Israel Science Foundation grant 182/13 and by the Lynn and William Frankel Center for Computer Science

The maximum clique problem is related to several other well-known graph problems. The chromatic number of any graph is bound from below by its maximum clique size, and the chromatic number of perfect graphs is identical to its maximum clique size. The size of the largest independent set in any graph G is identical to the size of the maximum clique in the complement graph of G (i.e., the graph in which u and v are adjacent if and only if they are not adjacent in G). The fast detection of cliques may also assist in the search for Ramsey graphs, which are characterized by their clique sizes. Further applications of maximum cliques can be found in bioinformatics (e.g., to infer evolutionary trees [11] and predict protein structures [32]), and in the analysis of random processes, where maximum cliques are sought in a dependency graph [13].

Integrating a (maximum) clique finding tool into a Logic Programming tool-chain has the potential to benefit search when solving hard graph problems. Such a tool can be used, for example, as a preprocessing step for instances of the graph coloring problem. After preprocessing, instances are processed by a constraint solver, or a SAT solver to find a solution. Moreover, using a tool-chain fitted for Prolog leads to new logic programming styles, such as those presented in [9], which integrates SAT solvers with Prolog, or [12] which integrates the `nauty` graph automorphism library with Prolog.

In this paper we demonstrate how a maximum clique tool can be integrated with Prolog, resulting in a library for SWI-Prolog [34] containing predicates that find maximum cliques in graphs. Moreover, we demonstrate how this library is integrated with an entire tool-chain written for Prolog. This tool-chain includes a collection of problem solving tools such as SAT and CSP solvers (e.g., [9]), finite-domain constraint compilers (e.g., [25, 26]), as well as a collection of additional symmetry breaking predicates which allow us to detect and prune equivalent solutions. We also observe that our tool-chain, augmented by `cliquer`, is competitive for graph coloring with results reported in the literature, and in some cases obtains previously unreported solutions.

The rest of this paper is structured as follows. Section 2 introduces the basic definitions and notations used throughout. Section 3 describes and illustrates the use of the `pl-cliquer` interface. Section 4 introduces the graph coloring problem, as well as a graph coloring solver, which is extended with optimizations based on the identification of a maximum clique. This example demonstrates how `pl-cliquer` augments an existing tool-chain. Section 5 defines the exam timetabling problem, as an application of the graph coloring problem. Sections 4 and 5 also present selected results of an experimentation using the standard graph coloring and exam timetabling benchmarks. Section 6 contains some technical details on `pl-cliquer`, and Section 7 concludes. Appendices A and B present results on the full benchmarks.

2 Preliminaries

A graph $G = (V, E)$ consists of a set of vertices $V = [n] = \{\ 1, \dots, n\ \}$ and a set of edges $E \subseteq V \times V$. In the context of the tools that we present in this paper, it is natural to consider only simple graphs. Meaning, we assume that graphs are undirected, and there are no self loops or multiple edges. The degree of a vertex $v \in V$ is denoted by $deg(v)$ and it is equal to the number of neighbors v has. The maximum degree of a graph $G = (V, E)$ is denoted $\Delta(G)$ and it is equal to the maximum over all vertex degrees. In this paper we represent graphs as Boolean adjacency matrices, a list of `N` length-`N` lists. We also make use of the DIMACS file format for graph representation, which contains the line `e` i j for every edge (i, j) of the graph, such that $i < j$.

A function $w\colon V \to \mathbb{N} \setminus \{0\}$, which assigns positive weights to the vertices of the graph is called a positive weight function. As an abuse of notation, we denote the weight of a set of vertices $U \subseteq V$ by $w(U) = \sum_{u \in U} w(u)$. For the sake of simplicity, throughout this paper, we assume that vertex weights are 1, thereby identifying the weight of U with its size.

A clique $C \subseteq V$ of $G = (V, E)$ is a set of vertices that are pairwise connected, meaning that if $u, v \in C$ then $(u, v) \in E$. If $|C| = k$ we call C a k-clique. The clique number of a graph G is denoted $\omega(G)$ and is the number of vertices in a largest clique of G. A clique $C \subseteq V$ such that $|C| = \omega(G)$ is called a maximum clique. The max-clique problem is about finding a maximum clique in a given graph G. The max-weighted-clique problem is about finding a clique $C \subseteq V$ such that the weight of C is maximal. The max-clique problem is known to be NP-Hard [18].

A vertex k-coloring of a graph $G = (V, E)$, for $k \in \mathbb{N}$, is a mapping $c\colon V \to [k]$ ($[k] = \{\ 1, \ldots, k\ \}$) such that $(u, v) \in E$ implies that $c(u) \neq c(v)$. The chromatic number of a graph G is denoted $\chi(G)$ and it is the smallest number k such that the vertices of G are k-colorable. The graph coloring problem is about finding a k-coloring of a given graph G. The minimum graph coloring problem is about finding a k-coloring of a given graph G such that $k = \chi(G)$. The graph coloring problem is known to be NP-Complete, and the minimum graph coloring problem is NP-Hard [18].

3 Interfacing Prolog with `cliquer`'s C library

The `pl-cliquer` interface is implemented using the foreign language interface of SWI-Prolog [34]. The C library of `cliquer` is linked against corresponding C code written for Prolog, which contains the low-level Prolog predicates connecting `cliquer` with Prolog. These low-level predicates are wrapped by a Prolog module, which extends their functionality and provides five high-level predicates. The five high-level predicates, available through the module are:

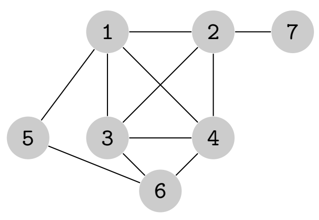
1. `graph_read_dimacs_file/5`
2. `clique_find_single/4`
3. `clique_find_multi/5`
4. `clique_find_n_sols/6`
5. `clique_print_all/6`

In the following we describe these in more detail and present several usage examples.

3.1 The `clique_read_dimacs_file/5` predicate

The call `graph_read_dimacs_file(DIMACS, NVert, Weights, Matrix, Options)` applies to convert a graph represented in the standard DIMACS format to a Prolog representation as a Boolean adjacency matrix (list of lists). Figure 1 illustrates an example graph with seven vertices, its DIMACS representation, and its corresponding adjacency matrix. If the DIMACS representation resides in the file `example.dimacs` then the following call:

```
?- graph_read_dimacs_file('example.dimacs', NVert, Weights, Matrix, []).
Matrix = [[0,1,1,1,1,0,0],
          [1,0,1,1,0,0,1],
          [1,1,0,1,0,1,0],
          [1,1,1,0,0,1,0],
          [1,0,0,0,0,1,0],
          [0,0,1,1,1,0,0],
          [0,1,0,0,0,0,0]],
NVert = 7,
Weights = [1,1,1,1,1,1,1]
```



```
p edge 7 11
e 1 2
e 1 3
e 1 4
e 1 5
e 2 3
e 2 4
e 2 7
e 3 4
e 3 6
e 4 6
e 5 6
```

```
[[0,1,1,1,1,0,0],
 [1,0,1,1,0,0,1],
 [1,1,0,1,0,1,0],
 [1,1,1,0,0,1,0],
 [1,0,0,0,0,1,0],
 [0,0,1,1,1,0,0],
 [0,1,0,0,0,0,0]]
```

Figure 1 An example graph (left), its DIMACS representation (middle), and corresponding adjacency matrix (right).

will bind (as indicated) the variable `Matrix` to the matrix depicted also on the right of Figure 1, the variable `NVert` to the number of vertices in the graph, i.e., 7, and the variable `Weights` to a list of ones (since the vertices of this graph are without weights). The last argument (an empty list in the example) specifies a list of options which may contain the options `edge(Value)` and `non_edge(Non)`. These respectively determine the symbols used to represent edges and non-edges in the adjacency matrix `Matrix`. The empty list indicates default settings. By default, edges are represented by the symbol 1 and non-edges by the symbol 0.

3.2 The `clique_find_single/4` predicate

The `clique_find_single/4` predicate provides an interface to the `cliquer` routine by the same name. It finds a single clique in the input graph. The predicate takes the form `clique_find_single(NVert, Graph, Clique, Options)`. The first argument indicates the number of vertices in the graph, the second argument is an adjacency matrix representing the graph, and the third argument is the output clique. The last argument is a list of options which fine-tune the behavior of `pl-cliquer`, constraining the size and weight of the sought after clique.

`Options` may include `min_weight(Min)` and `max_weight(Max)` limiting the clique weight to be between `Min` and `Max`. In order to obtain a maximum clique (which is the default behavior) both `Min` and `Max` should be 0. Other options are `maximal(Maximal)`, where `Maximal` is a Boolean specifying whether only maximal cliques should be found, `static_ordering(Order)` where `Order` is a permutation of $\{\ 1 \dots n\ \}$ specifying the order in which the n vertices of the graph are backtracked over by `cliquer`'s algorithm. The option `weights(Weights)` specifies a list of n `Weights` associated with the n vertices of the graph (by default all weights are 1). The empty list indicates default settings.

For example, calling `clique_find_single(NVert, Graph, Clique, Options)` with the graph from Figure 1, unifies `Clique` with the vertices `[1,2,3,4]`, which are easily verified as the maximum clique. Notice that during the runtime of `cliquer` additional information is printed to screen, which indicates the current workload and the maximal clique found so far.

```
 ?- Graph =  [[0,1,1,1,1,0,0], [1,0,1,1,0,0,1] |  ... ],
    clique_find_single(7, Graph, Clique, []).


  2/7 (max  1)  0.00 s  (0.00 s/round)
  4/7 (max  2)  0.00 s  (0.00 s/round)
  5/7 (max  3)  0.00 s  (0.00 s/round)
  7/7 (max  4)  0.00 s  (0.00 s/round)
  size=4 (max 7) 0 1 2 3

Graph = [[0, 1, 1, 1, 1, 0, 0] | ... ],
Clique = [1, 2, 3, 4].
```

3.3 The `clique_find_n_sols/6` predicate

The call `clique_find_n_sols(MaxSols, NVert, Graph, Sols, Total, Opts)` allows to find several cliques in the input graph. The first argument, `MaxSols`, indicates the maximal number of cliques to be found. The second argument `NVert` indicates the number of vertices in the graph, and third argument is an adjacency matrix representing the graph. The fourth argument `Sols`, is unified with the cliques that are found in the graph, and the fifth argument is unified with the number of cliques found. The last argument is the option list, which is identical to the one for `clique_find_single/4`.

For example, calling `clique_find_n_sols/6`, with the graph from Figure 1, with options indicating that at most ten cliques of weight 3 and 4 should be found, also indicating these cliques may not be maximal – returns a result that implies that six such cliques exist, five cliques of weight 3 and one clique of weight 4, and they are: `[[1, 2, 3], [2, 3, 4], [1, 2, 3, 4], [1, 3, 4], [1, 2, 4], [3, 4, 6]]`.

```
?- Graph =  [[0,1,1,1,1,0,0], [1,0,1,1,0,0,1] | ... ],
   NVert = 7, MaxSols = 10,
   Options = [min_weight(3), max_weight(4), maximal(false)],
   clique_find_n_sols(MaxSols, NVert, Graph, Sols, Total, Options).

Graph = [[0, 1, 1, 1, 1, 0, 0] | ... ],
Sols = [[1, 2, 3], [2, 3, 4], [1, 2, 3, 4], [1, 3, 4], [1, 2, 4], [3, 4, 6]],
Total = 6.
```

3.4 The `clique_find_multi/5` predicate

The `clique_find_multi/5` is similar in nature to `clique_find_n_sols/6`, except that it backtracks over the cliques that are found instead of collecting them in a list. This predicate takes the form `clique_find_multi(MaxSols, NVert, Graph, Sol, Opts)`, with parameters identical in description to those of `clique_find_n_sols/6` except that `Sol` will be unified with a single clique, which will change upon backtracking.

For example, calling `clique_find_multi/5`, with the graph from Figure 1, and options indicating that at most three cliques with weights between 3 and 4 should be found, such that these cliques may not be maximal – the call returns a result that implies that at least three such cliques exist: two cliques of weight 3 and one clique of weight 4, and they are: `[1, 2, 3]`, `[2, 3, 4]` and `[1, 2, 3, 4]`.

```
?- Graph =  [[0,1,1,1,1,0,0], [1,0,1,1,0,0,1] | ... ],
   NVert = 7, MaxSols = 3,
   Options = [min_weight(3), max_weight(4), maximal(false)],
   clique_find_multi(MaxSols, NVert, Graph, Sol, Total, Options).


Graph = [[0, 1, 1, 1, 1, 0, 0] | ... ],
Sol = [1, 2, 3] ;
Sol = [2, 3, 4] ;
Sol = [1, 2, 3, 4] ;
false.
```

3.5 The clique_print_all/6 predicate

The clique_print_all/6 predicate is primarily intended for debugging and exploring, and it will print all the cliques of a graph which comply to certain constraints. The predicate takes the form clique_print_all(NVert, Min, Max, Maximal, Graph, Total). For example, the following call to clique_print_all/6 will print all of the cliques in the example graph of Figure 1, which contain at least two vertices and at most three vertices, and unify Total with the total number of lines printed.

```
 ?- NVert = 7, Min = 2, Max = 3, Maximal = false,
    Graph = [[0,1,1,1,1,0,0],
             [1,0,1,1,0,0,1],
             [1,1,0,1,0,1,0],
             [1,1,1,0,0,1,0],
             [1,0,0,0,0,1,0],
             [0,0,1,1,1,0,0],
             [0,1,0,0,0,0,0]],
    clique_print_all(NVert, Min, Max, Maximal, Graph, Total).
[1, 2]
[2, 3]
[1, 2, 3]
[1, 3]
[3, 4]
[2, 3, 4]
[1, 3, 4]
[2, 4]
[1, 2, 4]
[1, 4]
[1, 5]
[5, 6]
[4, 6]
[3, 4, 6]
[3, 6]
[2, 7]
Total = 16.
```

The predicate clique_print_all/6 does not print out the graph, or the edge list. This is because in many cases, the graph is very large, and printing these details provides no constructive effect. Moreover, while the list of edges is available to cliquer when parsing the graph, it is not available to the interface of pl-cliquer. Nevertheless, one may extract a list of graph edges using an auxiliary predicate included with the pl-cliquer source code.

4 Solving the Graph Coloring Problem

The graph coloring problem has been studied vigorously, for both theoretical and practical purposes. Some of the real world applications of the graph coloring problem include: timetabling problems (e.g., [22]), frequency assignment (e.g., [1]) and register allocation (e.g., [8]). We demonstrate, using a logic program tool-chain augmented by `pl-cliquer`, a solution for the graph (vertex) coloring problem, with an application for the minimum examination timetabling schedule problem. We demonstrate how `pl-cliquer` integrates with an existing tool-chain, all specified as part of the Prolog programming process. We compare our graph coloring application to previous work using to two standard benchmark sets [7, 17], one from the graph coloring community, and the other from the exam timetabling community.

In the graph coloring problem we are given a graph $G = ([n], E)$ and a natural number $k > 0$, and we seek a labeling $c\colon [n] \to [k]$ of the graph vertices such that $(i, j) \in E \implies c(i) \neq c(j)$. In the minimum graph coloring problem, we seek the smallest k for which such a labeling exists. Both the graph coloring problem and the minimum graph coloring problem are known to be NP-Hard [18].

In practical scenarios, solving the graph coloring problem has often involved formulating it as an integer linear program [4] or as a constraint model [31]. We solve the graph coloring problem by modeling it using a constraint language and then applying the finite-domain constraint compiler `BEE` [25, 26], which stands for **B**en-Gurion University **E**qui-propagation **E**ncoder (written in Prolog), to encode it to an instance of Boolean satisfiability which is then solved using an underlying SAT solver (through its Prolog interface [9]). In the configuration for this paper we use CryptoMiniSAT v2.5.1 as the underlying SAT solver.

When describing our approach we distinguish between the constraint model for the decision problem: given a graph G and a number k — does there exist a vertex coloring with at most k colors? and the optimization problem — given a graph G what is the smallest number k for which there exists a vertex coloring with at most k colors? To model the optimization problem we apply the minimization option of the `BEE` solver which incrementally refines the number k for which the corresponding decision problem has a solution. The remainder of this section focuses on implementing the graph coloring decision problem, on which we later perform the minimization.

4.1 The Constraint Model for Graph Coloring

The basic constraint model is straightforward: for a graph $G = (V, E)$ and a given number (of colors) k, each vertex $u \in V$ is affiliated with a finite domain integer variable I_v taking a value in the range $[1, \dots, k]$ representing its color. For each edge $(u, v) \in E$, a constraint $I_u \neq I_v$ is added to the model, forcing distinct colors between adjacent vertices.

The following `graphColoring/3` predicate lists the high level Prolog code which implements this encoding for the graph coloring problem. Given a Boolean adjacency matrix `M` for a graph with `N` vertices, we represent a coloring of that graph as a list of `N` values, such that the value in position `I` of the list is the color of vertex `I`. The `graphColoring/3` predicate takes the following form: `graphColoring(Graph, KColors, Coloring)`. The first parameter is the adjacency matrix for the graph to be colored, the second parameter is the number of colors, and the last parameter will be unified with the coloring.

The call in the first line of the definition of `graphColoring/3` generates a constraint model that is satisfiable if and only if `Graph` has a coloring with `KColors` colors. It also generates a `Map` which relates the integer variables (the colors per vertex) with their Boolean

```
% Initial Coloring              % Different Colors
[A,B,C,D,E,F,G]                 int_neq(A,B)
                                int_neq(A,C)
% Coloring Declaration          int_neq(A,D)
new_int(A,1,5)                  int_neq(A,E)
new_int(B,1,5)                  int_neq(B,C)
new_int(C,1,5)                  int_neq(B,D)
new_int(D,1,5)                  int_neq(B,G)
new_int(E,1,5)                  int_neq(C,D)
new_int(F,1,5)                  int_neq(C,F)
new_int(G,1,5)                  int_neq(D,F)
                                int_neq(E,F)
```

Figure 2 Example of the constraint model for the graph in Figure 1.

(bit-blasted) representation. The second line is a call to BEE which compiles the constraint model to CNF, and the third line contains a call to the underlying SAT solver. The last line of `graphColoring/3` translates the coloring from BEE's (bit-blasted) representation of integers to that of Prolog.

```
graphColoring(Graph, KColors, Coloring) :-
    encode(coloring(Graph, KColors), Map, Constr),
    bCompile(Constr, CNF),
    sat(CNF),
    decode(Map, Coloring).
```

As an example, consider the graph in Figure 1. A call to `graphColoring(Graph, 5, Coloring)` would result in finding a coloring of that graph with 5 colors, and unifying `Coloring` with it. The first line of `graphColoring/3` will generate the constraint model, the second and third lines will compile and solve it. Figure 2 illustrates the constraint model for the example graph in Figure 1. The left column details the initial (unknown) coloring generated by the encoding process, as well as the variable declarations, and the right column lists the constraints forbidding adjacent vertices from having the same color. After solving the problem, in line 4 of `graphColoring/3` the coloring is translated back to Prolog values, and `Coloring` is unified with `[1,2,3,4,5,1,3]`, for example.

4.2 Throwing `pl-cliquer` into the mix

It is often the case that fast detection and iteration of cliques can be used to simplify and break symmetries in graph related problems [33, 6, 24]. In the case of graph coloring, when coloring the graph $G = (V, E)$, a clique $C \subseteq V$ must correspond to a set of vertices which are labeled by different colors, since C contains a set of mutually adjacent vertices. Given a clique $C \subseteq V$ of the graph, it is possible to arbitrarily label the vertices of C with $|C|$ different colors, thereby breaking symmetries and establishing a lower bound on $\chi(G)$.

The following `graphColoring/3` predicate is augmented with a preprocessing step which applies `pl-cliquer` in order to find a maximum clique of the input graph. This clique is passed to the encoder, to be used by it during the generation of the constraint model. The first and second lines of the predicate find a maximum clique in `Graph` and unify it with `MaxClique`. The remaining lines of code follow the same outline as in `graphColoring/3` described in Section 4.1, with the exception that `MaxClique` is taken into account as part of the encoding process.

```
% Partial Coloring              % Different Colors
[1,2,3,4,E,F,G]                 int_neq(1,E)
                                int_neq(2,G)
% Coloring Declaration          int_neq(3,F)
new_int(E,1,5)                  int_neq(4,F)
new_int(F,1,5)                  int_neq(E,F)
new_int(G,1,5)
```

Figure 3 Example of the constraint model for the graph in Figure 1 with a partial coloring.

```
graphColoring(Graph, KColors, Coloring) :-
    length(Graph, NVert),
    clique_find_single(NVert, Graph, MaxClique, []),
    encode(coloring(Graph, KColors, MaxClique), Map, Constr),
    bCompile(Constr, CNF),
    sat(CNF),
    decode(Map, Coloring).
```

The encoding process initially assigns colors $\{1, \ldots, |\texttt{MaxClique}|\}$ to the vertices of `MaxClique`. The remaining vertices are associated with finite-domain variables taking values between $\{1, \ldots, \texttt{KColor}\}$, representing their color. Finally, constraints are added which prevent adjacent vertices from sharing a color.

As an example, consider the graph in Figure 1. A call to the augmented predicate **`graphColoring(Graph, 5, Coloring)`** would result in finding a coloring of that graph with 5 colors, and unifying `Coloring` with it. The first line of **`graphColoring/3`** determines the number of vertices in the graph. The second line of **`graphColoring/3`** finds a maximum clique and unifies `MaxClique` with a list of its vertices. For example, given the graph in Figure 1, the solver might unify `MaxClique` with `[1,2,3,4]`. The third line generates the constraint model, and the fourth line compiles the constraint model to CNF. The fifth line calls a SAT solver and the sixth line translates the result to Prolog values, resulting in either a coloring for the graph, or an unsatisfiable result, implying the graph can not be colored by `KColor` colors. Notice that since the maximum clique is known, the encoding process may introduce a partial coloring of the graph, which substantially reduces the constraint model, because constraints involving clique vertices may now be omitted. Figure 3 illustrates the constraint model for the example graph in Figure 1. The left column details the partial coloring generated by the encoding process, as well as the variable declarations, and the right column lists the constraints forbidding adjacent vertices from having the same color.

4.3 Additional Optimizations & Results

In this section we mention additional optimizations that can be made, when solving the graph coloring problem, given that the maximum clique of the graph is known. So far, the graph coloring solver is composed of: (1) a preprocessing step which embeds the colors of a maximum clique, and (2) a constraint model which is translated to CNF and solved by a SAT solver. In addition to (1) and (2), we have also implemented, as a secondary preprocessing step, the optimizations discussed in [23]. These optimizations reduce symmetries, as well as the instance size leading to an improved constraint model. Once solved, the improved model is passed through a postprocessing step, also described in [23] in order to obtain the final coloring. For the sake of brevity, we refer the interested reader to the relevant paper [23] for a complete description of the optimizations. We tested this method on the DIMACS coloring instances, which were introduced in the Second DIMACS Implementation Challenge [17].

Table 1 Satisfiable Dimacs Instances Results.

		with `cliquer`		without `cliquer`	
Instance	k	time	status	time	status
anna.col	11	0.02	sat(`BEE`)	0.06	sat
david.col	11	0.02	sat(`BEE`)	0.05	sat
DSJC125.1.col	5	0.06	sat	0.03	sat
DSJR500.1.col	12	0.07	sat	0.31	sat
DSJR500.5.col	122	4661.28	*memory*	∞	*memory*
fpsol2.i.1.col	65	0.03	sat(`BEE`)	4.13	sat
fpsol2.i.2.col	30	0.07	sat	1.31	sat
fpsol2.i.3.col	30	0.07	sat	1.31	sat
games120.col	9	0.05	sat	0.07	sat
huck.col	11	0.02	sat(`BEE`)	0.04	sat
inithx.i.1.col	54	0.12	sat	4.38	sat
inithx.i.2.col	31	0.3	sat	2.03	sat
inithx.i.3.col	31	0.33	sat	2.2	sat
jean.col	10	0.02	sat(`BEE`)	0.03	sat
le450_15a.col	15	1.6	sat	0.78	sat
le450_15b.col	15	0.76	sat	0.68	sat
le450_25a.col	25	0.57	sat	1.12	sat
le450_25b.col	25	0.7	sat	1.13	sat
le450_5a.col	5	0.18	sat	0.22	sat
le450_5b.col	5	0.18	sat	0.23	sat
le450_5c.col	5	0.26	sat	0.36	sat
le450_5d.col	5	0.27	sat	0.34	sat
miles1000.col	42	0.02	sat(`BEE`)	1.29	sat
miles1500.col	73	0.02	sat(`BEE`)	3.83	sat
miles250.col	8	0.02	sat(`BEE`)	0.04	sat
miles500.col	20	0.02	sat(`BEE`)	0.23	sat
miles750.col	31	0.03	sat	0.69	sat
mulsol.i.1.col	49	0.02	sat(`BEE`)	0.98	sat
mulsol.i.2.col	31	0.13	sat	0.62	sat
mulsol.i.3.col	31	0.13	sat	0.64	sat
mulsol.i.4.col	31	0.14	sat	0.64	sat
mulsol.i.5.col	31	0.14	sat	0.63	sat
myciel3.col	4	0.03	sat	0.01	sat
myciel4.col	5	0.03	sat	0.01	sat
myciel5.col	6	0.03	sat	0.01	sat
myciel6.col	7	0.04	sat	0.04	sat
myciel7.col	8	0.09	sat	0.11	sat
queen5_5.col	5	0.03	sat	0.01	sat
queen6_6.col	7	0.03	sat	0.04	sat
queen7_7.col	7	0.04	sat	0.04	sat
queen8_12.col	12	0.1	sat	0.15	sat
queen8_8.col	9	0.92	sat	0.24	sat
queen9_9.col	10	6.98	sat	14.42	sat
queen10_10.col	11	21638.0	sat	2168.42	sat
school1.col	14	66.91	sat	1.34	sat
school1_nsh.col	14	27.08	sat	1.06	sat
zeroin.i.1.col	49	0.02	sat(`BEE`)	1.05	sat
zeroin.i.2.col	30	0.03	sat	0.56	sat
zeroin.i.3.col	30	0.03	sat	0.56	sat

Selected results for this set of benchmarks are given in Tables 1 and 2, a more complete set of results can be found in (B).

Table 1 compares solving times for the satisfiable instances of the DIMACS benchmarks with and without our augmented tool-chain. Table 2 similarly compares the solving times of the unsatisfiable instances. Both tables share the same structure: the first column lists the instance name and the second column lists the coloring size we seek. In Table 1 these values k correspond to the best known colorings described in the literature [23, 24]. In Table 2 these correspond to corresponding values $k-1$, the largest values for which there does not exist a coloring. The third and fifth columns detail the solving times with and without our augmented tool-chain, and the fourth and sixth columns detail the means by which the instance was solved. The means by which an instance was solved may be one of the following: (a) sat(`BEE`) – which means that `BEE` solved the constraints without calling the

Table 2 Unsatisfiable Dimacs Instances Results.

		with `cliquer`		without `cliquer`	
Instance	k	time	status	time	status
anna.col	10	0.01	unsat(`cliquer`)	1.63	unsat
david.col	10	0.01	unsat(`cliquer`)	0.79	unsat
DSJC125.1.col	4	0.03	unsat(`BEE`)	0.03	unsat
DSJR500.1.col	11	0.02	unsat(`cliquer`)	4.79	unsat
DSJR500.5.col	121	4597.34	unsat(`cliquer`)	∞	*memory*
fpsol2.i.1.col	64	0.02	unsat(`cliquer`)	∞	*timeout*
fpsol2.i.2.col	29	0.02	unsat(`cliquer`)	∞	*timeout*
fpsol2.i.3.col	29	0.02	unsat(`cliquer`)	∞	*timeout*
games120.col	8	0.01	unsat(`cliquer`)	0.82	unsat
huck.col	10	0.01	unsat(`cliquer`)	0.52	unsat
inithx.i.1.col	53	0.05	unsat(`cliquer`)	∞	*timeout*
inithx.i.2.col	30	0.05	unsat(`cliquer`)	∞	*timeout*
inithx.i.3.col	30	0.05	unsat(`cliquer`)	∞	*timeout*
jean.col	9	0.01	unsat(`cliquer`)	0.29	unsat
le450_15a.col	14	0.02	unsat(`cliquer`)	535.52	unsat
le450_15b.col	14	0.02	unsat(`cliquer`)	432.15	unsat
le450_25a.col	24	0.02	unsat(`cliquer`)	∞	*timeout*
le450_25b.col	24	0.02	unsat(`cliquer`)	∞	*timeout*
le450_5a.col	4	0.01	unsat(`cliquer`)	0.18	unsat
le450_5b.col	4	0.01	unsat(`cliquer`)	0.18	unsat
le450_5c.col	4	0.01	unsat(`cliquer`)	0.3	unsat
le450_5d.col	4	0.01	unsat(`cliquer`)	0.32	unsat
miles1000.col	41	0.01	unsat(`cliquer`)	∞	*timeout*
miles1500.col	72	0.01	unsat(`cliquer`)	∞	*timeout*
miles250.col	7	0.01	unsat(`cliquer`)	0.1	unsat
miles500.col	19	0.01	unsat(`cliquer`)	∞	*timeout*
miles750.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.1.col	48	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.2.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.3.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.4.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.5.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
myciel3.col	3	0.03	unsat	0.01	unsat
myciel4.col	4	0.03	unsat	0.03	unsat
myciel5.col	5	0.87	unsat	109.25	unsat
myciel6.col	6	22970.5	unsat	∞	*timeout*
myciel7.col	7	∞	*timeout*	∞	*timeout*
queen5_5.col	4	0.01	unsat(`cliquer`)	0.01	unsat
queen6_6.col	6	0.03	unsat	1.68	unsat
queen7_7.col	6	0.01	unsat(`cliquer`)	0.04	unsat
queen8_12.col	11	0.01	unsat(`cliquer`)	9.09	unsat
queen8_8.col	8	15.93	unsat	∞	*timeout*
queen9_9.col	9	2295.93	unsat	∞	*timeout*
queen10_10.col	10	∞	*timeout*	∞	*timeout*
school1.col	13	66.39	unsat(`cliquer`)	592.95	unsat
school1_nsh.col	13	26.6	unsat(`cliquer`)	793.89	unsat
zeroin.i.1.col	48	0.01	unsat(`cliquer`)	∞	*timeout*
zeroin.i.2.col	29	0.01	unsat(`cliquer`)	∞	*timeout*
zeroin.i.3.col	29	0.01	unsat(`cliquer`)	∞	*timeout*

SAT solver, or (b) sat – which means that the SAT solver was called, or (c) `unsat` – which means the SAT solver was called and returned with an unsatisfiable result, or (d) unsat(`BEE`) – meaning that the `BEE` constraint compiler determined during its compilation stages that the instance is unsatisfiable, or (e) unsat(`cliquer`) – meaning that `cliquer` determined that the instance is unsatisfiable, or (f) memory – meaning that the computer ran out of memory while solving this instance, or (g) timeout – meaning that the computation for this instance did not terminate within 24 hours. All times are listed in seconds.

Table 1 illustrates that little improvement is gained from using `cliquer` when solving satisfiable instances with $k = \chi(G)$. In fact, solving times substantially increase for three satisfiable instances when `cliquer` is applied as part of a preprocessing step. Table 2 illustrates that when `cliquer` is incorporated to solve unsatisfiable instances where $k =$

Table 3 Satisfiable Toronto Instances Results.

Instance	k (lit)	with `cliquer` time	with `cliquer` status	without `cliquer` time	without `cliquer` status
hec-s-92	17 (17)	0.08	sat	0.17	sat
sta-f-83	13 (13)	0.03	sat	0.15	sat
yor-f-83	18 (19)	0.46	sat	0.93	sat
ute-s-92	10 (10)	0.16	sat	0.13	sat
ear-f-83	22 (22)	0.3	sat	0.62	sat
tre-s-92	20 (20)	0.51	sat	1.023	sat
lse-f-91	17 (17)	0.13	sat	0.62	sat
kfu-s-93	19 (19)	0.4	sat	0.98	sat
rye-s-93	21 (21)	0.61	sat	1.67	sat
car-f-92	27 (28)	2.08	sat	6.95	sat
uta-s-92	29 (30)	3.7	sat	4.45	sat
car-s-91	27 (27)	1580.47	sat	∞	*timeout*
pur-s-93	31 (36)	6.71	sat	32.03	sat

$\chi(G) - 1$, solving times are considerably improved, often making the difference between being able to determine a result or not.

5 Exam Timetabling: An Application of Graph Coloring

The examination timetabling problem is about scheduling exams to a set of sequential timeslots, in such a way that conflicting exams are not scheduled to the same timeslot. An instance of the examination timetabling problem specifies n, the total number of exams, a set $D \subseteq [n] \times [n]$ of conflicting exams, in such a way that conflicts are symmetric (i.e., $(i, j) \in D \iff (j, i) \in D$), as well as a number of timeslots, m. We seek to find a legal schedule of size m, which is a tuple $S = \langle t_1, \dots, t_n \rangle$ describing a mapping from exams to timeslots such that t_i is the timeslot of exam i. Each timeslot takes a value in the set $\{ 1, \dots, m \}$, and for every pair of conflicting exams i and j — the exams are not scheduled to the same timeslot (i.e., $t_i \neq t_j$).

The examination timetabling problem is reducible to the graph coloring problem by considering the *conflict graph* derived from the problem instance. This is the graph with vertices corresponding to courses and edges induced by the constraint set D such that (i, j) is an edge of the graph if and only if $(i, j) \in D$. If the graph is m-colorable, then the coloring can be taken to be a legal schedule of size m.

In the minimum examination timetabling problem, we seek the minimum m for which there exists a legal schedule. The minimum examination timetabling problem is equivalently reducible to the minimum graph-coloring problem.

We tested our approach on the Toronto timetabling instances, which were introduced by Carter *et al.,* [7]. Selected results for this set of benchmarks are given in Tables 3 and 4, the complete set of results can be found in Appendix A Table 3 lists instances for which satisfiable results were found, illustrating the coloring size that was found using our augmented tool-chain. The table follows the same description as that of Table 1, except that the second column lists the optimal coloring size we found as well as the best known coloring we found in literature [30]. Table 4 lists the corresponding unsatisfiable instances which prove that the colorings found in Table 3 are optimal. The columns of this table follow the same description as those of Table 2.

Table 3 illustrates the improvement gained by using `cliquer` when solving satisfiable instances. The table lists four instances for which the best known colorings are improved, while colorings for the remaining instances match the best known results from literature. Moreover, Table 4 illustrates that when `cliquer` is incorporated to solve unsatisfiable instances, solving

Table 4 Unsatisfiable Toronto Instances Results.

		with `cliquer`		without `cliquer`	
Instance	k (lit)	time	status	time	status
hec-s-92	16	0.01	unsat(`cliquer`)	3736.02	unsat
sta-f-83	12	0.01	unsat(`cliquer`)	25.43	unsat
yor-f-83	17	0.01	unsat(`cliquer`)	∞	*timeout*
ute-s-92	9	0.1	unsat(`cliquer`)	0.65	unsat
ear-f-83	21	0.24	unsat	∞	*timeout*
tre-s-92	19	0.01	unsat(`cliquer`)	∞	*timeout*
lse-f-91	16	0.01	unsat(`cliquer`)	3718.11	unsat
kfu-s-93	18	0.02	unsat(`cliquer`)	∞	*timeout*
rye-s-93	20	0.03	unsat(`cliquer`)	∞	*timeout*
car-f-92	26	50.2	unsat	∞	*timeout*
car-s-91	26	∞	*timeout*	∞	*timeout*
uta-s-92	28	8.18	unsat	∞	*timeout*
pur-s-93	30	4.71	unsat	∞	*timeout*

times are considerably improved, often making the difference between solvable and unsolvable instances. Also notice that, to the best of our knowledge, the chromatic numbers of Toronto instances were not previously reported in the literature [30].

6 Technical Details

The package containing `pl-cliquer` is available for download from the `pl-cliquer` homepage at: `https://www.cs.bgu.ac.il/~frankm/plcliquer/`. The package contains a `README` file, which contains usage and installation instructions, as well as an `examples` directory containing the examples discussed in this paper. The C code for `pl-cliquer` may be found in the `src` directory. Also in the `src` directory are the module files for `pl-cliquer`.

`pl-cliquer` was compiled and tested on Debian Linux and Ubuntu Linux using the 7.x.x branch of SWI-Prolog. Note that `pl-cliquer` should compile and run on any architecture where `cliquer` will compile and run.

7 Conclusions

We have presented, and made available, a Prolog interface to the core components of the `cliquer` clique-finding tool [27]. The principle contribution of this paper is in the utility of the tool which we expect to be widely used. The tool provides a "drop in" clique finding utility, through which Prolog programs which address graph related problems may apply `cliquer` natively, through Prolog, as part of the solving process. Cliques may be generated, subject to programmer selected constraints on size, maximality etc., and may be generated deterministically or non-deterministically. Additionally, we illustrate in Prolog the standard approach to implement a graph coloring solver. The experiments we report on indicate that our tool-chain, augmented with `pl-cliquer`, is on par with results reported in the literature [30, 23, 24].

References

1 Karen I Aardal, Stan PM Van Hoesel, Arie MCA Koster, Carlo Mannino, and Antonio Sassano. Models and solution techniques for frequency assignment problems. *Annals of Operations Research*, 153(1):79–129, 2007.

2 Noga Alon and Ravi B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987. doi:10.1007/BF02579196.

3 Béla Bollobás. Complete subgraphs are elusive. *Journal of Combinatorial Theory, Series B*, 21(1):1 – 7, 1976. doi:10.1016/0095-8956(76)90021-6.

4 Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. *The Maximum Clique Problem*, pages 1–74. Springer US, Boston, MA, 1999. doi:10.1007/978-1-4757-3023-4_1.

5 R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, 004 1941. doi:10.1017/S030500410002168X.

6 M W Carter and D G Johnson. Extended clique initialisation in examination timetabling. *Journal of the Operational Research Society*, 52(5):538–544, 2001. doi:10.1057/palgrave.jors.2601115.

7 Michael W. Carter, Gilbert Laporte, and Sau Yan Lee. Examination timetabling: Algorithmic strategies and applications. *Journal of the Operational Research Society*, 47(3):373–383, 1996. doi:10.1057/jors.1996.37.

8 Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

9 Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Logic programming with satisfiability. *TPLP*, 8(1):121–128, 2008. doi:10.1017/S1471068407003146.

10 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi:10.1145/800157.805047.

11 William H. E. Day and David Sankoff. Computational complexity of inferring phylogenies by compatibility. *Systematic Zoology*, 35(2):224–229, 1986. URL: http://www.jstor.org/stable/2413432.

12 Michael Frank and Michael Codish. Logic programming with graph automorphism: Integrating nauty with Prolog (tool description). *Theory and Practice of Logic Programming*, 16(5-6):688–702, 009 2016. doi:10.1017/S1471068416000223.

13 Ove Frank and David Strauss. Markov graphs. *Journal of the American Statistical Association*, 81(395):832–842, 1986. URL: http://www.jstor.org/stable/2289017.

14 M. R. Garey and D. S. Johnson. " strong " NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, July 1978. doi:10.1145/322077.322090.

15 Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Math.*, 182(1):105–142, 1999. doi:10.1007/BF02392825.

16 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512 – 530, 2001. doi:10.1006/jcss.2001.1774.

17 David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.

18 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

19 Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 350–368. Springer, 2016. doi:10.1007/978-3-319-44953-1_23.

20 Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013. doi:10.3390/a6040618.

21 Ciaran McCreesh and Patrick Prosser. A parallel branch and bound algorithm for the maximum labelled clique problem. *Optimization Letters*, 9(5):949–960, 2015. `doi:10.1007/s11590-014-0837-4`.

22 Nirbhay K Mehta. The application of a graph coloring method to an examination scheduling problem. *Interfaces*, 11(5):57–65, 1981.

23 Isabel Méndez-Díaz and Paula Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, 2006.

24 Isabel Méndez-Díaz and Paula Zabala. A cutting plane algorithm for graph coloring. *Discrete Applied Mathematics*, 156(2):159–179, 2008.

25 Amit Metodi and Michael Codish. Compiling finite domain constraints to SAT with BEE. *TPLP*, 12(4-5):465–483, 2012. `doi:10.1017/S1471068412000130`.

26 Amit Metodi, Michael Codish, and Peter J. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)*, 46:303–341, 2013. `doi:10.1613/jair.3809`.

27 Sampo Niskanen and Patric R. J. Östergård. Cliquer user's guide. Technical Report version 1.0, Communications Laboratory, Helsinki University of Technology, Espoo, Technical Report T48, 2003. URL: `https://users.aalto.fi/~pat/cliquer.html`.

28 Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.*, 120(1-3):197–207, August 2002. `doi:10.1016/S0166-218X(01)00290-6`.

29 Patrick Prosser. Exact algorithms for maximum clique: A computational study. *Algorithms*, 5(4):545–587, 2012. `doi:10.3390/a5040545`.

30 Rong Qu, Edmund K Burke, Barry McCollum, LT Merlot, and Sau Y Lee. A survey of search methodologies and automated system development for examination timetabling. *Journal of scheduling*, 12(1):55–89, 2009.

31 Jean-Charles Régin. *Using Constraint Programming to Solve the Maximum Clique Problem*, pages 634–648. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. `doi:10.1007/978-3-540-45193-8_43`.

32 Ram Samudrala and John Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of Molecular Biology*, 279(1):287 – 302, 1998. `doi:10.1006/jmbi.1998.1689`.

33 D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85, 1967. `doi:10.1093/comjnl/10.1.85`.

34 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

A Toronto Instances

Tables 5 and 6 describe in more detail the results obtained for satisfiable and unsatisfiable instances of the Toronto benchmarks respectively. The first column lists the instance name, the second column lists the best coloring found as well as the previously best known coloring from literature [30], the third column lists the time it took `cliquer` to find a maximum clique in the graph, the fourth column lists the time it took `BEE` to compile the constraints to CNF, The fifth and sixth column detail the size of the CNF in terms of clauses and number of variables, the seventh column lists the time it took the SAT solver to obtain a result, and the last column detail the reason for that result. The values in column seven are the same as the status column of Table 1.

Table 5 Satisfiable Toronto Instances Results.

Instance	k (lit)	`cliquer`	BEE	#clauses	#vars	sat	Status
hec-s-92	17 (17)	0.01	0.05	6601	590	0.02	sat
sta-f-83	13 (13)	0.01	0.01	735	175	0.01	sat
yor-f-83	18 (19)	0.01	0.37	37569	1812	0.08	sat
ute-s-92	10 (10)	0.1	0.04	9100	770	0.02	sat
ear-f-83	22 (22)	0.01	0.22	23159	1639	0.07	sat
tre-s-92	20 (20)	0.01	0.36	55756	2881	0.14	sat
lse-f-91	17 (17)	0.01	0.1	17478	1418	0.02	sat
kfu-s-93	19 (19)	0.02	0.23	48300	2807	0.15	sat
rye-s-93	21 (21)	0.03	0.42	69838	4160	0.16	sat
car-f-92	27 (28)	0.2	1.15	179749	7634	0.73	sat
uta-s-92	29 (30)	0.1	2.1	338007	11490	1.5	sat
car-s-91	27 (27)	0.06	1.93	407155	12672	1578.48	sat
pur-s-93	31 (36)	0.5	2.85	1098306	39613	3.36	sat

Table 6 Unsatisfiable Toronto Instances Results.

Instance	k	`cliquer`	BEE	#clauses	#vars	sat	Status
hec-s-92	16	0.01	–	–	–	–	unsat(cliquer)
sta-f-83	12	0.01	–	–	–	–	unsat(cliquer)
yor-f-83	17	0.01	–	–	–	–	unsat(cliquer)
ute-s-92	9	0.1	–	–	–	–	unsat(cliquer)
ear-f-83	21	0.01	0.2	13841	1210	0.03	unsat
tre-s-92	19	0.01	–	–	–	–	unsat(cliquer)
lse-f-91	16	0.01	–	–	–	–	unsat(cliquer)
kfu-s-93	18	0.02	–	–	–	–	unsat(cliquer)
rye-s-93	20	0.03	–	–	–	–	unsat(cliquer)
car-f-92	26	0.2	1.12	159981	7138	48.88	unsat
car-s-91	26	0.06	1.09	373479	12053	∞	*timeout*
uta-s-92	28	0.1	1.5	310668	10933	6.58	unsat
pur-s-93	30	0.5	2.77	1005456	37849	1.44	unsat

B Dimacs Instances

Tables 7 and 8 describe in more detail the results obtained for satisfiable and unsatisfiable instances of the DIMACS benchmarks respectively. The tables description follows the description of Tables 5 and 6.

Table 7 Satisfiable Dimacs Instances Results.

Instance	k	`cliquer`	`BEE`	#clauses	#vars	sat	Status
anna.col	11	0.01	0.01	–	–	–	sat(BEE)
david.col	11	0.01	0.01	–	–	–	sat(BEE)
DSJC125.1.col	5	0.01	0.02	4017	553	0.03	sat
DSJR500.1.col	12	0.02	0.02	14928	1209	0.03	sat
DSJR500.5.col	122	4661.28	–	–	–	–	*memory*
fpsol2.i.1.col	65	0.02	0.01	–	–	–	sat(BEE)
fpsol2.i.2.col	30	0.02	0.04	9654	894	0.01	sat
fpsol2.i.3.col	30	0.02	0.04	9654	894	0.01	sat
games120.col	9	0.01	0.03	8820	1051	0.01	sat
huck.col	11	0.01	0.01	–	–	–	sat(BEE)
inithx.i.1.col	54	0.05	0.05	17248	1192	0.02	sat
inithx.i.2.col	31	0.05	0.13	103315	3705	0.12	sat
inithx.i.3.col	31	0.05	0.15	103315	3705	0.13	sat
jean.col	10	0.01	0.01	–	–	–	sat(BEE)
le450_15a.col	15	0.02	0.34	127561	6102	1.24	sat
le450_15b.col	15	0.02	0.32	117393	5879	0.42	sat
le450_25a.col	25	0.02	0.33	155049	6290	0.22	sat
le450_25b.col	25	0.02	0.34	190394	7420	0.33	sat
le450_5a.col	5	0.01	0.12	29092	2088	0.05	sat
le450_5b.col	5	0.01	0.11	30023	2117	0.06	sat
le450_5c.col	5	0.01	0.19	42630	1992	0.06	sat
le450_5d.col	5	0.01	0.19	44927	2051	0.07	sat
miles1000.col	42	0.01	0.01	–	–	–	sat(BEE)
miles1500.col	73	0.01	0.01	–	–	–	sat(BEE)
miles250.col	8	0.01	0.01	–	–	–	sat(BEE)
miles500.col	20	0.01	0.01	–	–	–	sat(BEE)
miles750.col	31	0.01	0.01	15	7	0.01	sat
mulsol.i.1.col	49	0.01	0.01	–	–	–	sat(BEE)
mulsol.i.2.col	31	0.01	0.06	41878	1574	0.06	sat
mulsol.i.3.col	31	0.01	0.06	41878	1574	0.06	sat
mulsol.i.4.col	31	0.01	0.06	44293	1642	0.07	sat
mulsol.i.5.col	31	0.01	0.06	43042	1608	0.07	sat
myciel3.col	4	0.01	0.01	83	30	0.01	sat
myciel4.col	5	0.01	0.01	393	91	0.01	sat
myciel5.col	6	0.01	0.01	1570	240	0.01	sat
myciel6.col	7	0.01	0.02	5736	589	0.01	sat
myciel7.col	8	0.01	0.06	19929	1386	0.02	sat
queen5_5.col	5	0.01	0.01	266	31	0.01	sat
queen6_6.col	7	0.01	0.01	1584	138	0.01	sat
queen7_7.col	7	0.01	0.02	2566	192	0.01	sat
queen8_12.col	12	0.01	0.07	19816	872	0.02	sat
queen8_8.col	9	0.01	0.03	7314	380	0.88	sat
queen9_9.col	10	0.01	0.06	12026	547	6.91	sat
queen10_10.col	11	0.01	0.09	21322	856	21637.9	sat
school1.col	14	66.39	0.51	42	16	0.01	sat
school1_nsh.col	14	26.6	0.47	107	34	0.01	sat
zeroin.i.1.col	49	0.01	0.01	–	–	–	sat(BEE)
zeroin.i.2.col	30	0.01	0.01	734	139	0.01	sat
zeroin.i.3.col	30	0.01	0.01	734	139	0.01	sat

Table 8 Unsatisfiable Dimacs Instances Results.

Instance	k	`cliquer`	BEE	#clauses	#vars	sat	Status
anna.col	10	0.01	–	–	–	–	unsat(cliquer)
david.col	10	0.01	–	–	–	–	unsat(cliquer)
DSJC125.1.col	4	0.01	0.02	–	–	–	unsat(BEE)
DSJR500.1.col	11	0.02	–	–	–	–	unsat(cliquer)
DSJR500.5.col	121	4597.34	–	–	–	–	unsat(cliquer)
fpsol2.i.1.col	64	0.02	–	–	–	–	unsat(cliquer)
fpsol2.i.2.col	29	0.02	–	–	–	–	unsat(cliquer)
fpsol2.i.3.col	29	0.02	–	–	–	–	unsat(cliquer)
games120.col	8	0.01	–	–	–	–	unsat(cliquer)
huck.col	10	0.01	–	–	–	–	unsat(cliquer)
inithx.i.1.col	53	0.05	–	–	–	–	unsat(cliquer)
inithx.i.2.col	30	0.05	–	–	–	–	unsat(cliquer)
inithx.i.3.col	30	0.05	–	–	–	–	unsat(cliquer)
jean.col	9	0.01	–	–	–	–	unsat(cliquer)
le450_15a.col	14	0.02	–	–	–	–	unsat(cliquer)
le450_15b.col	14	0.02	–	–	–	–	unsat(cliquer)
le450_25a.col	24	0.02	–	–	–	–	unsat(cliquer)
le450_25b.col	24	0.02	–	–	–	–	unsat(cliquer)
le450_5a.col	4	0.01	–	–	–	–	unsat(cliquer)
le450_5b.col	4	0.01	–	–	–	–	unsat(cliquer)
le450_5c.col	4	0.01	–	–	–	–	unsat(cliquer)
le450_5d.col	4	0.01	–	–	–	–	unsat(cliquer)
miles1000.col	41	0.01	–	–	–	–	unsat(cliquer)
miles1500.col	72	0.01	–	–	–	–	unsat(cliquer)
miles250.col	7	0.01	–	–	–	–	unsat(cliquer)
miles500.col	19	0.01	–	–	–	–	unsat(cliquer)
miles750.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.1.col	48	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.2.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.3.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.4.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.5.col	30	0.01	–	–	–	–	unsat(cliquer)
myciel3.col	3	0.01	0.01	37	15	0.01	unsat
myciel4.col	4	0.01	0.01	267	70	0.01	unsat
myciel5.col	5	0.01	0.01	1170	195	0.85	unsat
myciel6.col	6	0.01	0.02	4548	496	22970.47	unsat
myciel7.col	7	0.01	0.06	16499	1197	∞	*timeout*
queen5_5.col	4	0.01	–	–	–	–	unsat(cliquer)
queen6_6.col	6	0.01	0.01	1070	108	0.01	unsat
queen7_7.col	6	0.01	–	–	–	–	unsat(cliquer)
queen8_12.col	11	0.01	–	–	–	–	unsat(cliquer)
queen8_8.col	8	0.01	0.04	5838	324	15.88	unsat
queen9_9.col	9	0.01	0.06	9859	474	2295.86	unsat
queen10_10.col	10	0.01	0.08	18110	716	∞	*timeout*
school1.col	13	66.39	–	–	–	–	unsat(cliquer)
school1_nsh.col	13	26.6	–	–	–	–	unsat(cliquer)
zeroin.i.1.col	48	0.01	–	–	–	–	unsat(cliquer)
zeroin.i.2.col	29	0.01	–	–	–	–	unsat(cliquer)
zeroin.i.3.col	29	0.01	–	–	–	–	unsat(cliquer)