*Electronic Colloquium on Computational Complexity* - Reports Series 1995 - available via:
ECCC
TR95-033
FTP: ftp.eccc.uni-trier.de:/pub/eccc/
WWW: http://www.eccc.uni-trier.de/eccc/
Email: ftpmail@ftp.eccc.uni-trier.de with subject 'help eccc'

# 3-Coloring in time $O(1.3446^n)$: a no-MIS algorithm

Richard Beigel
Dept. of Computer Science
Yale University
New Haven, CT 06520

David Eppstein*
Dept. Inf. and Computer Science
University of California
Irvine, CA 92717

**Abstract**

We consider worst case time bounds for NP-complete problems including 3-SAT, 3-coloring, 3-edge-coloring, and 3-list-coloring. Our algorithms are based on a common generalization of these problems, called *symbol-system satisfiability* or, briefly, *SSS* [1]. 3-SAT is equivalent to (2,3)-SSS while the other problems above are special cases of (3,2)-SSS; there is also a natural duality transformation from $(a, b)$-SSS to $(b, a)$-SSS. We give a fast algorithm for (3,2)-SSS and use it to improve the time bounds for solving the other problems listed above.

## 1 Introduction

There are many known NP-complete problems including such important graph theoretic problems as coloring and independent sets. Unless P=NP, we know that no polynomial time algorithm for these problems can exist, but that does not obviate the need to solve them as efficiently as possible, indeed the fact that these problems are hard makes efficient algorithms for them especially important.

We are interested in this paper in worst case analysis of algorithms for two basic NP-complete problems: 3-coloring and 3-satisfiability. We will also discuss some other related problems including 3-edge-coloring and 3-list-coloring.

Our algorithms for these problems are based on the following simple idea: to find a solution to a 3-coloring problem, it is not necessary to choose a color for each vertex (giving something like $O(3^n)$ time). Instead, it suffices to only partially solve the problem by restricting each vertex to two of the three colors. We can then test whether the partial solution can be extended to a complete coloring in polynomial time (e.g. as a 2-SAT instance). This idea applied naively already gives a simple $O(1.5^n)$ time randomized algorithm; we improve this by taking advantage of local structure (if we choose a color for one vertex, this restricts the colors of several neighbors at once). It seems likely that our idea of only searching for a partial solution can be applied to many other combinatorial search problems.

*Work supported in part by NSF grant CCR-9258355 and by matching funds from Xerox Corp.

If we perform local reductions as above in a 3-coloring problem, we eventually reach a situation in which some uncolored vertices are surrounded by partially colored neighbors, and we run out of good local configurations to use. To avoid this problem, we translate our 3-coloring problem to one that also generalizes the other problems listed above: *symbol-system satisfiability*. In an $(a, b)$-SSS problem, we are given a collection of $n$ vertices, each of which can be given one of $a$ different colors. However certain color combinations are disallowed: we also have input a collection of $m$ *constraints*, each of which forbids one coloring of some $b$-tuple of vertices. Thus 3-satisfiability is exactly (2,3)-SSS, and 3-coloring is a special case of (3,2)-SSS in which the constraints disallow adjacent vertices from having the same color.

As we show, $(a, b)$-SSS problems can be transformed in certain interesting and useful ways: in particular, one can transform $(a, b)$-SSS into $(b, a)$-SSS and vice versa, one can transform $(a, b)$-SSS into $(\max(a, b), 2)$-SSS, and in any $(a, 2)$-SSS problem one can eliminate vertices for which only two colors are allowed, reducing the problem to a smaller one of the same form. Because of this ability to eliminate partially colored vertices immediately rather than saving them for a later 2-SAT instance, we can solve a (3,2)-SSS instance without running out of good local configurations. We also use some of these transformations to apply our solution to 3-SAT.

### 1.1 New Results

We show the following:

- A (3,2)-SSS problem with $n$ vertices can be solved in worst case time $O(1.3803^n)$, independent of the number of constraints. (This complexity had been conjectured by Russell Impagliazzo, in a personal communication, for the simpler problem of 3-colorability.) We also give a very simple randomized algorithm for solving this problem in expected time $O(n^{O(1)}2^{n/2}) \approx O(1.4142^n)$.
- 3-coloring in a graph of $n$ vertices can be solved in time $O(1.3446^n)$, independent of the number of edges in the graph.
- 3-list-coloring (graph coloring given a list at each vertex of three possible colors chosen from some larger set) can be solved in time $O(1.3803^n)$, independent of the number of edges.
- 3-edge-coloring in an $n$-vertex graph can be solved in time $O(1.5039^n)$, again independent of the number of edges.
- 3-satisfiability of a formula with $m$ clauses can be solved in time $O(1.3803^m)$, independent of the number of variables in the formula.

Throughout, $n$ denotes the number of vertices in a graph, and $m$ denotes the number of constraints in an SSS problem (clauses in a SAT problem).

### 1.2 Related Work

We have only found a small number of papers on worst case analysis of algorithms for NP-hard problems. Several authors have described algorithms for maximum independent sets [2, 7, 8, 11]; the best of these is Robson's [7], which takes time $O(1.2108^n)$.

For three-coloring, we know of two relevant references. Lawler [5] is primarily concerned with the general chromatic number, but he also gives the following very simple algorithm for 3-coloring: for each maximal independent set, test whether the complement is bipartite. The maximal independent sets can be listed with polynomial delay [4], and there are at most $3^{n/3}$ such sets [6], so this algorithm takes time $O(1.4422^n)$. Schiermeyer [10] gives a complicated algorithm for solving 3-colorability in time $O(1.415^n)$, based on the following idea: if there is one vertex $v$ of degree $n-1$ then the graph is 3-colorable iff $G-v$ is bipartite, and the problem is easily solved. Otherwise, Schiermeyer performs certain reductions involving maximal independent sets that attempt to increase the degree of $G$ while partitioning the problem into subproblems, at least one of which will remain solvable. Our $O(1.3446^n)$ bound significantly improves both of these results, and our title was chosen to reflect the fact that unlike both of these papers we do not use maximal independent sets.

For 3-satisfiability, the only worst case analysis we are aware of is again by Schiermeyer [9], who claims an $O(1.579^n)$ bound where now $n$ denotes the number of variables. As our bound depends not on variables but on clauses, our results are incomparable, but our results are an improvement over Schiermeyer's for problems in which $m < 1.417n$.

We were unable to locate prior work on worst case edge coloring. Since any 3-edge-chromatic graph has degree at most three, one can transform the problem to 3-vertex-covering at the expense of increasing $n$ by a factor of $3/2$. If we applied our vertex coloring algorithm we would then get time $O(1.559^n)$ which is somewhat improved by the bound stated above.

## 2 Symbol System Satisfiability

We now describe a common generalization of satisfiability and graph coloring, which we call *symbol-system satisfiability* (SSS). We are given a collection of $n$ vertices, each of which has a list of possible colors allowed. We are also given a collection of $m$ *constraints*, consisting of a tuple of vertices and a color for each vertex. A constraint is *satisfied* by a vertex coloring if not every vertex in the tuple is colored in the way specified by the constraint. We would like to choose one color from the allowed list of each vertex, in a way not conflicting with any constraints.

For instance, 3-satisfiability can easily be expressed in this form. We form $n$ vertices, one per variable of the satisfiability problem, each of which may be colored either *true* ($T$) or *false* ($F$). For each clause like $(x_1 \vee x_2 \vee \bar{x}_3)$, we make a constraint $((v_1, F), (v_2, F), (v_3, T))$. Then a clause is satisfied iff one of its terms is true.

In the $(a, b)$-*symbol-system satisfiability* ($(a, b)$-SSS) problem, we restrict our attention to instances in which each vertex has at most $a$ possible colors and each constraint involves at most $b$ vertices. The SSS problem constructed above from a 3-SAT instance is then a $(2, 3)$-SSS

problem, and in fact 3-SAT is easily seen to be equivalent to $(2,3)$-SSS. On the other hand, graph $k$-colorability can be translated directly to a form of $(k,2)$-SSS: we keep the original vertices of the graph and their possible colors, and add $k$ constraints per edge to enforce the condition that two adjacent vertices can not share any of the $k$ possible colors.

Of course since these problems are all NP-complete the theory of NP-completeness provides translations from one problem to the other, but the translations above are particularly direct.

As we now show, $(a,b)$-SSS problems can be transformed in certain interesting and useful ways. We first describe a form of duality that transforms $(a,b)$-SSS instances into $(b,a)$-SSS instances, exchanging constraints for vertices and vice versa.

**Lemma 1.** *If we are given an $(a,b)$-SSS instance, we can find an equivalent $(b,a)$-SSS instance in which each constraint of the $(a,b)$-SSS instance corresponds to a single vertex of the transformed problem, and each constraint of the transformed problem corresponds to a single vertex of the original problem.*

**Proof:** A coloring of the original vertices solves the original $(a,b)$-SSS problem iff for each constraint there is a pair $(V,C)$ in the constraint that does not appear in the coloring. In our transformed problem, we choose one vertex per original constraint, with the colors for the vertex being the pairs $(V,C)$ in the original problem. Choosing such a pair in a coloring of the transformed problem is interpreted as ruling out $C$ as a possible color for $V$ in the original problem. We then add constraints to our transformed problem to ensure that for each $V$ there remains at least one color that is not ruled out; each constraint corresponds to an $a$-tuple of pairs $(V,C)$ in which the $C$'s cover all the possible colors of $V$. □

This duality may be easier to understand with a small example. Suppose we start with the 3-sat instance $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_4)$. Then we make a $(3,2)$-SSS instance with three vertices $v_i$, one for each 3-SAT clause. Each vertex has three possible colors: $(1,2,3)$ for $v_i$, $(1,3,4)$ for $v_2$, and $(1,2,4)$ for $v_3$. The requirement that one of the values $T$ or $F$ be available to $x_1$ corresponds to the constraints $((v_1,1),(v_2,1))$ and $((v_2,1),(v_3,1))$; we similarly get constraints $((v_1,2),(v_3,2))$, $((v_1,3),(v_2,3))$, and $((v_2,4),(v_3,4))$. One possible coloring of this $(3,2)$-SSS instance would be to color $v_1$ 1, $v_2$ 3, and $v_3$ 4; this would give satisfying assignments in which $x_1$ and $x_3$ are $T$, $x_4$ is $F$, and $x_2$ can be either $T$ or $F$.

We can similarly translate an $(a,a)$-SSS instance into an $(a,2)$-SSS instance in which each vertex corresponds to either a constraint or a variable, and each constraint forces the variable colorings to match up with the dual constraint colorings; we omit the details as we do not use this construction in our algorithms.

Our next result on SSS problems gives a way of eliminating vertices for which few colors are available.

**Lemma 2.** *Let $v$ be a vertex in an $(a,2)$-SSS instance, such that only two of the $a$ colors are allowed at $v$. Then we can find an equivalent $(a,2)$-SSS instance with one fewer vertex.*

**Proof:** Let the two colors allowed at $v$ be $R$ and $G$. Add constraints $((u,A),(w,B))$ for every $(u,A)$ conflicting with $(v,R)$ and every $(w,B)$ conflicting with $(v,G)$. This constraint does not

reduce the space of solutions to the original problem since if both $(u, A)$ and $(w, B)$ were present in a coloring there would be no possible color left for $v$. Conversely if all such constraints are satisfied, one of the two colors for $v$ must be available. Therefore we can now find a smaller equivalent problem by removing $v$. □

When we apply the lemma above, the number of constraints can increase, but there exist only $(an)^2$ distinct constraints, which in our applications will be a small polynomial.

# 3 SSS Algorithms

We first demonstrate the usefulness of Lemma 2 by describing a very simple randomized algorithm for solving $(3,2)$-SSS problems in expected time $O(2^{n/2})$.

**Lemma 3.** *If we are given a $(3,2)$-SSS instance $I$, then in random polynomial time we can find an instance $I'$ with two fewer variables, such that if $I'$ is solvable then so is $I$, and if $I$ is solvable then with probability at least 1/2 so is $I'$.*

**Proof:** If no constraint exists, we can solve the problem immediately. Otherwise choose some constraint $((v, A), (w, B))$. Rename the colors if necessary so that both $v$ and $w$ have available the same three colors $R$, $G$, and $B$, and so that $A = B = R$. Restrict the colorings of $v$ and $w$ to two colors each in one of four ways, chosen uniformly at random from the four possible such restrictions in which exactly one of $v$ and $w$ is restricted to colors $G$ and $B$. It can be verified by examination of cases that this restriction has probability $1/2$ of not disallowing some given solution to the original problem. Now apply Lemma 2 and eliminate both $v$ and $w$ from the problem. □

**Corollary 1.** *In expected time $O(n^{O(1)}2^{n/2})$ we can find a solution to a $(3,2)$-SSS problem if one exists.*

**Proof:** We perform the reduction above $n/2$ times, taking polynomial time and giving probability at least $2^{-n/2}$ of finding a correct solution. If we repeat this method until a solution is found, the expected number of repetitions is $2^{n/2}$. □

We now outline a more complicated method of solving this problem deterministically with the somewhat better time bound of $O(1.38028^n)$.

The basic idea is to find some pair $(v, R)$ that is involved in constraints with at least three other vertices. If we choose to give $v$ color $R$, we eliminate one possible color from each of those other vertices and therefore by Lemma 2 we get a problem with four fewer vertices. On the other hand, if $v$ is colored $G$ or $B$, we can use Lemma 2 to eliminate $v$ itself. Since these two cases exhaust the possibilities, the total time can be analyzed by a recurrence $T(n) \leq T(n-1) + T(n-4)$, which solves to $O(1.38028^n)$.

There are a number of other cases, in which no vertex $v$ above exists; we provide a sketch of the full case analysis in an appendix, and state here without proof our result.

**Theorem 1.** *We can solve any $(3,2)$-SSS problem in time $O(1.38028^n)$.*

This immediately gives algorithms for some more well known problems, some of which we improve later. Of these, the least familiar is likely to be *list $k$-coloring*: given at each vertex of a graph a list of $k$ colors chosen from some larger set, find a coloring of the whole graph in which each vertex color is chosen from the corresponding list [3].

**Corollary 2.** *We can solve the 3-coloring and 3-list coloring problems in time $O(1.38028^n)$, and the 3-edge-coloring and 3-satisfiability problems in time $O(1.38028^m)$.*

# 4 Vertex 3-Coloring

Simply by translating a 3-coloring problem into a (3,2)-SSS problem, as described above, we can test 3-colorability in time $O(1.38028^n)$. We now describe some methods to reduce this time bound even further.

The basic idea is as follows: we find a small set of vertices $S \subset V(G)$ with a large set $N$ of neighbors, and choose one of the $3^{|S|}$ colorings for all vertices in $S$. For each such coloring, we translate the remaining problem to a (3,2)-SSS instance. The vertices in $S$ are already colored and need not be included in the (3,2)-SSS instance. The vertices in $N$ now have a colored neighbor, so for each such vertex at most two possible colors remain; therefore we can eliminate them from the (3,2)-SSS instance using Lemma 2. The remaining instance has $k = |V(G) - S - N|$ vertices, and can be solved in time $O(1.38028^k)$ by Theorem 1. The total time is thus $O(3^{|S|}1.38028^k)$. By choosing $S$ appropriately we can make this quantity smaller than $O(1.38028^n)$.

We can assume without loss of generality that all vertices in $G$ have degree three or more, since smaller degree vertices can be removed without changing 3-colorability.

As a first cut at our algorithm, choose $X$ to be any set of vertices, no two adjacent or sharing a neighbor, and maximal with this property. Let $Y$ be the set of neighbors of $X$. We define a rooted forest $F$ covering $G$ as follows: let the roots of $F$ be the vertices in $X$, let each vertex in $Y$ be connected to its unique neighbor in $X$, and let each remaining vertex $v$ in $G$ be connected to some neighbor of $v$ in $Y$. (Such a neighbor must exist or $v$ could have been added to $X$). We let the set $S$ of vertices to be colored consist of all of $X$, together with each vertex in $Y$ having three or more children in $F$.

We classify the subtrees of $F$ rooted at vertices in $Y$ as follows. If a vertex $v$ in $Y$ has no children, we call the subtree rooted at $v$ a *club*. If $v$ has one child, we call its subtree a *stick*. If it has two children, we call its subtree a *fork*. And if it has three or more children, we call its subtree a *broom*.

We can now compute the total time of our algorithm by multiplying together a factor of 3 for each vertex in $S$ (that is, the roots of the trees of $F$ and of broom subtrees) and a factor of 1.38028 for each leaf in a stick or fork. We define the *cost* of a vertex in a tree $T$ to be the product $p$ of such factors involving vertices of $T$, spread evenly among the vertices—if $T$ contains $k$ vertices the cost is $p^{1/k}$. The total time of the algorithm will then be $O(c^n)$ where

$c$ is the maximum cost of any vertex. It is not hard to show that this maximum is achieved in trees consisting of three forks, for which the cost is $(3(1.38028)^6)^{1/10} \approx 1.35423$. Therefore we can three-color any graph in time $O(1.35423^n)$.

We can improve this somewhat with some more work.

**Lemma 4.** *Let $T$ be a tree with three subtrees, at least two of which are forks or brooms. Then $T$ can be colored with cost per vertex $(6(1.38028)^2 + 3(1.38028)^3)^{1/10} \approx 1.3446$.*

**Proof:** If all three subtrees are brooms, we can choose a color for each subtree root and obtain cost $3^{2/13} \approx 1.1841$. Otherwise, let $x$, $y$, and $z$ be the roots of the three subtrees, in order by subtree size; then $x$ and $y$ are forks or brooms and $z$ is not a broom. Let $k$ denote the number of children of $z$. We test each of the nine possible colorings of $x$ and $y$. In six of the cases, $x$ and $y$ are different, forcing the root of $T$ to have one particular color. In these cases the only remaining vertices after translation to a (3,2)-SSS problem and application of Lemma 2 will be the children of $z$, so in each such case $T$ accumulates a further cost of $(1.38028)^k$. In the three cases in which $x$ and $y$ are colored the same, we must also take an additional factor of 1.38028 for $z$ itself. Therefore the total cost per vertex is at most $((6 + 3(1.38028))(1.38028)^k)^{1/(8+k)}$, or less if $x$ or $y$ is a broom. This expression is maximized when $k = 2$. □

The worst case among trees with four or more children occurs when $T$ has exactly four forks; in this case the cost is $(3(1.38028^8))^{1/13} \approx 1.3267$. If $T$ has three children, and is not covered by Lemma 4, then it has at most four grandchildren, and the cost is at most $(3(1.38028)^4)^{1/8} \approx 1.3478$; this then is the worst case remaining.

The next idea is to improve this case by reducing the number of grandchildren.

**Lemma 5.** *We can in polynomial time cover $G$ by a forest like the one described above, but in which no tree has exactly one fork, two sticks, and no other subtrees.*

The proof involves a complicated case analysis, which is relegated to an appendix in this extended abstract. With this result, the only three-child trees not covered by Lemma 4 are those with three or fewer grandchildren, for which the cost is $(3(1.38028)^3)^{1/7} \approx 1.3432$. Therefore the worst case now occurs in Lemma 4, and we can 3-color any graph in time $O(1.3446^n)$. We state this as a theorem:

**Theorem 2.** *We can solve the 3-coloring problem in time $O(1.3446^n)$.*

## 4.1 Small improvements

Two small improvements are possible. First, it is possible to amortize any configuration that contains 3 forks against configurations with 3 sticks. this results in an average cost per node approximately 1.3438 and a 3-coloring algorithm that runs in time approximately $1.3438^n$. The configurations with 3 sticks can be further amortized against even cheaper configurations, resulting in an $O(c^n)$ algorithm where $c < 1.3438$.

Second, suppose $G$ contains some degree three vertex $v$ such that all vertices within distance 2 also have degree three; $v$ will be colorable iff two of its neighbors are the same color. If this neighborhood of $v$ contains no cycles, then by identifying two neighbors in each of three ways, and removing $v$ and the three neighbors in the third subtree, we can reduce the original problem to three subproblems at a cost per vertex of $3^{1/4}$. If there is a cycle of degree three vertices, we can in other ways reduce the problem size at low cost. So without loss of generality every vertex of $G$ is within distance two of a vertex with degree at least four. We can use this fact to improve the time bound by making as many trees as possible in $F$ have at least four children. We can make a constant fraction of $G$ be covered by such trees, after which we apply the previous methods to color the rest of $G$. The cost per vertex in a constant fraction of $G$ will then be smaller, reducing the time to $O(c'^n)$ for some $c' < c < 1.3438$.

We have omitted the details of both small improvements from this extended abstract, because the improvements are small compared to the complexity of the analysis.

# 5 Edge Coloring

Suppose we want to 3-edge-color an $n$-vertex graph G. Note that we can assume $G$ has maximum degree 3, that there are no vertices of degree 1, and that there are no two adjacent degree-2 vertices.

If we just 3-color the edge graph $L(G)$, we would get time $O(1.3446^m)$, and since there are at least 2/3 as many vertices as edges, this could be reformulated in terms of $n$ as $O(1.559^n)$.

However if there are only $2m/3$ vertices in $G$, then all vertices in $L(G)$ must have degree four making our coloring algorithm faster. We take advantage of this situation as follows. Instead we form trees in L(G) by selecting a maximal set $A$ of edges that do not have any common neighbors. Let $B$ be the neighbors of $A$, and set the parent of each edge in $B$ to be its unique neighbor in $A$. Let $C$ denote the remaining edges; each edge in $C$ will be adjacent to from one to four edges in $B$.

Our algorithm will in general color edges in $A$, which restricts the colors of edges in $B$ leaving only $C$ in the eventual (3,2)-SSS instance. We now analyze this algorithm by spreading to the vertices of $G$ the costs coming from the choices of colors in $A$ and the size of the (3,2)-SSS instance. We will also modify somewhat the coloring technique in some cases.

We assign each edge in $C$ a cost of 1.38028, which will be spread evenly among its neighbors in $B$. Each tree then accumulates a charge of $3(1.38028)^c$ for some $c$ that counts the number of nearby edges in $C$. We spread this charge to the vertices of $G$, as in our 3-coloring algorithm, so we have to know how many vertices are in each tree. Endpoints of $A$ can be assigned a unique tree. Endpoints of $B$ may sometimes be shared between two or three trees, in which case we count them as contributing a fraction of a vertex per tree. There remain also some endpoints of $C$; if such a point exists we think of it as contributing 1/3 of a vertex to the trees containing each adjacent edge in $C$. (If there are only two adjacent edges, the costs only improve; if an edge in $C$ is shared among two trees we contribute 1/6 of a vertex to each.)

Thus each tree in $L(G)$ may have a number of different types of subtrees, which can be

clubs, sticks, or forks as before (brooms can not occur however). The subtrees can be further partitioned into further cases depending on which edges of $C$ occur in whole or fractionally, and how large a fraction of a vertex is counted in that subtree. By a case analysis omitted here, one can show that the most costly type of fork is one where the tree gets the whole 1.38028 charge from each grandchild, and can only spread that charge onto 1/3 of a vertex per grandchild. The most costly type of stick is one where an endpoint of the edge in $B$ is shared between two trees, so that the two trees each get $1/2+1/6$ of a vertex and split the 1.38028 charge. The worst type of club is one where the endpoint of the club is split three ways, giving only 1/3 of a vertex per tree. Forks are worse than sticks which are worse than clubs for the coloring technique above (in which we color just the root of the tree).

Also note that we can assume that every tree with two forks has four children, since otherwise we could reroot the tree at one of the forks and increase the number of four-child trees. We can also use the same analysis as in our vertex coloring algorithm to avoid the case of three children one of which is a fork and two of which are clubs.

We color trees with four forks by coloring two children on opposite sides of the root edge. If they are different, the coloring of the root and all children is forced, else the other two children have a choice of two colors left. So the cost is $(6+3(1.38028)^4)^{3/26} \approx 1.3856$ per vertex. We use a similar idea to color cases with three forks and another child. The worst remaining four-child tree is two forks and two sticks; we just color the root, with cost $(3(1.38028)^5)^{3/20} \approx 1.50156$.

In the three-child cases, there are two possible bad cases: if there are three sticks, the cost is $(3(1.38028)^{3/2})^{1/4} \approx 1.485$. And if there is one child of each type, the cost is $(3(1.38028)^{5/2})^{3/14} \approx 1.5039$. As a consequence of this analysis we have the following result.

**Theorem 3.** *We can solve the 3-edge-coloring problem in time $O(1.5039^n)$.*

We omit further details in this extended abstract.

## Acknowledgments

The first author is grateful to Russell Impagliazzo and Richard Lipton for bringing this problem to his attention. Both authors are grateful to Laszlo Lovasz for helpful discussions.

## References

[1] R. Floyd and R. Beigel. *The Language of Machines: An Introduction to Computability and Formal Languages.* Computer Science Press, New York, 1993.

[2] T. Jian. An $O(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Trans. Comput.* C-35 (1986) 847–851.

[3] T. R. Jensen and B. Toft. Graph Coloring Problems. Wiley, 1995, pp. 18–21.

[4] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Inf. Proc. Lett.* 27 (1988) 119–123.

[5] E. L. Lawler. A note on the complexity of the chromatic number problem. *Inf. Proc. Lett.* 5 (1976) 66–67.

[6] J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.* 3 (1965) 23–28.

[7] J. M. Robson. Algorithms for maximum independent sets. *J. Algorithms* 7 (1986) 425–440.

[8] M. Shindo and E. Tomita. A simple algorithm for finding a maximum clique and its worst-case time complexity. *Systems and Computers in Japan* 21:3 (1990) 1–13.

[9] I. Schiermeyer. Solving 3-satisfiability in less than $1.579^n$ steps. *6th Worksh. Computer Science Logic*, Springer-Verlag (1993) 379–394.

[10] I. Schiermeyer. Deciding 3-colourability in less than $O(1.415^n)$ steps. *19th Int. Worksh. Graph-Theoretic Concepts in Computer Science*, Springer-Verlag (1994) 177–182.

[11] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM J. Comput.* 6 (1977) 537–546.

## Appendix I: Case Analysis for (3,2)-SSS

We describe the cases necessary to show that a $(3,2)$-SSS problem can be solved in time $O(1.38028^n)$. We do this by showing that in any instance there is a configuration that can be replaced by several smaller configurations, such that the recurrence describing the total number of subproblems after repeatedly performing such replacements solves to $O(1.38028^n)$ or less.

Two other ideas help in this case. First, if $((v,R),(w,R))$ is the only constraint involving $(v,R)$, then you can add four constraints of the form $((v,[GB]),(w,[GB]))$ (i.e. always choose a coloring in which one of $v$ and $w$ is $R$). Second, if you have constraints $((v,R),(w,G))$, $((v,R),(w,B))$, and $((w,R),(x,R))$, you can add a transitive constraint $((v,R),(x,R))$ without changing the set of possible colorings.

1. If some pair $(v,R)$ has constraints involving three or more other vertices, then as described in the main part of the paper choose either to give $v$ color $R$ or one of the other two colors $B$ and $G$. The second choice removes just v from the problem, and the other removes v and its neighbors, so the time is bounded by the recurrence $T(n) \leq T(n-1)+T(n-4)$, which solves to $O(1.38028^n)$. (1.38028 is the positive real root of the polynomial $c^4 - c^3 - 1 = 0$; it can be represented by radicals but the formula describing it is messy.)

2. If some pair $(v,R)$ has only one constraint (to say $(w,R)$), we have subcases:

   (a) If $((v,R),(w,R))$ is not the only constraint involving $(w,R)$, and the additional constraint with $(w,R)$ involves a third vertex $x$, then give $w$ either color $R$ or give it one of $B$ or $G$. If you color $w$ $R$ you remove three vertices from the graph, and if you give it one of the other two colors it's then safe to color $v$ $R$ and remove two vertices from the graph. So you get recurrence $T(n) \leq T(n-2)+T(n-3)$, which solves to roughly $O(1.32^n)$.

   (b) If you have two constraints $((v,[RB]),(w,R))$ then it is never necessary to color $v$ $B$, so eliminate $v$ at no cost.

   (c) If $((v,R),(w,R))$ is also the only constraint involving $(w,R)$, then $v$ and $w$ will be colorable iff one of the four choices $([vw],[BG])$ is available. We can safely add to our problem the four constraints $(([vw],[BG]),([vw],[BG]))$. If one of those four pairs (say $(v,B)$) has only one additional constraint (to say $(x,B)$) then choose either to color $x$ $B$ (forcing $v$ to be $[GR]$ and reducing the problem by two vertices) or color $x$ $[GR]$ (leaving $v$ and $w$ always colorable and reducing the problem by three vertices). The recurrence is the same as the first subcase above.

   (d) On the other hand, if $(v,B)$ has two additional constraints (to the same vertex $(x,R)$ and $(x,G)$ else you would have a previous case) then you can safely add a transitive constraint $(v,B),(z,C)$ where $(z,C)$ is any pair that conflicts with $(x,B)$. So to avoid having a previous case, $(x,B)$ must be constrained by one or both of $(w,[BG])$, say $(w,B)$. Then if $(w,B)$ has two additional constraints one is back to $(x,B)$ and the other must be to e.g. $(x,R)$. One also gets a similar picture connecting $(v,G)$

and $(w, G)$ by two edges each either to $x$ or to some other vertex $y$. If everything is connected to $x$, so that $x$ forms an articulation point, you can remove $v$ and $w$ (and possibly eliminate some bad colors in $x$). If everything is connected through $x$ and $y$, you can again remove $v$ and $w$ (and possibly add a constraint between $x$ and $y$). Either way the problem is reduced at no cost.

3. If some pair $(v, R)$ has three or more constraints, but only to two other vertices (it couldn't be to just one or we could eliminate $(v, R)$) we have subcases.

   (a) $(v, R)$ has exactly three constraints, say to $(w, B)$, $(w, G)$ and $(x, R)$. Then unless a previous case applies, $(w, R)$ has two or more constraints. To avoid additional transitive constraints on $(v, R)$, the constraints for $(w, R)$ must be among $(v, G)$, $(v, B)$, and $(x, R)$. Then make a binary choice of coloring $x$ $R$ and $v$ one of $[BG]$ or coloring $x$ one of $[BG]$, $v$ $R$, and $w$ $R$. The cost is again $O(1.32^n)$.

   (b) $(v, R)$ has four constraints say to $([wx], [BG])$. Then to avoid additional transitive constraints the constraints for $(w, R)$ must be among $([wx], [GB])$ and symmetrically. So we can always color all three vertices $R$, and remove three vertices from the problem for no cost.

4. If no previous case applies, every pair $(v, R)$ has exactly two constraints. Therefore the graph of pairs (vertex,color) and constraints is a disjoint union of cycles. If there is some sequence $A, B, C, D, E$ of pairs each belonging to distinct vertices, then we can always find a coloring with $C$, with $B$, or with $A$ and $D$. The first two choices eliminate three vertices and the last eliminates all five (perhaps more if the other neighbor of $A$ is distinct from $E$). Therefore we get the recurrence $T(n) \leq 2T(n-3) + T(n-5)$ which gives cost $O(1.36396^n)$.

5. If there is some square $A, B, C, D$ of constrained pairs in different vertices, choose a coloring with $A$ and $C$ or with $B$ and $D$. Either choice eliminates four vertices so the cost is $2^{n/4}$.

6. If there is a sequence $A, B, A$ of vertices in a cycle of constrained pairs, and if the transitive rule can't add more constraints, the constraints involving the other three colors of these vertices must form a sequence $B, A, B$. We can always find a coloring which uses the middle choice in the two sequences, so both vertices can be removed from the problem for no cost.

7. If there is a sequence $A, B, C, A$ of vertices in a cycle, and the color of the two occurrences of $A$ is different, then we can always find a coloring in which either $B$ or $C$ is used, giving cost $2^{n/3}$.

8. If there is a sequence A-B-C-D-A not forming a square, we have subcases.

   (a) The cycle eventually breaks out of the square, forming a sequence $A, B, C, D, E$ analyzed above.

(b) The cycle eventually touches the square vertices out of order, forming another of the cases above.

(c) The entire cycle repeats two or more times around a square. If there are three repetitions all four vertices can be covered and removed from the problem for free. If there are two repetitions, any set of three can be covered, so the overall problem is solvable iff the remaining color is free on one of the four vertices. We can replace the four vertices by a configuration of two vertices with the same property, reducing the problem size by two for no cost.

9. If all remaining cycles of color pairs are triangles, then form a three-regular bipartite graph connecting vertices to triangles. Such a graph always has a matching, which can be used to solve the remaining problem. So such a problem is always solvable for no cost.

## Appendix II: Exterminating the Bad Trees

In this appendix we show that, as part of our 3-coloring algorithm, we can cover the graph $G$ with a rooted forest in which all leaves are at depth one or two, and in which one bad case is eliminated: there are no trees having one fork, two sticks, and no other subtrees.

To prove this, we start with the cover described in the section on 3-coloring, and perform various local manipulations in an attempt to increase the potential

$$(-c_1 n) + c_2 A + c_3 B + c_4 C + c_5 D$$

where $A$ is the number of trees with four children, $B$ is the number of other trees, $C$ is the number of forks and brooms, and $D$ is the number of trees that are not the bad case. We choose $c_1 > c_2 > c_3 > c_4 > c_5$ so that one of $A$, $B$, $C$, or $D$ can decrease by a small amount as long as a more important quantity increases. The potential above can only increase $O(n)$ times so the number of rearrangements will be $O(n)$ and our algorithm for finding a good forest will take polynomial time.

Given adjacent vertices $x$ and $y$, each in different trees, say that $x$ *can steal* $y$ if $y$ is a grandchild in its tree, or if it is a child in a three-child tree in which there is another fork or broom. In the second case we can remove y by rerooting its tree at the fork or broom, possibly leaving some leftover vertices which can be reattached elsewhere or creating another tree. This will increase the potential if $A$ or $B$ increases in the process. We might also do this to increase $C$ but we have to be more careful since stealing $y$ may also reduce $C$.

Say that $x$ *can move to* $y$ if $y$ is the root of its tree, or if it's a second level vertex such that adding $x$ to the tree would create a new fork or would not create a new bad configuration.

**Lemma 6.** *For any adjacent pair $x$ and $y$ in different trees, either $x$ can steal $y$ or $x$ can move to $y$.*

Now suppose we have a bad tree $T$ consisting of a root $v$, three children $p$, $q$, and $r$, two grandchildren $u$ and $w$ in the fork rooted at $p$, and two grandchildren $x$ and $y$ in the sticks rooted at $q$ and $r$ respectively.

Since $G$ has degree at least three, both $x$ and $y$ have two additional adjacencies somewhere in $G$, possibly in $T$ or possibly in a different tree. We examine a number of cases.

1. If $x$ or $y$ can move to a third vertex, do so. This increases $C$ or $D$ and does not increase any of the other terms in the potential.

2. If $x$ (or $y$) has both neighbors in other trees than $T$, and can't move to either, steal both neighbors and make trees rooted at $x$ and $p$, increasing $B$. This could only fail if both vertices are in different forks or brooms of the same tree, and at least one is the root of its subtree, but then $x$ could instead move there and we would have the previous case.

   At this point we know that both $x$ and $y$ have a neighbor in $T$ other than their parents, and if either of them also has a neighbor in another tree it's one that can be stolen.

3. If $x$ is adjacent to $y$ and one of them can steal a third vertex, steal that vertex and make two trees, one rooted at $x$ or $y$ and the other at $p$.

4. If $x$ (or $y$) is adjacent to $p$, $q$, $r$, or $v$, we can rearrange the edges in $T$ to form a better case.

5. If $x$ and $y$ are both adjacent to $u$ (or $w$), reroot $T$ at $p$ creating a case with two forks and a club.

   At this point we know that (without loss of generality) we have edges $xu$ and $yw$, so $T$ has the shape of two pentagons sharing a common edge $vp$. There is still a third adjacency unaccounted for at each of six vertices. We know that the only possible internal edge involving $x$ or $y$ is $xy$.

6. If $x$ and $y$ can steal different vertices, do so and make two trees rooted at $x$ and $y$. The only problem occurs if both stolen vertices are in two forks in the same tree, but then $x$ or $y$ could have moved to one of them.

7. If $x$ and $y$ can steal the same vertex $z$, and $z$ is a child in its tree, move both $x$ and $y$ to $z$ and make another fork.

8. If $x$ and $y$ can steal the same vertex $z$, $z$ is a grandchild in its tree $T'$, and $T'$ contains another fork or broom, replace $T'$ with two trees rooted at the fork and at $z$.

9. If $x$ and $y$ can steal the same vertex $z$, which is a grandchild in the only fork or broom of its tree, recenter the tree at the root of the fork or broom making a configuration with two forks. This increases $C$ unless the leftover vertices get together to make an extra tree and increase $B$.

10. If $x$ and $y$ can steal the same vertex $z$, which is a grandchild in a stick, and the tree contains no forks or brooms, then stealing $z$ does not destroy any existing forks, and unless a previous case occurs $x$ and $y$ have no other adjacencies except possibly with each other. We then consider subcases:

(a) If $xy$ is an edge, rearrange $T$ and steal $z$ to form a four-child tree.

(b) If $u$, $w$, $q$, or $r$ can be moved to another tree without creating a new bad configuration, do so and reroot $T$ if necessary to form a tree with three sticks.

(c) If $u$ (or $w$, $q$, or $r$) can steal a vertex, do so and create trees rooted at $u$ and $y$ (stealing also $z$ to include in the tree rooted at $y$). The only problem occurs if $u$ wants to steal $z$— in this case create two trees rooted at $z$ and $v$.

In the remaining cases, the third adjacency at $u$, $w$, $q$, and $r$ must either be internal or connect to a vertex to which $u$ or the others can move in a way that would create a fork in a new bad tree $T'$. Some of the cases below involve performing such a move and stealing $z$; note that $T'$ cannot already contain a fork so even if $z$ is part of $T'$, both the move and the steal can happen without interfering with each other.

(d) If $u$ and $q$ (or $w$ and $r$) can both be moved to create new forks (possibly in bad configurations), do so and reroot $T$ at $y$ (stealing $z$ to include in the rerooted $T$).

(e) If $u$ and $w$ (or $q$ and $r$) can both be moved to create new forks, do so and reroot $T$ at $v$.

(f) If edge $ur$ (or $qw$) exists, reroot $T$ at $u$ and form two forks rooted at $x$ and $r$.

(g) If edges $uq$ and $wr$ exist, rearrange $T$ to form two forks rooted at $q$ and $r$.

(h) If edges $uq$ and $uw$ exist (or similar cases) rearrange $T$ to form a tree with four children.

(i) If the only internal edges are $uw$ and $qr$, but $u$ (or $w$, $q$, or $r$) can move to form a fork, move it and reroot at $y$ (leaving a bad case but increasing the number of forks).

(j) If the only internal edges are $uw$ and $qr$, and the only connections between $T$ and $G$ are at $p$, $v$, and $z$, then $T$ can be colored no matter what colors we choose at these three vertices so we can delete six vertices from $G$ at no cost.

This eliminates all possibilities where $x$ or $y$ has an outside adjacency. Therefore in the remaining cases $x$ and $y$ are adjacent to each other and have total degree three. We can think of $T$ as forming a configuration in the form of a tetrahedron with four subdivided edges, having $x$, $y$, $p$, and $v$ as its four corners.

Then we could also form an equivalent bad configuration rooted at x or y, so after going through the same case analysis again we can assume that $p$ and $v$ also have no outside adjacencies. Therefore all connections to $G - T$ are through $u$, $w$, $q$, or $r$.

11. If we have an edge $ur$ (or $wq$), reroot $T$ at $u$ forming two forks and a club.

12. If we have two edges $uw$ and $uq$ (or three similar cases) reroot $T$ at $u$ forming four children.

13. If we have two edges $uw$ and $qr$ (or $uq$ and $wr$) reroot at $x$ giving two forks and a club.

At this point we know that there is at most one internal edge, so some vertex in $T$, say $u$, has no extra internal edges. The four vertices $x$, $y$, $p$, and $v$ will be colorable iff we choose

colors for $u$, $w$, $q$, and $r$ in such a way that (a) all four colors are the same, (b) three colors are the same, or (c) all three colors get used.

14. If $u$ has at most one edge to $G - T$, then no matter how a coloring of $G$ forces $w$, $q$, and $r$ to be colored, we have two free colors for $u$ and can extend the coloring of the rest of $G$ to all of $T$. Therefore we can remove $u$ and the four internal vertices of $T$ from $G$ at no cost.

15. If $u$ can move to an outside vertex, move it and reduce the number of bad configurations.

16. In the remaining case, both of $u$'s outside adjacencies are either in trees with only three children, or are grandchildren in their trees. Steal both vertices (possibly decreasing $B$) and form a new tree rooted at $u$ with four children (increasing $A$).

This exhausts the cases and shows that, if there is a bad tree, the potential can always be increased. Since it can not increase indefinitely, we eventually find a forest with no bad trees.