Another Look at Graph Coloring via Propositional Satisfiability

Allen Van Gelder

Computer Science Dept., SOE-3
University of California
Santa Cruz, CA 95064, USA

Abstract

This paper studies the solution of graph coloring problems by encoding into propositional satisfiability problems. The study covers three kinds of satisfiability solvers, based on postorder reasoning (e.g., grasp, chaff), preorder reasoning (e.g., 2cl, 2clsEq), and back-chaining (modoc). The study evaluates three encodings, one of them believed to be new. Some new symmetry-breaking methods, specific to coloring, are used to reduce the redundancy of solutions. A by-product of this research is an implemented lower-bound technique that has shown improved lower bounds for the chromatic numbers of the long-standing unsolved random graphs known as DSJC125.5 and DSJC125.9. Independent-set analysis shows that the chromatic numbers of DSJC125.5 and DSJC125.9 are at least 18 and 40, respectively, but satisfiability encoding was able to demonstrate only that the chromatic numbers are at least 13 and 38, respectively, within available time and space.

Key words: graph coloring, propositional satisfiability, constraint satisfaction, symmetry breaking, independent set analysis

1 Introduction

We assume the reader is familiar with the *satisfiability problem*, which seeks to determine if any assignment to the propositional variables of a Boolean formula causes it to evaluate to *true*. In recent research, planning problems, hardware and software verification problems and others have been encoded as satisfiability problems. We look at solving graph coloring via satisfiability.

Email address: avg@cs.ucsc.edu (Allen Van Gelder).

Nearly all complete satisfiability solvers are in the DPLL family (for Davis, Putnam, Loveland, and Logemann [5]). They search for a satisfying assignment by fixing variables one by one and backtracking when an assignment forces the formula to be *false*. The procedure is not very effective in its original form, but it has been enhanced with various techniques to reduce the search space.

Reasoning techniques can be broadly classified as *preorder* and *postorder*. Preorder techniques are applied as the search goes forward, and include binary-clause reasoning, equivalent-literal identification, and other efficient reasoning steps whose goal is to show that certain variable bindings cannot lead to a satisfying assignment [3,19,24,14,1]. The most complete preorder reasoning is done by `2cl` [24], but the implementation is too inefficient for currently challenging problems. A more efficient implementation of preorder reasoning is found in `2clsEq` [1], which introduces a special form of resolution named *hyperbinres*.

Postorder techniques are applied when the search is about to backtrack, because a "conflict" has been discovered [20,25,2,18]. Postorder techniques are variously called non-chronological backtracking, conflict-directed back-jumping, clause recording, and learning. These techniques are compared in a recent paper [26]. As of 2001, the leading implementation in this category is generally agreed to be `chaff` [18], although the technique was introduced into high-performance SAT solvers in `grasp` [20] and `relsat` [2] during the 1990s. More recently, new solvers have been implemented using ideas from `chaff`. There are substantial difficulties in combining full preorder reasoning with postorder techniques, and only one prototype has been reported [23]. However, `2clsEq` is able to use postorder reasoning, together with carefully selected preorder reasoning [1].

A somewhat different approach is to use propositional model elimination. Model elimination is distinguished by being a back-chaining theorem prover [15,16,13]. It has been adapted for propositional use and is capable of producing either a proof or a counter-example [22]. The implementation is named `modoc`. The motivation for back-chaining is that the proof search spreads out from the goal and might confine itself to relevant clauses.

Since the Dimacs competition for Graph Coloring in 1993, there has been a lot of progress in satisfiability solvers, so we think it is worth taking another look at this technique for solving Graph Coloring. In particular, we are interested in alternative encodings that might perform better than the standard encoding for large problems. In 1993, neither solvers nor computers were powerful enough to solve large problems. Today the situation is shown to be different.

We are particularly interested in whether there are interactions between encoding techniques and solver styles. It is not the purpose of this paper to evaluate the latest and greatest satisfiability solvers; indeed, the bragging rights will undoubtedly change by the time this paper is published. Therefore, we used solvers that implement important ideas and have proved to be reliable. Experimental results are presented in Section 6.

It is now thought that high-performance satisfiability solvers may have commercial value, and this perception has driven the development of such solvers since the turn of the century. A practical benefit of solving graph coloring by translating to satisfiability is that new satisfiability solvers can be utilized as they become available, with almost no implementation effort. This benefit might be limited to the coloring of "structured" graphs, because the highest-powered satisfiability solvers are geared toward "structured" formulas (as opposed to randomly generated formulas). In fact, the large-scale competition among numerous satisfiability solvers held in conjunction with the 2005 Conference on Theory and Practice of Satisfiability Testing shows almost a complete dichotomy between the best solvers for "industrial" benchmarks and the best for "random" benchmarks [10].

2 Notation

In CNF, the formula is a conjunction of clauses and each clause is a disjunction of literals; each literal is a propositional variable x or its negation $\neg x$. If $q = \neg x$ is a negative literal, $\neg q$ is considered to be its complement, x. We denote a clause as $[q_1, q_2, \ldots, q_k]$ and a formula as $\{C_1, C_2, \ldots, C_m\}$. An empty formula is *true* and $[]$, the empty clause, is *false*. We also define the *tautologous clause* $\top$, which is true under any assignment.

3 Traditional Encoding (*tr*)

We start with an undirected graph G with n vertices and m edges. The question is whether it can be colored with K colors, numbered 0, ..., $K - 1$. Vertex numbers range from 1 through n. The traditional encoding uses K propositional variables for each graph vertex, nK in all. We call this encoding *tr*.

If v is a vertex, then v_c is a propositional variable that means vertex v has color c, $c = 0, \ldots, K - 1$. The propositional formula contains mK negative binary clauses,

$$[\neg u_c, \neg v_c],$$

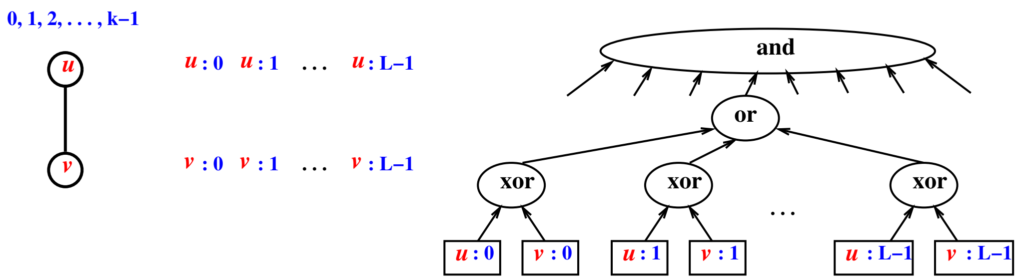

Fig. 1. Circuit outputs 1 if and only if colors of u and v differ in at least one bit.

where (u, v) is an edge. It also contains n positive K-clauses,

$$[v_0, v_1, \ldots, v_{K-1}],$$

where v is a vertex.

4 Circuit-Based Encodings

We start with an undirected graph G with n vertices and m edges. For simplicity assume the number of colors K is a power of 2. Let $L = \lg(K)$ (more generally, $L = \lceil \lg(K) \rceil$). The colors are integers 0, …, $K - 1$, written with L bits. Bit 0 is the low-order bit. If v is a vertex, then v_b is a propositional variable that means that bit b of the color of vertex v is 1. There are nL such variables.

Let (u, v) be an edge in G. The requirement that u and v have different colors can be stated as the requirement that they differ on at least one bit. The circuit in Figure 1 represents this constraint. One point is that there are several ways to encode this circuit into clauses. Another point is that we do not need any "positive" clauses to force the the vertex to have "some" color. If we have "and" together the circuit fragments shown above for each edge, then the graph is K-colorable if and only if some input setting produces an output of 1.

*4.1 Standard Bitwise (*xg*) Encoding*

The usual way to represent the bit-wise constraints is equivalent to replacing "xor" gates with an "and-or" equivalent, and distributing to obtain a product of sums, i.e., conjunctively joined disjunctive clauses. We call this encoding *xg* (see Figure 2).

There are K clauses of $2L$ literals each for each edge. For example, the starred clause requires that u and v cannot both have color 3.

$$
\begin{array}{ll}
[u_0, v_0, u_1, v_1, u_2, v_2, \ldots, u_{L-1}, v_{L-1}] & \\
[\neg u_0, \neg v_0, u_1, v_1, u_2, v_2, \ldots, u_{L-1}, v_{L-1}] & \\
[u_0, v_0, \neg u_1, \neg v_1, u_2, v_2, \ldots, u_{L-1}, v_{L-1}] & \\
[\neg u_0, \neg v_0, \neg u_1, \neg v_1, u_2, v_2, \ldots, u_{L-1}, v_{L-1}] & \qquad (*) \\
\ldots & \\
[\neg u_0, \neg v_0, \neg u_1, \neg v_1, \neg u_2, \neg v_2, \ldots, \neg u_{L-1}, \neg v_{L-1}] &
\end{array}
$$

Fig. 2. Clauses for edge (u, v) in the *xg* encoding.

The main advantage of the *xg* encoding is that it uses about $n \lg K$ variables instead of nK in the traditional encoding. The main disadvantage is that all the clauses are long.

*4.2 Xor Explicit (*xe*) Encoding*

It is interesting that the circuit fragment can be encoded in other ways. In particular, we can introduce variables to represent the outputs of the "xor" gates, a trick that goes back to Tseitin in the 1960's. We call this encoding *xe* because the "xors" are explicitly represented.

Let $x_{u,v,b}$ represent the gate that "xors" bit b of vertices u and v. There are L such variables per edge, mL for the whole graph. We need four 3-clauses to enforce $x_{u,v,b} = \mathrm{xor}(u_b, v_b)$:

$$
\begin{array}{c}
[u_b, v_b, \neg x_{u,v,b}] \\
[\neg u_b, \neg v_b, \neg x_{u,v,b}] \\
[\neg u_b, v_b, x_{u,v,b}] \\
[u_b, \neg v_b, x_{u,v,b}]
\end{array}
$$

Ranging over L bits, this produces $4L$ 3-clauses for each edge. In addition we need one L-clause per edge for the "or" gate:

$$[x_{u,v,0}, x_{u,v,1}, \ldots, x_{u,v,L-1}] \,.$$

4.3 Disallowed Combinations of Bits

Finally, we consider the case that K is not a power of 2. Define $D = 2^L - K$, the number of "disallowed" colors. For each vertex we need some clauses to prohibit that vertex from taking on a disallowed color. We could simply

Table 1
Sizes of CNF formulas for various encodings. The graph has n vertices and m edges. There are K colors; $L = \lceil \lg(K) \rceil$.

Encoding	Variables	Clauses	Literals
tr	nK	mK 2-clauses	$(n+2m)K$
		n K-clauses	
xg	nL	mK $(2L)$-clauses	$2mKL + nL^2/4$
		$nL/2$ $(L/2)$-clauses	
xe	$(n+m)L$	4mL 3-clauses	$13mL + nL^2/4$
		m L-clauses	
		$nL/2$ $(L/2)$-clauses	

prohibit each of the D colors separately, using D K-clauses, but we can do much better when D is large.

An example makes the idea clear, using vertex v. Suppose $K = 11$, so $L = 4$ and $D = 5$. Color 10 in binary is 1010. So if $v_3 = 1$, then v_2 must be 0. Also, if $v_3 = 1$ and $v_1 = 1$, then v_0 must be 0. The clauses are:

$$[\neg v_3, \neg v_2]$$
$$[\neg v_3, \neg v_1, \neg v_0]$$

In general, all the 1-bits to the left of a 0-bit imply the 0-bit, in the binary representation of $K - 1$.

One clause is needed for each 0 in the binary representation of the maximum color using L bits. The scheme works for any maximum color and is also useful for symmetry breaking. For prohibiting colors greater than $K - 1$, the maximum number of clauses needed is $L - 1$ per vertex; this only occurs when the high-order bit of $(K - 1)$ is 1. The maximum number of literals is about $L^2/4$; the exact formula is $\lceil \frac{L+1}{2} \rceil \lfloor \frac{L+1}{2} \rfloor$. This occurs for about $L/2$ ones followed by $L/2$ zeros. We use this value for size comparisons.

4.4 Comparison of Encoding Sizes

Table 1 shows the sizes of formulas based on various encodings. Examination of this table suggests that the *xe* encoding becomes interesting when the number of colors is in the range 16 to 32, or higher.

5 Breaking Symmetry

It is well known that unsatisfiable coloring problems take "forever" because all permutations of colors are tried due to symmetry, unless something is done to prevent this. A standard gimmick is to look for a clique quickly, and force the colors on that clique. We investigated some generalizations of this idea.

If there are K colors, then we can select *any* $K-1$ vertices, say u^1, ..., u^{K-1}, and require u^i to have a color less than i. If the first C of the selected vertices are in a clique, then the colors of u^1, ..., u^C are forced. But more generally, if u^1, ..., u^{K-1} comprise a dense subgraph, then there will be relatively few possible colorings.

Several heuristics for choosing a "hot spot" of this sort can be thought of easily. We implemented and evaluated two heuristics. They both try to "close down" on a clique from above, rather than build up from below. In the process, a subset of the vertices are ordered.

b1 Our first try uses the node of maximum degree, d, as the "start," u^1. Ties are broken by the sum of the neighbors' degrees. Then the neighbors of u^1 are ordered by decreasing degree and placed in sequence following u^1, as u^2, ..., u^{d+1}. Now, for any number of colors, K, the sequence with restricted colors is u^1, ..., u^{K-1}. Of course, if the highest-degree node doesn't *have* $K-2$ neighbors, the graph is trivially K-colorable, even $(K-1)$-colorable.
The computation time is in $O(n^2)$ for a graph with n vertices.

b2 The second heuristic is more complicated, but also more effective in the experiments. For each vertex v the vertices w_i adjacent to v are dynamically ranked as described below. The sequence produced is v (as u^1) followed by the w_i in descending rank order. Now d is the degree of v, but not necessarily the maximum degree of any vertex in the graph.
The initial ranking of w_i is based on how many triangles of the form (w_i, v, w_j) exist. Whichever vertex, say w_{low}, is lowest in rank (for fixed v) is placed last in the v-sequence; i.e., it is tentatively u^{d+1}. Then w_{low} is effectively removed as an adjacency of v for ranking purposes; that is, other w_i's no longer get credit for their triangles involving w_{low}. Repeatedly, rankings are updated and vertices are placed in front of the suffix of the sequence, growing it from u^{d+1} to u^d, to u^{d-1}, and eventually down to u^2. In this way, some prefix of the final sequence is a clique involving v. Whichever sequence produces the largest clique in this way is kept. Ties are broken by favoring larger degrees.
Since d is not necessarily an upper bound on the number of colors required, additional vertices are added arbitrarily after u^{d+1} to make the sequence as long as the maximum degree in the graph; these vertices have never been needed in practice.
The computation time is in $O(n^2 + n\, d_{max}^2)$ for a graph with n vertices and maximum degree d_{max}.

The final sequence, u^1, ..., u^{d+1}, is used in two ways. For encoding K colors it supplies the u^1, ..., u^{K-1} mentioned above, upon which colors are restricted. It is also used for a greedy coloring procedure to provide the sequence in which vertices are greedily assigned colors. This produces an upper bound on the number of colors that require testing.

Thus, during preprocessing, the chromatic number for the graph is bracketed between the size of the clique at the beginning of the final v-sequence and the number of colors used by the greedy procedure. This range is searched by encoding the graph coloring problem into a satisfiability problem for various candidate numbers of colors. The search may be sequential or binary search.

Our implementation, named `solvecolor`, is written in ANSI C, and is available from the author. It encodes the problem into a file containing the CNF formula and forks the satisfiability solver as a separate Unix process. For these experiments, the satisfiability solver was tested on formulas even if `solvecolor` knew the answer due to its preprocessing. This experimental design ensures that

(1) The satisfiability solver verifies the optimal solution, without relying on the encoder's opinions about cliques and greedy colorings. It is worthwhile to emphasize that the encoding *uses* a sequence of vertices, but does not rely on any properties of that sequence, except that no vertex appears more than once.
(2) The satisfiability solver is run on at least one unsatisfiable formula related to each graph. Any satisfiable formula might be solved in linear time by fortunate guesses, but there is no known way to verify unsatisfiability in polynomial time, even with fortunate guesses.

Point (2) had some surprising consequences in the experiments.

6 Experimental Results

So far we have tried `2cl`, `chaff`, `2clsEq`, and `modoc` on the two circuit-based encodings, *xg* and *xe*, and the traditional encoding *tr*. The programs represent three styles of SAT solving: `chaff` is generally accepted as a pioneer for the postorder style, although newer solvers in this genre are outperforming it now; `2cl` represents the preorder style; `2clsEq` is a mix of preorder and postorder; and `modoc` represents the back-chaining model-elimination style.

All CPU times are seconds based on an Intel Xeon, 2 GHz, 4 GB memory, 512K secondary cache. For calibration, `dfmax` takes 16.96 seconds on r500.5.b. Runs that exceeded their allocated CPU time or terminated abnormally for other reasons, are indicated by "+?" in the tables. Runs done on other platforms (Sun UltraSparc 60, 450 MHz, or Intel Pentium4, 2.66 GHz) are normalized into “Xeon units.” For example, 30 UltraSparc seconds = 12 Xeon seconds = 8 Pentium4 seconds.

6.1 Preliminary Results

We report the general findings of this phase without giving detailed tables. We tested without any symmetry-breaking clauses, and with the simple (*b1*) and more complicated (*b2*) symmetry-breaking heuristics described in Section 5. The *b1* symmetry-breaking heuristic produced one to three orders of magnitude speedup compared to no symmetry-breaking. For a typical example, this table shows some times using the *tr* encoding on *myciel5* for 5 colors (uncolorable).

	Symmetry breaking:	none	*b1*	*b2*
CPU secs. for:	`chaff`	1025	15	20
	`2clsEq`	27	2	3

Other programs and encodings showed similar patterns.

The *b2* symmetry-breaking heuristic is much better at finding large cliques than the *b1* heuristic. In some cases this produced another order of magnitude speedup, especially with graphs that have a high chromatic number. Aside from speeding up the test for a specific number of colors, the better lower bound eliminates some of the tests that might otherwise be needed. For these reasons, all tests reported in the tables use the *b2* symmetry-breaking heuristic.

We found that `2cl` was considerably slower than the other programs. It was dropped from later experiments. We also noticed that `chaff` usually ran out of memory before it ran out of time. The version in these experiments is the original version to be distributed, called `Mchaff`.

All of the graphs studied may be found at the COLOR02 web site and many are described in the Workshop proceedings [21]. Most of the graphs are generated from applications. The remaining graphs are the *myciel* series (constructed) or the *DSJC* series (random). Table 2 shows the numbers of vertices and undirected edges in these graphs.

We chose the *myciel* series for our initial tests because it has no cliques to provide an easy way to break symmetries: the clique number remains at 2 while the chromatic number grows. Thus clique-based symmetry-breaking methods are doomed, and we were curious whether our symmetry-breaking methods would be an improvement.

Table 2
Numbers of vertices and undirected edges in graphs tested, as well as chromatic numbers (χ) and sizes of cliques found by heuristic *b2*.

Graph	Vertices	Edges	χ	*b2* Clique
Matrix Partitioning				
abb313GPIA	1557	53356	9	8
ash331GPIA	662	4181	4	3
ash608GPIA	1216	7844	4	3
ash958GPIA	1916	12506	4	3
will199GPIA	701	6772	7	6
Mycielski				
myciel5	47	236	6	2
myciel6	95	755	7	2
myciel7	191	2360	8	2
Classroom Scheduling				
school1_nsh	352	14612	14	14
school1_sh	385	19095	14	14
Random				
DSJC125.1	125	736	5	4
DSJC125.5	125	3891	??	10
DSJC125.9	125	6961	??	33

Graph	Vertices	Edges	χ	*b2* Clique
Register Allocation				
fpsol2.i.1	496	11654	65	65
fpsol2.i.2	451	8691	30	30
fpsol2.i.3	425	8688	30	30
inithx.i.1	864	18707	54	54
inithx.i.2	645	13979	31	31
inithx.i.3	621	13969	31	31
mulsol.i.1	197	3925	49	49
mulsol.i.2	188	3885	31	31
mulsol.i.3	184	3916	31	31
mulsol.i.4	185	3946	31	31
mulsol.i.5	186	3973	31	31
zeroin.i.1	211	4100	49	49
zeroin.i.2	211	3541	30	30
zeroin.i.3	206	3540	30	30

The *myciel* family has instances numbered 2, 3, and 4 that proved to be very easy, and are omitted from the tables. The first mildly challenging instance is *myciel5* for 5 colors (uncolorable), and the *myciel6* for 6 colors is extremely difficult. Apparently, the satisfiable (i.e., colorable) versions are all pretty easy in this family.

The *DSJC* series contains randomly generated graphs, and a small selection of these are included at the request of the referees. Today's leading complete sat solvers are designed to solve problems from industrial applications, which contain a lot of "structure." We anticipate that such solvers will do poorly on encodings of random graphs, because it has been observed that they do poorly on randomly generated CNF formulas.

The programs tested permit many parameters to be varied. For `chaff`, we used parameters recommended by the author to reduce the memory requirements, compared to the default parameters. With the parameters used, `chaff` still used more than 1 GB of memory routinely. For `2clsEq` and `modoc`, we used the default parameters. We observed that `2clsEq` often used well over 1 GB of memory, whereas `modoc` had a much smaller memory footprint, usually under 256 MB.

6.2 Main Results

We now turn to the computationally intensive experiments. For each combination of satisfiability solver and encoding, the same graphs were tested.

Heuristic *b2* was used for symmetry breaking throughout.

The controlling program, *solvecolor*, started with a number of colors one less than the clique size that it found, to be sure the satisfiability solver was exercised on an unsatisfiable formula, and to avoid relying on its own clique computation for correctness. Then, the number of colors was increased by one repeatedly until the satisfiability solver found a solution. The loop was terminated when the the number of colors exceeded that found by the well-known greedy algorithm by two.

It might seem that a more sophisticated procedure to vary the number of colors would be more efficient, especially if the clique size and greedy number are well separated. An alternative was coded that performs binary search. One would think that allowing more colors would make the problem easier.

However, our experience is that increasing the number of colors unnecessarily makes the encoded propositional formulas so much larger that their sheer bulk degrades performance seriously, and risks exhausting memory after putting in a lot of time. So after some preliminary experimentation with strategies, we settled on the simplest, increasing one step at a time.

To avoid a deluge of numbers we show only the times for the chromatic number and one less. (One exception is *myciel7*, for which we show the results for 5 colors; no combination of encoding and solver succeeded in showing any larger uncolorable value. Other exceptions are *DSJC125.5* and *DSJC125.9*.) We should mention that the preprocessing and encoding times were quite minor in comparison to the times taken by the satisfiability solvers. In one extensive run `solvecolor` tested 24 graphs and consumed 0.76 CPU hours on its own, while the satisfiability solvers consumed 11.0 CPU hours. The solver times on the critical numbers of colors are compared for the most challenging graphs in Table 3. The corresponding data for all 27 graphs tested are shown in Tables 4, 5, and 6.

6.3 Discussion

One of the goals of this work was to find out if various encodings fit better or worse with various solver techniques. The complete tables (4, 5, and 6) make it clear that the traditional encoding is usually the best performer, often by orders of magnitude.

However, looking at the most challenging graphs (see Table 3), we see that the *xg* and *xe* encodings work better for `chaff` when the chromatic number is high (*fpsol.i.1* and *inithx.i.1*). This large number of colors is where we expect the benefits of *xg* and/or *xe* encodings to kick in, but we are reaching the limits of

Table 3
Most challenging graphs, critical color numbers. Programs run with *b2* symmetry-breaking heuristic. Time-outs are indicated by "+?".

Graph / Sat Solver	no. of colors	Unsat CPU secs. by Encodings tr	xg	xe	no. of colors	Sat CPU secs. by Encodings tr	xg	xe
abb313GPIA								
`chaff`	8	81647	8	50305+?	9	1772	15412+?	30638+?
`2clseq`	8	18000+?	27000+?	18000+?	9	18000+?	27000+?	18000+?
`modoc`	8	27000+?	5400+?	5400+?	9	27000+?	5400+?	5400+?
fpsol2.i.1								
`chaff`	64	3	12	9	65	18000+?	17	16
`2clseq`	64	5	18000+?	387+?	65	6	4961	55+?
`modoc`	64	2	27000+?	10569+?	65	13	27000+?	27000+?
inithx.i.1								
`chaff`	53	3	18	44	54	7200+?	36	1384
`2clseq`	53	7	8485+?	2+?	54	12	5400+?	2+?
`modoc`	53	2	27000+?	27000+?	54	18	27000+?	27000+?
myciel6								
`chaff`	6	3978+?	3253+?	2247+?	7	0	0	0
`2clseq`	6	2980	18000+?	18000+?	7	0	0	2
`modoc`	6	27000+?	27000+?	27000+?	7	0	1	2
DSJC125.5								
`chaff`	12	1176	823	3768	19	18000+?	6269	18000+?
`2clseq`	12	689	18000+?	18000+?	19	18000+?	18000+?	18000+?
`modoc`	12	18000+?	18000+?	18000+?	19	18000+?	18000+?	18000+?

computer resources. In fact, the quick abnormal terminations of `2clsEq` are apparently due to being unable to allocate its needed data structures. Here is a comparison of the approximate encoding sizes for *inithx.i.1* for 53 colors (uncolorable) or 54 colors (colorable).

	tr	*xg*	*xe*
variables (thousands)	32	5	117
literals (millions)	3.2	11.9	1.7

The numbers for *fpsol.i.1* are equally daunting. Notice the rock/paper/scissors quality of the numbers: there is no clearcut preferred encoding.

To keep things in perspective, remember that `solvecolor`, the encoding program has identified a 54-clique in this graph and is encoding it for only 53 colors in the unsatisfiable case. As described in Section 5, the vertices of the 54-clique have their colors restricted so that u^1 must get color 0, u^2 must get color 0 or 1, but is adjacent to u^1, so it must get color 1, and so on, through u^{53}, which must eventually get color 52. There is no color available for the 54-th vertex of the clique.

We thought that the satisfiability solvers would detect this condition quite trivially. With the *tr* encoding, unsatisfiability can be shown with unit-clause propagation. Indeed, all the solvers succeed handily in this case.

We were quite surprised to discover that the circuit-based encodings conceal this "trivial" refutation. If the solver is smart enough, it only needs to consider the variables associated with 54 vertices, about 6 percent of the total variables. It is evident that `2clsEq` and `modoc` have drifted away from this critical set of variables somehow. Although `chaff` takes longer than it did on the *tr* encoding, it seems to have maintained the focus.

Turning to the satisfiable versions of *fpsol.i.1* and *inithx.i.1*, we observe that `chaff` succeeded only with the circuit-based encodings. For some reason the other programs had no difficulty with the *tr* encoding, while `chaff` had great difficulty. We do not understand why this happened; a conjecture is that the random restarts and the large number of variables caused `chaff` to keep forgetting its progress.

The *xg* encoding was also very effective for `chaff` to show that *abb313GPIA* is not 8-colorable; this is the largest graph tested, with 1557 vertices, although its chromatic number is moderate. The *tr* encoding took almost 24 hours on this problem. If the time limit had been comparable to other runs, this would have showed up as a failure. The *xe* encoding ran out of memory after about 14 hours. Again, we conjecture that random restarts may have slowed the progress of `chaff`, and possibly the smaller number of variables used by the *xg* encoding turned out to reduce or eliminate restarts.

On the other hand, there is no indication that `2clsEq` and `modoc` ever do better with the circuit-based encodings, and often do orders of magnitude worse with them, compared to the traditional encoding. The data are insufficient to draw a firm conclusion, but there is a suggestion that the strategy of `chaff`, rapid searching and limited reasoning, is more compatible with the circuit-based encodings. The *xg* encoding minimizes the number of variables, making a smaller search space than the alternatives.

Both `2clsEq` and `modoc` are based on more extensive reasoning, and the wide clauses of the *xg* encoding are an impediment for them. Reasoning seems to have carried the day for `2clsEq` to prove that *myciel6* cannot be 6-colored. The *tr* encoding produces mostly binary clauses, and `2clsEq` performs extensive binary-clause reasoning. We note that another program, `2cl`, also performs extensive binary-clause reasoning, and also solved this problem, in 3247 CPU seconds.

The best competition on *myciel6* seems to be from `smallk`, a graph coloring decision program designed for testing 3 through 8 colors [4]. Some results are:

graph	colors	`smallk` time	best sat-solver time	sat-solver (encoding)
myciel6	6	2728	2980	`2clsEq` (*tr*)
abb313GPIA	8	86400+?	8	`chaff` (*xg*)

This program, which does many reasoning steps that are also done by `2clsEq` and `2cl`, showed that *myciel6* cannot be 6-colored in 2728 CPU seconds. It also showed that *myciel5* cannot be 5-colored in 0.52 CPU seconds, whereas the best of the satisfiability solvers required about 2 CPU seconds.

Another interesting result was the demonstration that *abb313GPIA* has a 9-coloring. This graph was produced by Hossain and Steihaug, who reported that the question of whether this graph is 9-colorable is open, and that two graph-coloring programs, `dsatur` and `smallk`, were unsuccessful after 3 days [7]. We were able to solve this problem with the traditional encoding, but not with the *xg* or *xe* encodings. Only `chaff` has solved it so far, taking 1772 seconds.

As supplied, `smallk` does not attempt to find a 9-coloring, but the authors provide instructions on how to change some parameters so that it will make the attempt, along with a warning that it is likely to run very slowly. We tried `smallk` for 24 hours to test for an 8-coloring on *abb313GPIA* and it did not terminate. As mentioned above `chaff` proved that *abb313GPIA* is not 8-colorable. We are unsure whether any other program has duplicated this feat.

The numbers for *fpsol.i.1* are equally daunting. Notice the rock/paper/scissors quality of the numbers: there is no clearcut preferred encoding.

Two of the three random graphs studied are open problems and we made only minor progress toward resolving them. Unfortunately, there has been erroneous information about *DSJC125.5* and *DSJC125.9* propagated through the Internet and some conference proceedings. The largest clique size for *DSJC125.5* is 10, as shown by running `dfmax` [21]. Our *b2* heuristic found a clique of this size.

Our sat encodings for 12 colors were found to be unsatisfiable, establishing a lower bound of 13 for the chromatic number of *DSJC125.5*; this appears to be an improvement on previously published lower bounds, but see the next paragraph. However, no valid coloring with fewer than 19 colors was discovered, whereas other researchers have achieved 17 colors [8,12,17].

Using analysis by independent sets, we have shown that the chromatic number of *DSJC125.5* is at least 16. Because this is somewhat off the topic of this paper, it is presented in Appendix A. The satisfiability solvers we used ran out of time or memory or both trying to solve the encodings for 13, 14, and 15 colors, all of which must be unsatisfiable by the result in Appendix A.

For *DSJC125.9*, our *b2* heuristic found a clique of size 33, whereas 34 is optimum, per `dfmax`. The *tr* encoding for 37 colors was shown to be unsatisfiable. A lower bound of 40 is shown via analysis by independent sets in Appendix A. The best valid coloring found was 46 colors. Johnson *et al.* have achieved 44 colors [8]. The sparser graph *DSJC125.1* presented no problems.

The method we used seems to be fairly robust. We have solved a fair range of problems, at least one previously open. We have *myciel6* with 6 colors, a relatively small graph, but one that is structurally very difficult. Then we have *fpsol2.i.1* with 64–65 colors and *inithx.i.1* with 53–54 colors, both large graphs with large coloring numbers. Finally, we have *abb313GPIA*, which dwarfs the rest of the graphs, but needs only 9 colors. However, the success needs to be qualified by the observation that there was no one combination of encoding and satisfiability solver that solved all of the problems within the resource limits.

In terms of the general merit of using satisfiability solvers for graph coloring, we believe there are a few significant accomplishments among these tests. First, one solver succeeds on *myciel6* with 6 colors; that is, it proves the graph is uncolorable. We understand that this is beyond the reach of most graph coloring programs. Second, we were able to prove that *abb313GPIA* is 9-colorable and is not 8-colorable. The question of 9-colorability of this graph was open and the question of 8-uncolorability might also have been open [7].

7 Conclusion

We were able to "piggy-back" on the large amounts of software development effort invested in satisfiability over the last decade. We introduced a few new heuristics to guide the encoding and handed the problem over to several existing satisfiability solvers (as of 2002). The preprocessing and encoding took a minor fraction of the overall running time. The results seem to be comparable to, if not better than, those obtained by full-fledged coloring algorithms.

Extensive empirical testing of "all possible" satisfiability solvers on the encodings is beyond the scope (and computational resources) of this research. Our simple strategy of invoking the solver for successive numbers of colors requires the use of complete solvers. Possible future work would be to test incomplete satisfiability solvers on the encodings and compare results to incomplete graph coloring heuristics, both of which abound.

The method of symmetry breaking described here might be applicable both to algorithms that work on graph coloring directly and to other constraint satisfaction problems. By dynamically pruning from an overly large set of candidates, a sequence is formed once that is useful for varying numbers of colors. On the instances tested, the method often found a maximum clique, but the sequence is useful even if this does not happen.

The circuit-based encodings showed disappointing results, but were not complete failures. More study is needed to see if the *xe* encoding can be improved,

Table 4
`Chaff` runs with *b2* symmetry-breaking heuristic, critical color numbers.

Graph	no. of colors	Unsat CPU secs. by Encodings tr	xg	xe	no. of colors	Sat CPU secs. by Encodings tr	xg	xe
abb313GPIA	8	81647	8	50305+?	9	1772	15412+?	30638+?
ash331GPIA	3	0	0	0	4	0	0	17
ash608GPIA	3	0	0	0	4	0	1	235
ash958GPIA	3	0	0	1	4	0	4	733
fpsol2.i.1	64	3	12	9	65	18000+?	17	16
fpsol2.i.2	29	1	155	2089	30	1	5	11
fpsol2.i.3	29	1	5	2366	30	1	5	7
inithx.i.1	53	3	18	44	54	7200+?	36	1384
inithx.i.2	30	1	8	34	31	1	13	2063
inithx.i.3	30	1	7	46	31	1	13	7053
mulsol.i.1	48	1	4	1	49	1	4	6
mulsol.i.2	30	0	3	7200+?	31	0	3	5
mulsol.i.3	30	0	10	4363+?	31	0	3	5
mulsol.i.4	30	0	16	3651+?	31	0	4	6
mulsol.i.5	30	0	1	5	31	0	3	1
myciel5	5	20	8	6	6	0	0	0
myciel6	6	3978+?	3253+?	2247+?	7	0	0	0
myciel7	5	22	18000+?	50	8	0	0	0
school1_nsh	13	1	54	1702	14	1	1253	156
school1_sh	13	1	4	294	14	1	23	111
will199GPIA	6	0	0	1	7	0	2	158
zeroin.i.1	48	1	4	2	49	1	4	1
zeroin.i.2	29	0	1	1	30	0	1	8
zeroin.i.3	29	0	1	1	30	0	1	9
DSJC125.1	4	0	0	0	5	0	1	1
DSJC125.5	12	1176	823	3768	19	18000+?	6269	18000+?
DSJC125.9	37	6529	18000+?	18000+?	46	18000+?	10133+?	18000+?

and to understand under what circumstances the *xg* or *xe* encodings are likely to outperform the traditional encoding.

Acknowledgments

We thank Prof. Michael Trick for maintaining his graph coloring web site and for stimulating our interest in this problem through workshops.

References

[1] F. Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 7–16, Cincinnati, OH, 2002.

[2] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

Table 5
`2clsEq` runs with *b2* symmetry-breaking heuristic, critical color numbers.

	no. of	Unsat CPU secs. by Encodings			no. of	Sat CPU secs. by Encodings		
Graph	colors	tr	xg	xe	colors	tr	xg	xe
abb313GPIA	8	18000+?	27000+?	18000+?	9	18000+?	27000+?	18000+?
ash331GPIA	3	0	0	1	4	1	1	17
ash608GPIA	3	0	0	1	4	2	2	63
ash958GPIA	3	1	1	2	4	4	5	163
fpsol2.i.1	64	5	18000+?	387+?	65	6	4961	55+?
fpsol2.i.2	29	1	1098	2030+?	30	2	729	3600+?
fpsol2.i.3	29	1	1344	18000+?	30	2	930	14418+?
inithx.i.1	53	7	8485+?	2+?	54	12	5400+?	2+?
inithx.i.2	30	2	8436+?	19+?	31	4	5400+?	17+?
inithx.i.3	30	2	6171+?	15+?	31	4	5400+?	17+?
mulsol.i.1	48	1	1386	27000+?	49	2	943	27000+?
mulsol.i.2	30	1	341	6031+?	31	18000+?	27000+?	5400+?
mulsol.i.3	30	1	405	6407+?	31	18000+?	27000+?	5400+?
mulsol.i.4	30	1	415	20301+?	31	18000+?	358	4557+?
mulsol.i.5	30	1	566	3286+?	31	18000+?	246	5400+?
myciel5	5	3	213	783	6	0	0	0
myciel6	6	2980	18000+?	18000+?	7	0	0	2
myciel7	5	272	18000+?	18000+?	8	0	1	11
school1_nsh	13	2	46	18000+?	14	2	2727	18000+?
school1_sh	13	2	105	325+?	14	2	3154	18000+?
will199GPIA	6	0	1	2	7	3	6	51
zeroin.i.1	48	2	580	27000+?	49	3	241	27000+?
zeroin.i.2	29	1	111	13760+?	30	1	44	3600+?
zeroin.i.3	29	1	94	3595+?	30	1	43	3600+?
DSJC125.1	4	0	0	0	5	0	0	11
DSJC125.5	12	689	18000+?	18000+?	19	18000+?	18000+?	18000+?
DSJC125.9	37	18000+?	18000+?	18000+?	46	26958	18000+?	18000+?

[3] A. Billionnet and A. Sutter. An efficient algorithm for the 3-satisfiability problem. *Operations Research Letters*, 12:29–36, July 1992.

[4] J. Culberson and I. Gent. Frozen development in graph coloring. *Theoretical Computer Science*, 265(1–2):227–264, 2001. (`smallk` source available at URL `http://www.cs.ualberta.ca/~joe/Coloring/`).

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[6] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[7] S. Hossain and T. Steihaug. Graph coloring in the estimation of sparse derivative matrices: Instances and applications. In *DAM Special Issue on Graph Coloring and Its Generalizations (submitted)*, 2004. (preliminary version presented at *International Symposium on Mathematical Programming* 2003).

[8] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May-June 1991.

[9] Abhai Johri and David W. Matula. Probabilistic bounds and heuristic

Table 6
`Modoc` runs with *b2* symmetry-breaking heuristic, critical color numbers.

Graph	no. of colors	Unsat CPU secs. by Encodings tr	xg	xe	no. of colors	Sat CPU secs. by Encodings tr	xg	xe
abb313GPIA	8	27000+?	5400+?	5400+?	9	27000+?	5400+?	5400+?
ash331GPIA	3	0	0	0	4	0	2	27
ash608GPIA	3	0	0	4	4	1	3	979
ash958GPIA	3	0	0	7	4	1	9	1164
fpsol2.i.1	64	2	27000+?	10569+?	65	13	27000+?	27000+?
fpsol2.i.2	29	1	27000+?	27000+?	30	4	27000+?	27000+?
fpsol2.i.3	29	1	27000+?	27000+?	30	4	27000+?	27000+?
inithx.i.1	53	2	27000+?	27000+?	54	18	27000+?	27000+?
inithx.i.2	30	1	27000+?	27000+?	31	8	27000+?	27000+?
inithx.i.3	30	1	27000+?	27000+?	31	7	27000+?	27000+?
mulsol.i.1	48	0	352	11193+?	49	3	12748	27000+?
mulsol.i.2	30	0	27000+?	27000+?	31	2	1256	27000+?
mulsol.i.3	30	0	27000+?	27000+?	31	2	1406	27000+?
mulsol.i.4	30	0	27000+?	27000+?	31	2	1784	27000+?
mulsol.i.5	30	0	27000+?	1190+?	31	2	1670	43
myciel5	5	2	292	690	6	0	0	0
myciel6	6	27000+?	27000+?	27000+?	7	0	1	2
myciel7	5	21	5751	14400+?	8	0	11	4
school1_nsh	13	0	27000+?	27000+?	14	3	27000+?	27000+?
school1_sh	13	1	27000+?	27000+?	14	4	27000+?	27000+?
will199GPIA	6	0	3389	27000+?	7	1	27000+?	27000+?
zeroin.i.1	48	0	1819	14301+?	49	3	23279	5400+?
zeroin.i.2	29	0	337	2338	30	2	4884	287
zeroin.i.3	29	0	570	27000+?	30	2	3430	291
DSJC125.1	4	0	1	308	5	58	811	18000+?
DSJC125.5	12	18000+?	18000+?	18000+?	20	18000+?	18000+?	18000+?
DSJC125.9	37	18000+?	18000+?	18000+?	47	18000+?	18000+?	18000+?

algorithms for coloring large random graphs. Technical Report 82-CSE-06, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas, 75275, June 1982.

[10] Daniel LeBerre *et al.* SAT 2005 Competition First Stage Results. 2005. URL `http://www.satcompetition.org/2005/firststage.html`.

[11] Frank Thomson Leighton. A graph colouring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–503, 1979.

[12] Gary Lewandowski and Anne Condon. Experiments with parallel graph coloring heuristics and applications of graph coloring. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 309–334. American Mathematical Society, 1996.

[13] R. Letz, K. Mayr, and C. Goller. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13(3):297–337, December 1994.

[14] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI*, 2000.

[15] D. W. Loveland. A simplified format for the model elimination theorem-proving procedure. *Journal of the Association for Computing Machinery*, 16(3):349–363, July 1969.

[16] J. Minker and G Zanon. An extension to linear resolution with selection function. *Information Processing Letters*, 14(3):191–194, June 1982.

[17] Craig Morgenstern. Distributed coloration neighborhood search. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 335–357. American Mathematical Society, 1996.

[18] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, June 2001.

[19] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1995.

[20] J. P. Silva and K. A. Sakallah. GRASP–a new search algorithm for satisfiability. In *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 220–227. IEEE Comput. Soc. Press, 1996.

[21] M. A. Trick, editor. *Computational Symposium on Graph Coloring and its Generalizations (COLOR02)*. Ithaca, NY, 2002.
URL `http://mat.gsia.cmu.edu/COLOR02`.

[22] Allen Van Gelder. Autarky pruning in propositional model elimination reduces failure redundancy. *Journal of Automated Reasoning*, 23(2):137–193, 1999.

[23] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002. (Also available at URL `ftp://ftp.cse.ucsc.edu/pub/avg/CBJ/sat-pre-post.ps.gz`.).

[24] Allen Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996. (Also available at URL `ftp://ftp.cse.ucsc.edu/pub/avg/kclose-tr.ps.gz`).

[25] H. Zhang. SATO: An efficient propositional prover. In *14th International Conference on Automated Deduction*, pages 272–275, 1997.

[26] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, Nov. 2001.

A Independent-Set Analysis of *DSJC125.5* and *DSJC125.9*

In this appendix we consider lower bounds for the chromatic numbers of *DSJC125.5* and *DSJC125.9*, two random graphs in the COLOR02 benchmark suite that have been unsolved for more than a decade. The largest cliques in *DSJC125.5* and *DSJC125.9* were found to be sizes 10 and 34, respectively, using `dfmax`. These provide what turn out to be rather weak lower bounds.

We present lower bounds for the chromatic numbers of the graphs *DSJC125.5* and *DSJC125.9*, using independent-set analysis. Johri and Matula used independent-set analysis to obtain probabilistic lower bounds on random graphs, giving tables for probability .999999 [9]. Johnson *et al.* state probabilistic lower bounds for these graphs, attributed to a program written by Thomason, but do not explain what the phrase "with high probability" means for specific values of n [8]. Numerous researchers have proposed heuristic algorithms for graph coloring based on independent-set analysis [11,9,6,8]. The underlying idea is that each color class must be an independent set and each pair of color classes must be disjoint. For conciseness we use the abbreviation "j-IS" for "independent set of size j." The size of the largest independent set of a graph G is conventionally denoted as $\beta(G)$, or just β if the graph is understood from context.

A *partition of* n is defined to be a nonincreasing sequence of positive integers that total to n. Its *length* is the number of elements in this sequence. A *sub-partition* is a prefix of a partition. For any valid k-coloring of a graph with n vertices, the sizes of the color classes can be arranged to form a partition of n with length k [9]. To determine whether a particular graph G with n vertices can be k-colored, it suffices to consider all partitions of n of length k, where the first number in the partition is at most $\beta(G)$. For the remainder of this discussion $n = 125$.

The first step is to find $\beta(\mathit{DSJC125.5})$, for example, using `dfmax` [8] to find a maximum *clique* in the complement graph. Throughout, the computer procedures find cliques in the complement graph whenever independent sets are mentioned. We found $\beta(G) = 10$. This immediately yields a lower bound of $\lceil 125/10 \rceil = 13$ for the chromatic number, which seems to be an improvement on any previously published for *DSJC125.5*. We are able to strengthen this to 16.

The second step is to enumerate all the independent sets of sizes 10, 9, and 8. The procedures are implemented in `matlab` and are available from the author. The logic is similar to `dfmax`, but simplified and modified to record *all* independent sets of the target size. Some independent sets of size 9 are subsets of those of size 10, etc. This step reveals that there are two independent sets of size 10 and 100 of size 9 and 2352 of size 8.

The third step is to enumerate all the disjoint unions of independent sets of sizes 10 and 9. The technique is to repeating try to expand a previously found disjoint union by including one more disjoint independent set. First, it is found that the two 10-IS's are not disjoint. Next enumerate disjoint unions of one 10-IS and one 9-IS to give all 2-colorable subsets of size 19, and enumerate disjoint unions of two 9-IS's to give all 2-colorable subsets of size 18. Extend these to include one more 9-IS in all possible ways to give all 3-colorable subsets of sizes 28 and 27. Extend these to include one more 9-IS in all possible ways, etc.

A subpartition is called *achievable* if there is some disjoint union of independent sets whose sizes correspond to the subpartition. It turns out that the only achievable subpartitions using 10 and 9 are (10, 9, 9, 9, 9) and (9, 9, 9, 9, 9) and prefixes of those subpartitions. In other words there are no disjoint unions of six independent sets of sizes 10 and 9. This whole approach is opportunistic, and becomes feasible because of the few subpartitions that require further analysis. Actually, the programming time was greater than the computing time, overall.

There are only a few partitions of 125 of length 15, once we are restricted to those having only achievable subpartitions involving sizes 9 and 10. They are shown in the following table, with gaps following the parts already known to be achievable.

10	9	9	9	9		8	8	8	8	8	8	8	8	8	7
10	9	9	9		8	8	8	8	8	8	8	8	8	8	8
9	9	9	9	9		8	8	8	8	8	8	8	8	8	8

The fourth and final step is to try to expand the disjoint unions shown to the left of the gaps above to include 8-IS's, shown to the right of the gaps. The following table shows the longest *achievable* subpartitions that extend the above table by using 8-IS's.

10	9	9	9	9	8	8	8		
10	9	9	9	8	8	8	8	8	
9	9	9	9	9	8	8	8	8	8

It follows that there are no partitions of 125 of length 15 all of whose subpartitions are achievable.

Claim 1 *The chromatic number of* DSJC125.5 *is at least 16.*

For *DSJC125.9* the same approach yields a lower bound of 40 for the chromatic number. The largest independent set is size 4. There are nine 4-IS's, but at most five can be combined with disjoint unions. Thus at least 105 vertices need to be partitioned into color classes of size at most 3; 35 disjoint 3-IS's is the best that might be achieved.

Claim 2 *The chromatic number of* DSJC125.9 *is at least 40.*