DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Exploring the k-colorable Landscape with Iterated Greedy

JOSEPH C. CULBERSON AND FENG LUO

Apr. 9, 1995

ABSTRACT. Many heuristic algorithms have been proposed for graph coloring. The simplest is perhaps the greedy algorithm. Many variations have been proposed for this algorithm at various levels of sophistication, but it is generally assumed that the coloring will occur in a single attempt. We note that if a new permutation of the vertices is chosen which respects the independent sets of a previous coloring, then applying the greedy algorithm will result in a new coloring in which the number of colors used does not increase, yet may decrease. We introduce several heuristics for generating new permutations that are fast when implemented and effective in reducing the coloring number.

The resulting Iterated Greedy algorithm(IG) can obtain colorings in the range 100 to 103 on graphs in $\mathcal{G}_{1000,\frac{1}{2}}$. More interestingly, it can optimally color k-colorable graphs with k up to 60 and $n = 1000$.

We couple this algorithm with several other coloring algorithms, including a modified TABU search, and one that tries to find large independent sets using a pruned backtrack. With these combined algorithms we find 86 and 87 colorings for $\mathcal{G}_{1000,\frac{1}{2}}$, but find paradoxical results on other classes.

Finally, we explore the areas of difficulty in probabilistic graph space under several parameterizations. We check our system on k-colorable graphs in $\mathcal{G}_{300,p,k}$ for $0.05 \le p \le 0.95$ and $2 \le k \le 105$. We perform many experiments on $\mathcal{G}_{1000,\frac{1}{2},k}$. We find a narrow ridge where the algorithms fail to find the specified coloring, but easy success everywhere else. We test our algorithm on graphs that follow the ridge boundary for $p = \frac{1}{2}$ for n up to 10000 vertices.

1991 *Mathematics Subject Classification.* Primary 90C35, 68R10; Secondary 94C15, 05C15, 05C80.

Ths work was supported in part by Natural Sciences and Engineering Research Council Grant No. OGP8053.

This paper is in final form and no version of it will be submitted for publication elsewhere.

Part I. Introduction

1. Overview

Graph coloring is perhaps one of the most notorious of the NP-complete problems. It has many applications in scheduling and timetabling. The best deterministic exact algorithms are hopelessly exponential; the best known polynomial time approximation algorithms cannot guarantee a coloring assignment within less than $O(n^{0.4})$ [**2**] of the optimal coloring and are very complex. It is known that there is an $\varepsilon > 0$ such that approximate graph coloring with ratio n^{ε} is NP-hard[**16**]. This indicates dismal prospects in the worst case.

Numerous algorithms have been developed for approximate coloring. The simple greedy algorithm starts with some permutation of the vertices and as each vertex is considered in turn, it is assigned the minimum color that does not cause a conflict. For random graphs in which each vertex pair is assigned an edge with probability 1/2, it is known that asymptotically this will on average use about twice as many colors as required[**8**]. In the worst case it will use $\Theta(n)$ times as many colors as needed[**10**]. Many heuristics have been proposed for ordering the vertices for coloring by the greedy algorithm. But usually the greedy algorithm is seen as a one shot approach to coloring.

In this paper we will apply the greedy algorithm iteratively, basing each successive ordering on the preceding coloring, using various heuristics. One feature of this approach is that the successive coloring numbers will be non-increasing.

Our heuristics can be used to generate new permutations starting with any valid coloring. Thus we can use Brélaz's DSATUR [**4**] algorithm to form an initial coloring, and then proceed with our heuristics. Alternatively, we can begin with an algorithm like that of Bollobás and Thomason [**3**]. This algorithm recursively selects a a large independent set from the uncolored vertices, and applies the next color to it. We also explore the use of TABU search [**9**], developing a method to combine it with iterated greedy by alternating between them.

Finally we explore the coloring landscape with this system, as we vary a known coloring number, and the density of the graph. We find classes of k-colorable graphs that even our full system cannot color well. On the other hand, most k-colorable graphs are easily colored, even for values of k well outside the ranges suggested in previous analyses [**23, 13**].

In the next section we describe various classes of graphs we use, since we will make reference to them throughout the paper. Following that, we describe in some detail each of the algorithms we use, relating our implementations to similar methods in the literature. In the final sections, we present experimental results and analyses for various combinations of these algorithms.

2. Graph Classes

Random graphs $\mathcal{G}_{n,p}$ have n vertices, and for each pair of vertices an edge is assigned with probability p. In one sense, the class of random graphs is easy, since the greedy algorithm is known to assign approximately twice the chromatic number when coloring graphs with $p = 1/2$ [**19, 8**]. On the other hand, they are difficult to color optimally. And in general for a graph G in this class we do not know what the chromatic number $\chi(G)$ is.

We define several types of k-colorable random graphs. These graphs are always k-colorable, although the chromatic number may occasionally be less than k in the graphs produced in one of these classes.

Equi-partite graphs, $\mathcal{G}_{n,p,k}$ have n vertices, partitioned into k as nearly equal sized sets as possible. Edges are assigned with probability p provided that the vertices in question are in distinct elements of the partition. No edges are assigned between vertices within any element of the partition. Kučera[**14**] showed that on a subset of this class of graphs the greedy algorithm would almost surely (in the asymptotic sense) take $O(n^{\epsilon}k)$ colors.

In *k-colorable graphs* each vertex is assigned at random one of the colors $0 \ldots k-1$. The vertices of the same color form the partition elements, and edges are then assigned as they are in the equi-partite class. This class is also used in the literature [**6, 11, 20, 23**]. In [**11, 20**] the edge assignment probability is adjusted so that the expected number of edges is $p\binom{n}{2}$. In addition, Johnson *et al*[**11**] introduce a clique of size k to ensure that $\chi(G) = k$. Since in our experiments we vary p and k to explore the boundary, making such an adjustment would only shift the boundary.

Highly variable k-colorable graphs, $\mathcal{G}_{n,p,k,\epsilon}$, with $0 \le \epsilon < k$, allow us to increase the variability in the size of the partition elements. For each vertex in turn, choose γ at random with $0 \le \gamma \le \epsilon$. Then assign the integer i chosen at random from the range $[\gamma, k-1]$. (We use colors $0 \ldots k-1$). The special case $\epsilon = 0$ generates a k-colorable graph.

In a variation called *smoothly variable graphs*, the colors assigned to the vertices are computed using $kx(\alpha x + 1 - \alpha)$. Here $0 \le \alpha \le 1$ is a variability parameter and x is a random number from the interval $[0, 1)$. Again for $\alpha = 0$ we have the class of k-colorable graphs.

The last class of k-colorable graphs are the *flat graphs*. We present an algorithmic description of this class. Since IG seems to find it easier to get good approximations to k when the variance of such measures as degree sequence, or partition element size, is increased, the flat graphs are designed to minimize such variations. First, as in equi-partite graphs, the set of vertices is partitioned into k as nearly as possible equal sized sets. The expected number of edges is computed as $\sum_{i<j} p|s_i||s_j|$. These edges are distributed as equitably as possible between the pairs of sets $(s_i, s_j), 1 \le i < j \le k$. For each pair of sets (s_i, s_j), for the subgraph induced by $s_i \cup s_j$, we compute an upper bound on the degree of a vertex $\bar{d}_{ij}(s_k), k = i, j$. $\bar{d}_{ij}(s_k) = \lceil \frac{e_{ij}}{|s_k|} \rceil$ is the ceiling of the expected degree of a vertex in s_k within the subgraph. The edges allocated to the pair (s_i, s_j) are assigned at random to $s_i \times s_j$ subject to the constraint that the maximum degree of any vertex in set s_k within the induced subgraph does not exceed $\bar{d}(s_k) + f$, where f is a non-negative flatness parameter. Note that there is no explicit constraint on the total degree of a vertex, since the contribution to the total degree of a vertex from edges assigned to different sets is not constrained.

We also tested the algorithm on the DIMACS bench mark suite of graphs. The flat graphs we contributed to this suite have $f = 0$.

Part II. Algorithm Descriptions

3. The Iterated Greedy Algorithm

The Iterated Greedy (IG) graph coloring algorithm uses the greedy, or simple sequential, graph coloring algorithm repeatedly to obtain ever better colorings. On each iteration, the permutation presented to the greedy algorithm is generated so that the vertices of the independent sets identified in the previous coloring are adjacent in the permutation. It is trivial to prove that this ensures that the new coloring will use no more colors than the previous coloring.

Assuming that a coloring has been generated using the greedy coloring, it follows that for colors $c_2 > c_1$ every vertex with color c_2 is joined by an edge to some vertex of color c_1. However, the converse is not true. If the vertices are rearranged so that the vertices of color c_2 precede those of color c_1 and the greedy algorithm is applied, it is possible that some of the vertices from the c_1 set will receive the same color as those in c_2. An accumulation of such partition changes may result in a reduced coloring.

There are several heuristics that might be used to rearrange the vertices given a coloring of a graph[**5**]. Assume that the current coloring uses k colors. The vertices of each color set are placed as a group.

(i) **Reverse:** Place the vertices of the ith color set as group $k - i + 1$.
(ii) **Random:** Place the groups in random order.
(iii) **Largest First:** Place the groups in order of decreasing size.
(iv) **Smallest First:** Place the smallest groups first.
(v) **Increasing Total Degree:** Place the groups in increasing order by total degree of the group.
(vi) **Decreasing Total Degree:** Place the groups in decreasing order by total degree of the group.

On each iteration, the algorithm randomly selects one of these heuristics to generate the next permutation. The proportion of times that a heuristic is to be used is input as a set of parameters to the program. We have found that a ratio of 50:50:30 for **Largest:Reverse:Random** is usually as good as any setting over a wide range of graphs from various classes. Occasionally we use 70:50:30, which usually has similar performance, but offers some improvement on certain graph classes. **Reverse** tends to give the largest amount of mixing between the color sets, while **Largest First** emulates the well known paradigm of attempting to find large independent sets[**20, 3**]. In the experiments in this paper, when the sets are ordered by size, sets of equal size are ordered by decreasing color class, thus combining the features of **Reverse** and **Largest First**. Without **Random**, the process tends to stagnate on cyclic patterns from which no further progress is made. The other heuristics are based on degree sequence. Tests in [**5**] indicated that these are less effective on the graph classes we consider.

This algorithm falls into the class of neighborhood search algorithms[**20**] which include TABU[**9**] and simulated annealing[**11**]. However, the neighborhoods are different here, in that instead of forming a new partition by the direct manipulation of vertices between partition elements, we instead create new permutations which are then recolored by the greedy algorithm. We have no direct need of a measure of improvement or quality of result, except as a termination condition, since the algorithm is guaranteed never to create a worse coloring, and the neighbor chosen is selected randomly by the reordering heuristic.

Termination of the algorithm can occur when either a specified coloring objective has been achieved, or when a specified number of iterations have occurred without any improvement. Part of the measure of progress used is $\sum_{1\le i\le n} c_i$, where c_i is the color assigned to vertex i. We call this the *coloring sum*. It is easy to show [**5**] that there are graphs for which the minimum coloring sum is arbitrarily less than the coloring sum of any optimal coloring. Similarly, there are graphs for which the chromatic number is arbitrarily less than the number of colors used for any minimum coloring sum. Nevertheless, experience shows that this measure does reflect progress towards improved colorings for most random and k-colorable graphs. To assure that improved colorings are reflected in the measure of progress, a term nc is added to the coloring sum, where n is the number of vertices and c is the number of colors currently used.

3.1. Greedy Implementation and Timing. The greedy algorithm is the heart of the iterated greedy approach. To improve the running time of the program the following algorithm was used.

As each vertex is assigned a color, it is added to a linked list of vertices having the same color. The lists are implemented in a pair of arrays, with a third array used for the starting point of the lists. When a vertex is chosen to be colored, it is tested against each color class previously assigned to see if it can receive that color.

To determine if a vertex can join a color class, it is tested against each vertex in that class until either a conflict is found, or all are tested. Thus, assuming the graph is from $\mathcal{G}_{n,p}$, and that we are on the first iteration of IG, the expected number of vertices tested per class will be (less than) $1/p$. Thus, for fixed p, the expected running time will be $O(nk)$, where k is the expected number of colors assigned. Since for the greedy algorithm it is known that the expected value of k for a graph in $\mathcal{G}_{n,p}$ is $O(n/\log n)$, the expected running time is $O(n^2/\log n)$. For subsequent iterations, the independence assumptions no longer hold, but the result is a reasonable estimate.

For k-colorable graphs, if the number of colors used is small, the time may be dominated by the tests which succeed; that is by the tests which allow the vertex to join some color class. Assuming for example that the graph is k-colorable equi-partite, a pass achieving a k-coloring would take time in $O(nk + n^2/k)$. The first term is the time for rejection of vertices from other color classes as in the above analysis. The second term accounts for the time to build the color classes. The second term reflects the fact that there are approximately n/k vertices in a class, and that each of these has to be tested against all vertices previously inserted into the class. Thus, each of the k classes requires $O((n/k)^2)$ time to build. If k is a constant, then the time becomes $O(n^2)$.

For larger dense graphs, this approach yields significant savings over more naive $O(n^2)$ implementations.

In the following table we list representative times per iteration for equi-partite graphs with $p = 1/2$ and various values of k and n for our bench mark machine. The times are computed by dividing the coloring time actually used from the first iteration to the last iteration by the number of iterations. This "average" is slightly bogus because the coloring time per iteration will decrease as the coloring number decreases for the ranges of colors we consider. However, most of the iterations occur while the search is stuck in the plateau region described in section 10. We wish merely to give an indication of how the coloring times change, not to yield a basis for scientific extrapolation. Other graph classes, in particular other probabilities, might yield somewhat different results. The time for the initial greedy coloring is not included, since it includes some extra initialization time not used in the subsequent iterations of greedy.

n	k	Iterations	Avg. Time
1000	50	1843	0.046
2000	65	1174	0.145
5000	75	420	0.738
10000	136	4570	3.198

After the initial pass, the 5000 vertex graph had a coloring of 477. After 10 iterations this was reduced to 446, and the average time per iteration in this interval was 0.867 seconds. In the last 5 iterations, the coloring was reduced from 100 to 75 with an average time per iteration of 0.34 seconds.

After the initial greedy pass, the 10000 vertex graph had a coloring of 873. It required 71 iterations to achieve an 800 coloring, and the average time per iteration during this interval was 3.793 seconds. In the last five iterations the coloring dropped from 204 to 136, and the average time for these five iterations was 0.98 seconds. The last iteration dropped the coloring from 137 to 136 and required 0.87 seconds.

4. Brélaz's DSATUR Algorithm

Brélaz's DSATUR algorithm[4] is similar to the greedy algorithm, but differs in the heuristics used to select the next vertex for coloring. The order is determined dynamically as the algorithm progresses.

Initially, the vertices are sorted by decreasing degree. The *saturation* of a vertex is the number of distinct colors that have been assigned to adjacent vertices; that is, it is the number of colors that cannot be used by the vertex out of the currently assigned colors. Initially all saturations are zero. The selection of the next vertex to be colored is based on maximum saturation, with ties being broken by the degree ordering.

The first i vertices colored, for some i, will form a maximal clique. Clearly we cannot use fewer than i colors on such a clique, and so this subset of vertices is optimally colored. By choosing vertices that are maximally constrained, that is vertices of maximum saturation, the hope is that there will be less probability of introducing unnecessary colors later in the coloring process. For example, Brélaz[**4**] shows that this algorithm will always correctly color 2-colorable graphs. On the other hand, 3-colorable graphs are given in [**22**] for which DSATUR uses $\Theta(n)$ colors.

Our version of this algorithm allows the initial ordering of the vertices to be any permutation. When DSATUR is coupled with the remaining algorithms, the initial ordering seems to make little difference in over all performance for most graph classes.

5. MAXIS: The Maximum Independent Set Approach

At the core of many coloring programs is a routine to find a large independent set in an uncolored subgraph. Once a satisfactory set is found, it is colored, and the independent set routine is applied to the remaining subgraph. The independent set routine may be a single pass with an heuristic for choosing the next vertex, or it may be a backtrack routine. In this section we describe our maximum set approach. We begin by describing the idea in general, in a way that makes it easy to compare to a wide variety of other similar approaches.

We describe an independent set algorithm pictorially in Figure 1. The independent set routine either implicitly or explicitly keeps three sets of vertices; the independent set I which is initially empty, the set of vertices S from which we may select a vertex to be added to I, and the set R of vertices which are adjacent to some vertex in I. When a vertex is selected for inclusion in I, its neighbors in the induced subgraph $G[S]$ are moved from S to R. When i vertices are selected we say we are at *depth* i, with the understanding that we may backtrack on the selection process. For notational convenience, we will use I_i, S_i and R_i to refer to the sets at depth i.

Several algorithms found in the literature can be described in terms of this picture by varying the selection process. An algorithm equivalent to the greedy algorithm can be obtained by ensuring that the vertices are selected from S in the order of the permutation given to the greedy algorithm. No backtrack is used in this case.

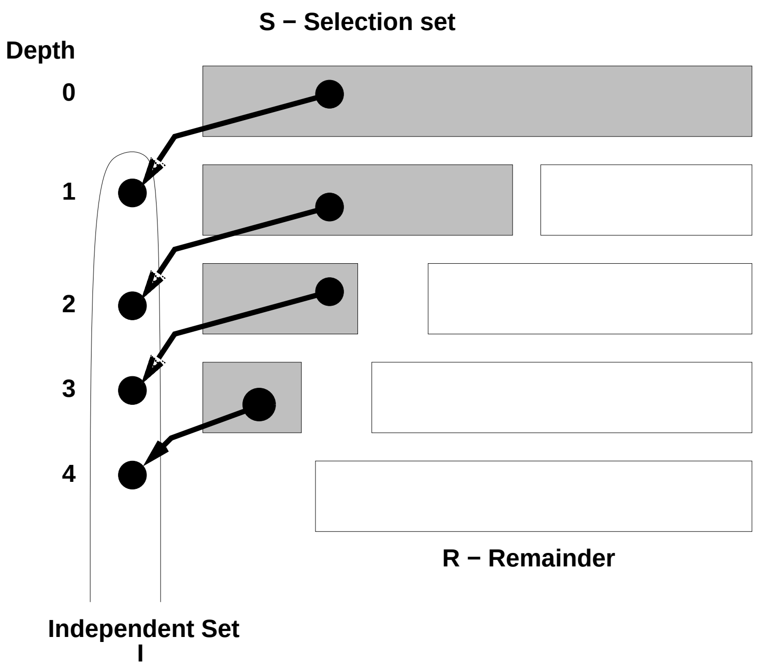


Figure 1. A Pictorial Representation of the Independent Set Routine

In [**10**] Johnson presents an algorithm that is guaranteed to give a coloring within a factor of $O(n/\log n)$ of the optimal. In this algorithm, selection at depth i is by minimum degree in the induced subgraph $G[S_i]$. No backtrack is used.

The idea behind using minimum degree is that if we use low degree vertices then the remaining subgraph will be larger. This should increase the likelyhood of finding a larger independent set. Johnson[**10**] shows that if we could guarantee to find the maximum independent set at each step, then we could color within a factor of $O(\log n)$ of the optimal. However, this is in itself a well known NP-complete problem.

On the other hand, we are not interested in the largest independent sets per se, but rather in the minimal number of independent sets required to cover the vertices. Leighton [**15**] developed the Recursive Largest First (RLF) algorithm to minimize the number of edges remaining in the uncolored induced subgraph $G[R]$. At depth 0, RLF selects the vertex of largest degree from S_0. At depth $i > 0$, RLF selects a vertex in S_i that has the maximum number of edges to vertices in R_i. The idea is that this leaves a sparser uncolored subgraph, decreasing the expected number of colors required for the remaining vertices.

The selection of the largest degree vertex from S_0 in RLF has the justification that since this vertex must receive some color, we may as well select it first, and thus likely reduce the number of edges in $G[R]$ when the independent set is completed. We combine the same idea with Johnson's algorithm [**10**], selecting the largest degree vertex in $G[S_0]$ and the minimum degree from $G[S_i], i > 0$.

Clearly, there is a balance to be sought between selecting vertices of high degree in order to reduce the number of edges remaining in the uncolored graph, and selecting vertices of low degree to obtain larger independent sets. To this end we also explored the idea of selecting vertices which are near the mean degree of vertices in the induced subgraph $G[S_i]$. The mean degree is computed and the sort is by absolute difference from the mean.

In Table 1 we show the performance of some interesting variations of this algorithm on graphs in $\mathcal{G}_{1000,\frac{1}{2}}$. A1 is Johnson's algorithm, A2 uses maximum degree for the selection from S_0, and minimum degree after that, A3 uses maximum degree within S_i for all i, A4 uses mean degree for all i and A5 uses mean degree on the first selection with minimum after that.

Not only did A2 give a significant color improvement over the Johnson algorithm, but it also reduced the execution time by about 10%. On the other hand, A3 increased the execution time, likely because of the extra calls to the independent set routine. A4 produces results similar to but slightly better than the greedy algorithm applied to a random permutation. In [**11**] extensive tests showed that RLF used from 105 to 110 colors with the highest proportion at 108, while Brelaz's DSATUR algorithm [**4**] used 113 to 119 colors on this class of graphs. Greedy typically uses 124 to 130 colors.

Johri and Matula[**12**] (see also Manvel[**17**]) Bollobás and Thomason[**3**], and

Graph	A1	A2	A3	A4	A5
1	108	105	153	119	108
2	110	103	150	118	109
3	110	105	150	121	109
4	109	105	151	120	109
5	106	104	151	118	108
6	108	105	150	119	109
7	109	107	153	120	109
8	108	104	151	122	105
9	111	104	151	122	108
10	109	103	151	120	108

TABLE 1. MAXIS without backtrack

Johnson et al [**11**] developed extremely greedy coloring algorithms. These algorithms extend the independent set routine by allowing backtrack and other ideas.

Bollobás and Thomason used a backtrack approach with heuristic pruning that gave large if not maximal sets. When the uncolored subgraph was reduced to about 200 vertices they switched to a brute force maximum independent set search, and in the range of 35 to 50 vertices they did a brute force coloring. The method of pruning was based on several parameters to limit searches when large independent sets were found. When more than one independent set of maximal size was found, the one with the largest total degree in $G[S]$ was kept. This tends to reduce the number of edges in the remaining subgraph. This approach yielded average colorings of 86.9 on a set of ten graphs from $\mathcal{G}_{1000,\frac{1}{2}}$. The approach used in [**12, 17**] was similar and yielded average colorings of 95.9.

Johnson et al [**11**] developed a program called XRLF which is based on Leighton's RLF [**15**]. Repeated trials are allowed for generating the independent set at each stage of the coloring algorithm, and at each depth i during each trial, random sampling is used to find a vertex from S_i of large degree into the set R_i. The independent set with the largest total degree into R is then used for the next color. When the remaining uncolored graph is small enough, a switch to brute force coloring is made. The switch points and the number of trials are parameters to XRLF. With appropriate parameters they found 86 colorings of graphs in $\mathcal{G}_{1000,\frac{1}{2}}$ in about 68 hours on a machine comparable to a VAX$^{\text{TM}}$ 750. (VAX is a trademark of the Digital Equipment Corporation).

We implemented an approach similar to that of Bollobás and Thomason[**3**] which we call MAXIS. We do not switch to brute force coloring for small remaining sets, relying instead on IG and TABU to clean up after MAXIS. We retain the idea introduced in [**3**] that when two or more independent sets of maximal size are found, we keep the one with the largest total degree. This reduces the number of edges in the remaining uncolored subgraph, and often improves the coloring. MAXIS does not compute R_i.

In MAXIS the vertices are sorted by the degree sequence of the subgraph, using one of three different sorting criteria. The sort is either by increasing degree, by decreasing degree, or by the absolute difference from the mean, which is computed beforehand. The vertices are sorted by degree sequence within each subgraph $G[S_i]$. When branching on selection, the vertices are selected in order. Once selected, a vertex is removed from S_i before the next branch. (It is thus possible, when using an incomplete search, that the set found will not be maximal, but the probability of this would appear to be low.) The effects of different sorting mixes will be described in the experimental sections.

To limit the search time, we prune the search. Various methods of limiting the branching factor were tried[**5**]. In order to provide maximum user control, the current version of the program takes as input a sequence of integer pairs $[n_1, l_1] \ldots [n_k, l_k]$, where the number of pairs is specified by the user. The n_k are in decreasing order so they specify a set of intervals (with a default largest value of $n + 1 = |V| + 1$). MAXIS determines the interval $n_j \leq |S_i| < n_{j+1}$ for the current subgraph S_i, and then uses the corresponding l_j as the branch limit at this point. This hackers-delight allows us to fine tune the truncation.

Finally, we use a cutoff rule to avoid search that cannot possibly improve on our current best independent set. The rule used is: *If the number of untried vertices in the current non-adjacent subgraph S_i plus the number in I_i is* less than *the number in the best independent set found so far, then backtrack.* If we replace "less than" in this rule by "less than or equal" we get a still smaller search, but this would prevent the generation of equal sized sets, and thus prevent selection based on total degree. This rule slightly improves running time on random graphs, but is most effective in reducing running time on k-colorable graphs, where k is small with respect to the expected coloring of a random graph.

For example, in $\mathcal{G}_{1000,\frac{1}{2},40}$, the specified sets are of size 25. When the partially complete independent set I_i has 15 or more vertices, it is highly probable that it is a subset of one of the specified sets, and that all vertices in the non-adjacent subgraph S_i are in fact also in the same specified set. However, if the last cutoff pair is $(0, 5)$ for example, without the rule MAXIS would search many combinations of this specified set in vain. Additionally, after finding a set of size 25, background sets of size 14 or 15 will almost never be fully expanded, since the corresponding selection sets S_i leading to them will be too small. Experiments showed significant time savings on such graphs.

6. TABU Algorithm

Hertz and de Werra[**9**] describe the application of a technique called TABU search developed by Glover[**7**]. Using this technique they were able to color graphs in $\mathcal{G}_{1000,\frac{1}{2}}$ with 93 colors. Using a more sophisticated technique, in which they first found large independent sets using TABU and then applied TABU coloring to the remaining vertices, they obtained at least one 87 coloring.

The TABU algorithm in one sense is dual to the greedy algorithm. The greedy algorithm generates independent sets by adding vertices one at a time so that no conflicts are created. TABU starts with a partition of the vertices, with vertices assigned at random, and then tries to remove conflicts by re-assigning vertices.

More precisely, suppose at step t we have a partition $s_t = (V_1, \ldots, V_k)$. We call s_t a *point* in the search space. We may generate a new point (referred to as a *neighbor* of s_t) by selecting at random any vertex in conflict, and moving it at random to any other partition element.

TABU works by generating a set of such neighbors, and then selecting the neighbor with fewest conflicts to be the new partition at step s_{t+1}. To prevent cycling through a small set of good but suboptimal points, a list of the last q moves is kept, and the reversals of these moves are not allowed. As each new move is generated, the oldest move is removed from the TABU list, and the newest move is added.

Note that the number of conflicts could refer to either the number of edges or the number of vertices, or some function of the two. The choice will affect the nature of the subgraphs of vertices that remain in conflict as the algorithm progresses. Minimizing vertices will generally tend to result in dense subgraphs, while minimizing edges allows for sparser subgraphs. Hertz and de Werra use the number of edges, but do not discuss the trade-offs. In limited tests, this seemed to be as good as any other measure we found.

They do suggest a number of improvements to their algorithm. One is the use of an aspiration function, another is the immediate acceptance of a neighbor if its conflict value is less than the current value. They also use a type of brute force search when the number of conflicts is small. Out TABU algorithm does not use aspiration, requires at least a minimum number of neighbors be generated even if a better solution has been found, and uses a type of brute force search when the number of conflicts is small.

Parameters to our TABU are the number of neighbors to be generated, the minimum number of neighbors and the size of the TABU list. Following Hertz and de Werra, we use a TABU list of size 7. For 1000 vertex graphs we usually generate 600 neighbors chosen at random, while for 300 vertices we generate only 200. The minimum number of neighbors is usually set between 2 and 10. The idea behind this last is that if it is very easy to generate an improved neighbor, then we are likely at a point where there are many improvements and so should look for an even better one.

Using these techniques, we were able to get results similar to those reported in [**9**] for colorings down to the mid 90's on $\mathcal{G}_{1000,\frac{1}{2}}$.

7. Combining The Algorithms

One of the problems inherent in the TABU design is its all or nothing character, at least if the goal is to find a proper approximate coloring. TABU starts with a partition of the desired size, and attempts to remove conflicts until there are none, or terminates because it has exceeded some upper limit on its search time. Of course, it will always give an approximately correct solution, which may be useful if the goal is to color vertices with the given number of colors while minimizing the number of conflicts.

We can build a composite algorithm from TABU and Iterated Greedy as follows. First, we start with some valid k-coloring of the graph. We then set $k' = \max\{k-c, k_t\}$ where c is an input parameter called *step down* and is usually small, and k_t is a *target coloring* also input to the program. (The program terminates when a k_t-coloring is found). We sort the vertices using the current coloring by decreasing size of independent sets, use the first k' sets to seed the partition, and then distribute the remaining vertices (from the $k' - k$ smallest sets of the current coloring) over these sets. In our implementation, each vertex is tested in turn to see in which independent set it creates the fewest conflicts, and is added to that set. In this way we hope to give TABU a boost over a random partition as an initial step.

Second, if TABU fails to find a correct coloring in the prescribed time, then the partition number is incremented by one, and the process continued. This means that the newest partition element is initially empty, but vertices in conflict can be moved into it. An untried improvement might be to move some portion of the conflict vertices into this new element immediately.

Third, after repeating the second step c times, which means the partition is of the same size as the current coloring, we can use IG to rearrange the current best coloring and try again.

Our overall process can now be described:

(i) Generate an initial coloring using some algorithm, for example Brélaz or greedy.
(ii) Perform some number of IG steps.
(iii) Apply TABU as described above.
(iv) Repeat steps (ii) and (iii) until a k_t-coloring is found or iteration limits are exceeded.

In step (iii), we use the best result found by IG (the coloring with the smallest coloring sum found so far) as the initializer. If TABU fails to find a reduced coloring, then we allow TABU to attempt to find a new coloring of the same size, and if it does we use that as the new coloring to initialize the next IG step. In this way, we have a chance of getting out of valleys that can trap IG.

In case of failure, the grand loop is terminated by a fixed upper limit on the number of iterations, usually from 2 to 10. IG and TABU are terminated after some upper limit of iterations have occurred since the last improvement. For TABU, an improvement is a reduction in the least number of conflicts seen so far. For IG, an improvement is a reduction in the sum of colors.

Finally, to allow comparisons to pure TABU coloring strategies, we can start the process with a trivial n-coloring, allow no iterations of IG and then start

TABU to search for the desired coloring. This will differ from the Hertz and de Werra method in that the initial partition will be generated by inserting each vertex in (random) order into the partition element that minimizes the number of conflicts with it.

Part III. Experimental Results

8. Introductory Comments on Experimental Methods

As described in preceding sections of this paper, each of the algorithms has several parameters that can be set in various ways to direct the search. The interactions of these parameters can be complex, even within single algorithms.

For example, in section 5 we noted that there is a trade-off between finding large independent sets and leaving an uncolored subgraph that is sparse. In [**5**] it was observed that MAXIS performed best when the vertex from S_0 was selected according to the mean degree sorting criteria, with the remaining selections using the minimum degree sorting. This advantage is particularly evident on equipartite graphs, as shown in Table 2. The numbers in parentheses are the cutoff limits described in section 5. Notice how these results contrast with the results

	Colors Used When S_0 Degree Sorted By		
Graph	Mean	Minimum	Maximum
1	66	86	85
2	63	84	75
3	66	84	85
4	69	85	78
5	67	87	74
6	73	87	80
7	66	85	82
8	76	71	83
9	65	82	86
10	67	84	85
Total	678	835	813

TABLE 2. MAXIS Using $(500, 8), (200, 6), (10, 3), (0, 10)$ on $\mathcal{G}_{1000,\frac{1}{2},60}$

in Table 1, where the non-backtracking version performed best when the first vertex was selected by maximum degree.

The trade-offs in setting the various parameters are not easy to predict in backtracking and long running neighborhood searches, especially when combined with variations in the classes of graphs to be colored. One can easily imagine then that the interactions between different algorithms using different approaches and parameter settings are even more difficult to predict, and are sometimes surprising.

Due to the large number of graph classes, algorithms and parameter settings, we cannot hope to detail all the possible interactions. This surfeit of research opportunity is not helped by our propensity to generate ever more tuning mechanisms as we look deeper into each algorithm. Our approach in this paper is to use good settings for each algorithm determined independently, then to look at the interactions these algorithms have on certain classes of graphs, and in this way to discover some broad indications of the interaction effects.

Unless otherwise stated, all experiments using MAXIS will use the mean sorting heuristic for selection from S_0, and minimum sorting for the remaining S_i. In most experiments, the branching limits will be set by a single pair $(0, h)$, for small values of h, meaning that at every depth the maximum branching will be h. (It may be less if $|S_i| < h$). This has the advantage that it is easier to see how the search tree size varies with variations in the settings; the search increases with increasing h. Also, more complex tunings tend to be more graph class specific, and in application settings we may not know what class of graph we are being asked to color. This particularly applies to k-colorable graphs, where the usual measures of degree and edge density do not indicate the unexpectedly good coloring.

For TABU, we will use a neighborhood of 600 when dealing with 1000 vertex graphs (or larger), a TABU-list size of 7 and a minimum neighborhood of 2. For some of the 300 vertex graphs, a neighborhood of 200–300 will be used.

For IG, we will use 50:50:30 or 70:50:30 as the ratios of **Largest-Reverse-Random** heuristics.

The experimental results reported here were taken on a variety of machines and over a substantial period of time. When CPU times are reported, they are from experiments run on a SUN SPARC 10/40, with 96 megabytes of memory. The published rating using the SPECInt92 system is 50.2. The baseline for this rating is a VAX 780. We would caution the reader against placing too much confidence on machine timings when comparisons are made to other machines and programs. In particular, although we try to make our code as fast as possible, we cannot guarantee the relative coding efficiencies of our different programs. We have also noticed variations of up to 25% when experiments are repeated on the same machine using the same code.

When possible, we try to emphasize machine and implementation independent measures, such as the number of iterations of the greedy algorithm, or of the TABU algorithm, and how these measures vary as certain parameters of the algorithms or the generators of the input graphs are varied. When timings are used for comparison purposes we try to indicate how the variations in time correlate with changes in these parameters.

9. Comparison of Algorithms on Random Graphs

In order to give the reader some indication of the relative efficiency of our different algorithms, and to provide at least one point of comparison to similar algorithms in the literature, we provide some experimental results on graphs in $\mathcal{G}_{1000,\frac{1}{2}}$. In Figure 2 the number of iterations required to produce various colorings of $\mathcal{G}_{1000,\frac{1}{2}}$ is plotted for ten different example graphs. In this case, the initial coloring is obtained by the greedy algorithm on a random permutation. The algorithm was terminated when either a 102-coloring was obtained or 10000 iterations elapsed from the last improvement. It terminated on a 103-coloring in three cases. IG has been used to obtain 100-colorings and even a 99-coloring of a $\mathcal{G}_{1000,\frac{1}{2}}$, but the number of iterations increased enormously. It seems likely that 100–102 is the limit that one can normally expect of IG in an hour or so on our benchmark machine.

If DSATUR is used to obtain the first coloring, the initial colorings are reduced from the 122–126 range to the 113–117 range. In Table 3 CPU times are compared for obtaining 110 and 106-colorings with and without DSATUR on a sample of ten graphs. DSATUR uses approximately one second, so in each case the time reflects quite accurately the number of iterations used. The number of iterations and CPU time are reduced in these examples, but the difference is well within one standard deviation. Other tests yield no clear indication that DSATUR improves the performance of IG on these graphs. For any given graph, it seems equally likely that the use of DSATUR will increase the number of iterations to reach a target color, as it is to decrease the number.

	Pure IG CPU		DSATUR/IG CPU	
Color	Mean	Std. Dev.	Mean	Std. Dev.
110	4.2	1.9	4.0	1.7
106	35.4	16.9	31.8	13.5

TABLE 3. Pure IG versus DSATUR/IG on $\mathcal{G}_{1000,\frac{1}{2}}$

On a set of 10 random graphs TABU obtained 95-colorings in average of 942 seconds(standard deviation(std) 415). The maximum time used was 1668 seconds. TABU has been used to obtain 93-colorings or better [**5**, **9**] but at increased cost. Thus, TABU significantly outperforms IG on this class.

However, allowing IG to perform up to 200 iterations without improvement and then switching to TABU leads to further improvements. The average CPU time over these ten graphs to obtain a 95-coloring was reduced to 473 seconds(std 213). Other experiments [**5**] also confirm that IG mixed with TABU gives improved performance over either algorithm alone on this class. However, the direction of improvement is one way; the improvement comes from using IG to prepare a coloring to hand to TABU. Once the coloring is below 95 on $\mathcal{G}_{1000,\frac{1}{2}}$,

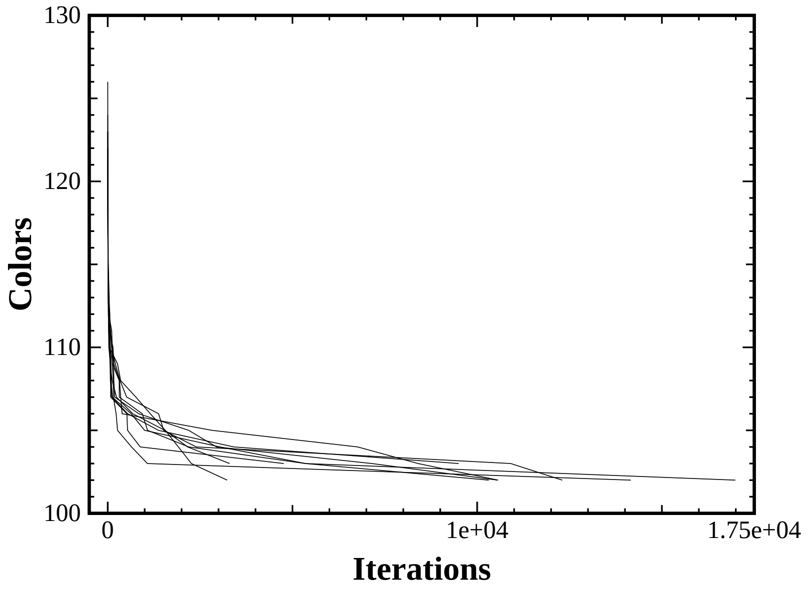


FIGURE 2. Performance of IG on $\mathcal{G}_{1000,\frac{1}{2}}$

it is quite rare for IG to reduce the coloring handed to it by TABU even when tens of thousands of iterations are allowed.

No mixture of IG and TABU competes well against the MAXIS approach on this class of graphs. Using MAXIS with the cutoff limit $(0,3)$ 90–93-colorings were obtained in an average of 83 seconds(std 2). Initializing TABU using these colorings, 89-colorings were obtained in every case in an average of 101 seconds total(i.e. including the MAXIS time)(std 26). The maximum time used was 165 seconds.

In Table 4 we show what MAXIS can do when allowed a more extended search followed by IG and TABU. In this search, the cutoffs for MAXIS were set at $(600,7)(400,6)(0,5)$, IG was allowed 5000 iterations and then TABU attempted to reduce the coloring by another 4. TABU was allowed 50000 iterations without improvement before the coloring number was stepped up. Thus, depending on whether it was handed an 88 or 89-coloring, TABU first attempted an 84 or 85-coloring. Finally, the IG/TABU main loop alternation was allowed to continue for 3 iterations. However, all the coloring improvements were obtained in the first main loop iteration. Note that on the second and third main loop iterations TABU first attempted an 83-coloring, as that was specified as a target coloring, and the 4-color step down from 86 or 87 would be less than or equal to 83. TABU would then step up to $84, 85, \ldots 86$ (or 87) before failing, allowing 50000 iterations each step after the last improvement.

The colors obtained by each algorithm are shown and the times for obtaining the colors are in parentheses. These times are running totals, that is they include the times of the previous coloring. The total CPU time is the total spent over the three main loop iterations. Note that not only did IG improve on MAXIS only once, using only a second to make the improvement, but also this coloring was further improved by TABU.

Graph	MAXIS	IG	TABU	Total CPU
1	88 (1725)		86 (4210)	12201
2	89 (1730)		86 (2967)	8659
3	89 (1710)		86 (3164)	9050
4	89 (1705)		87 (3608)	14519
5	89 (1683)		87 (3308)	11624
6	88 (1627)		87 (4189)	13057
7	89 (1667)		87 (3381)	12372
8	88 (1801)		86 (4848)	10912
9	89 (1736)		86 (2727)	9254
10	89 (1681)	88 (1682)	86 (4812)	14750

TABLE 4. Colorings and times using MAXIS/IG/TABU on $\mathcal{G}_{1000,\frac{1}{2}}$

These colorings are approximately comparable in time and efficiency with other published results using maximal independent set approaches[**3, 11**].

10. The k-colorable Landscape

In this section we explore the coloring landscape of various classes of k-colorable graphs described in section 2. We now multiply the possible interactions between the different algorithms by all the different parameterizations of the graphs. However, certain trends and characteristics do become apparent. We will try to identify the most obvious of these, but much future research needs to be done.

10.1. Performance of IG on the Landscape. We summarize here, mostly in the form of graphical plots, the behavior we have found for IG over various classes of k-colorable graphs. The hardest graphs to *approximate*, for a fixed n, k, p that we have so far found, are the flat graphs. (In all of the experiments in this paper we use flatness zero). However, of the more or less "natural" classes the equi-partite are the most difficult, and these are the main focus of this presentation.

In Figure 3 we display typical behavior for coloring graphs from $\mathcal{G}_{n,p,k}$. The difficulty in coloring these graphs seems to be fixed in a plateau region, in this case from about 101 down to approximately 90 colors. It is within this plateau that the number of iterations grows exponentially as k increases. The figure is constructed by choosing the graph requiring the median (rank 5) number of iterations out of a set of 10 coloring runs for each class.

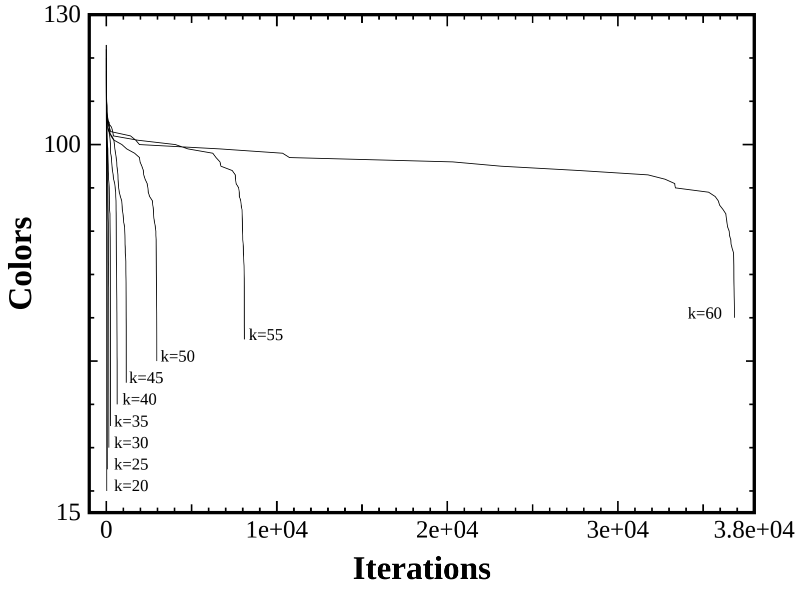


FIGURE 3. The Growth of the Plateau on $\mathcal{G}_{1000,\frac{1}{2},k}$, $20 \leq k \leq 60$

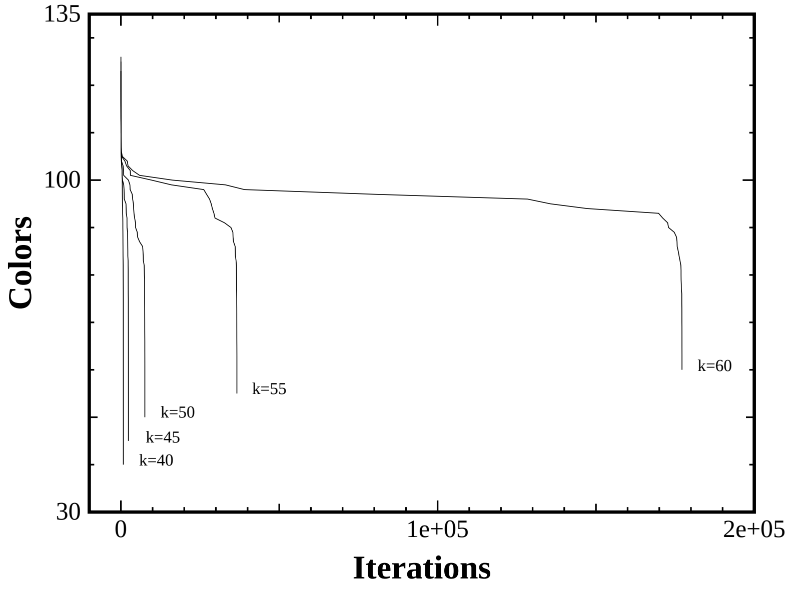


FIGURE 4. The Plateau on 1000 vertex flat graphs, $p = \frac{1}{2}$

In Figure 4 the same presentation is made for flat graphs, where the median is chosen from a sample of 5 experiments. Although the figures look similar, it should be pointed out that the scaling may be misleading. The 60-colorable flat graph required over 177000 iterations, whereas the equi-partite 60-colorable graph required less than 38000. Thus, the 55-colorable flat graph required about as much work as the 60-colorable equi-partite graph.

In Figure 5 we show how the search characteristics change when we allow the partition elements to vary in size, using equi-partite, smoothly variable and highly variable graphs. These are $n = 1000, k = 40, p = 0.5$ graphs, which fall in the region of intermediate difficulty. We plot the progress of the algorithm for 15 graphs in each case. The results are similar for smoothly and highly variable graphs, with the highest variations showing a slightly sharper curvature for the first case. In these graphs, increasing the variation gives us graphs which are easier to approximate. However, in some cases obtaining the specified coloring actually takes longer than for equi-partite graphs. Apparently the independent sets of certain sizes are more difficult to distinguish from the background. The largest sets will stand out, and the small sets can be replaced with naturally occurring sets. For large values of ϵ in these graphs the algorithm occasionally finds a coloring less than 40. In these cases the smallest partition is empty, or so small that it can be successfully merged into a larger set.

Similar changes occurred when we allowed the probability to vary between different partition element pairs. It seems that the greater the variation allowed in any of the graph generator parameters, the easier it becomes to obtain a good approximation. It was with these observations in mind that the flat graph generator was designed. A set of ten experiments is shown for flat graphs on 1000 vertices, with $k = 40$, in Figure 6.

Apparently the algorithm must sort out the independent sets of the specified k-partition of the graph from the independent sets that occur due to the random edge assignments. This suggests, and the empirical evidence supports, that the maximum difficulty will occur when the independent sets we create are about the same size as the largest that occur in a random graph of the same density.

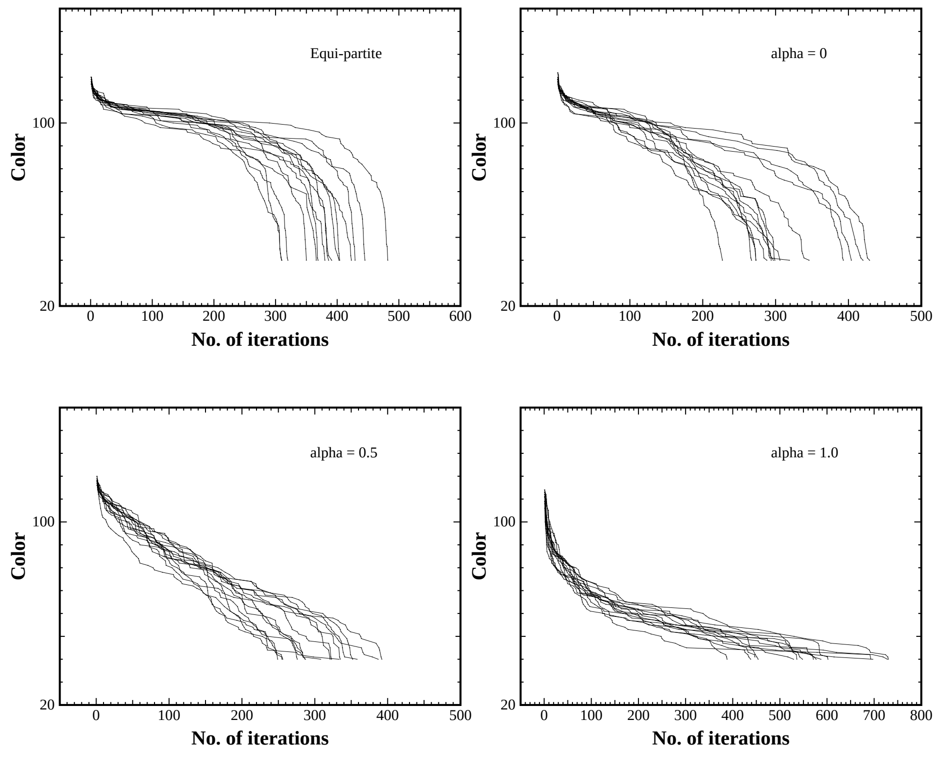
Matula's formula [**18**] (see e.g. [**21**] for an exposition) for the maximal independent set size is

$$\mathcal{I}(n) = 2\log_b n - 2\log_b \log_b n + 1 + 2\log_b(e/2)$$

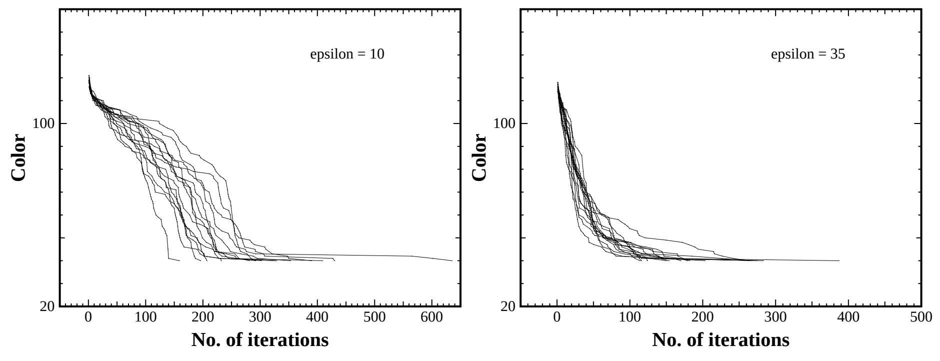
where $b = 1/(1-p)$. When we set $k \approx n/\mathcal{I}(n)$ we find that IG, and most other algorithms, simply cannot obtain the specified coloring. (As in [**20**], to be more precise at high probabilities, we should adjust for the density we induce by not adding edges within the partition elements).

Assuming that this is a reasonable approximation to the boundary of the region where difficulties arise, we let

$$k = k_r(n) = \frac{n}{r\mathcal{I}(n)}$$



Smoothly Variable



Highly Variable

FIGURE 5. The Effect of Variability in Partitioning

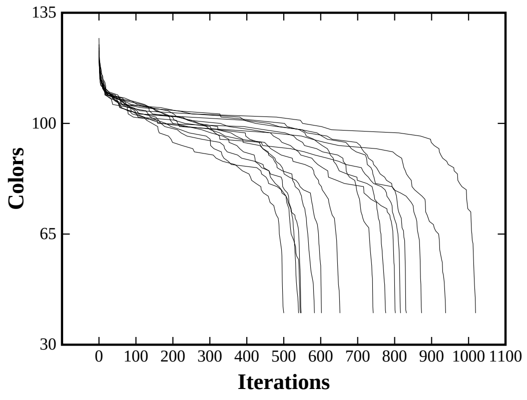


FIGURE 6. Pure IG on flat graphs $n = 1000, p = \frac{1}{2}, k = 40$

for various values of r. In Table 5 we explore the increase in difficulty as the edge density varies for fixed n. The graphs have 1000 vertices. Ten graphs were used for each data point. The values of k are chosen so that the indicated r values are approximated. As one would expect, it takes considerably longer to find the k-coloring as the density is reduced for fixed values of r, especially near the boundary. The ratio of **largest:reverse:random** used in the heuristics is 50:50:30. Other tests showed that altering the program parameters as the density varies seems to make little difference in performance over the considered ranges. For a summary of how IG varies as the parameter ratios vary, see the next subsection.

	$r = 1.5$			$r = 2.0$			$r = 2.5$		
		Iterations			Iterations			Iterations	
p	k	mean	std	k	mean	std	k	mean	std
0.25	22	10439.0	3176.6	17	644.7	235.6	13	98.1	19.8
0.30	26	4580.4	1146.3	20	375.3	110.3	16	119.8	17.7
0.40	34	1386.6	233.8	26	205.7	33.3	20	59.1	12.2
0.60	54	364.3	82.7	40	67.7	14.3	32	29.3	5.8
0.70	66	153.2	23.8	49	37.2	4.7	39	20.5	3.1
0.75	73	96.3	18.7	54	24.9	3.8	43	13.4	2.9

TABLE 5. Effects of varying density on $n = 1000$ equi-partite graphs.

For lower densities, the independent sets are larger, and for the considered ranges, there are more maximal sets. Thus, it is perhaps not surprising that IG requires more time in general to sort out the hidden coloring on graphs of lower density.

Discussion of IG in section 3 shows how IG *could* work, in the sense that it never produces a worse coloring, but allows for improved colorings. But it is less clear why it *should* work on k-colorable graphs. That is, what is the mechanism that causes it to find the distinguished sets so consistently when $k << n/\mathcal{I}(n)$?

It is worth noting that a single iteration of greedy is expected to perform relatively poorly on k-colorable graphs [**23, 14**], and indeed it usually colors k-colorable graphs with a coloring similar to what one would expect on a random graph with the same edge density.

The drift through the plateau takes a relatively long time. Let us suppose for now that it is just a random search process, except that the coloring number is not allowed to increase. How do we explain the rapid drop below the plateau?

We present an intuitive illustration based on Figure 7.

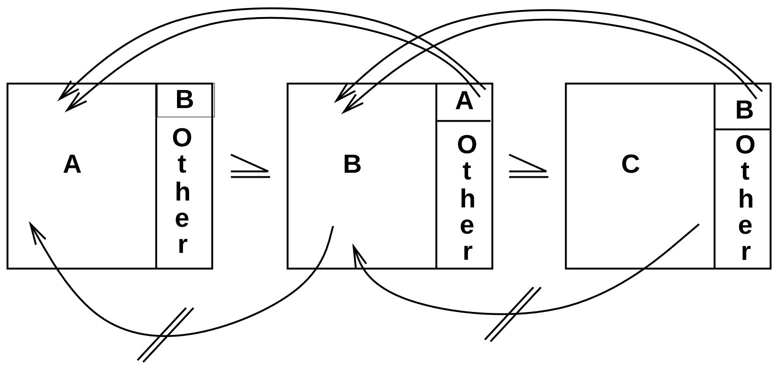


FIGURE 7. Illustration of IG's search

The basic observation is that once a small enough coloring has been discovered, then with high probability there *must* be some sets in the coloring that have large intersection with the specified sets. In Figure 7 this is illustrated by the relative sizes of the regions in each box, where the boxes represent the color classes that have been identified by the algorithm, and the regions represent the distribution of the vertices of the various specified sets in the k-colorable graph.

Now consider what may happen when the greedy algorithm is coloring the vertices in the center set. We assume that the boxes to the left represent the new coloring of this partial pass of greedy, while the boxes to the right are sets from the previous coloring. A vertex from the specified set 'B' may be colored with the same color as the vertices of the leftmost set only if no edge exists between any vertex in that set and the vertex in 'B'. The probability of this is quite low, since every vertex not in the 'B' region of the left box will have an edge to a 'B' vertex with probability p. On the other hand, 'A' vertices in the center set will have a much higher probability of being added to the leftmost set, since only the non-'A' vertices can have edges to 'A' vertices, and there are not very many of those. Similarly, when greedy reaches the vertices of the rightmost set, the 'B' vertices are more likely to be moved into the center set than are the non-'B' vertices. Thus, on each pass, few 'B's leave the set dominated by 'B's and there is a differential tendency to add more 'B's to it.

This example illustrates the tendency of IG to amplify concentrations of vertices from large independent sets into specified color sets. This appears to be the driving force behind IG, although our understanding is as yet far too crude to predict such things as the edge of the plateau, the depth of the plateau, the rate of descent, the effect of varying the reordering heuristics, or the paradoxical effects of mixed algorithms discussed in sections 10.3 and 11.

However, this example does indicate why failure occurs as k approaches the boundary. In this case, the specified sets have no advantage over the huge number of other maximal independent sets in the graph. These other sets are unlikely to yield a k-coloring.

These arguments also yield an intuition as to why increasing the variance in the sizes of the specified sets seems to aid the algorithm in obtaining an approximation. The larger the largest sets are, the more likely it is that a color set will have a higher proportion of vertices from them, and the more susceptible these sets are to amplification.

10.2. Effects of Varying the Heuristics in IG. In this section we examine the effectiveness of IG on graphs from the class of 1000 node equi-partite graphs $\mathcal{G}_{1000,\frac{1}{2},k}$ for $k = 20 \ldots 60$ as we vary the ratios of the reordering heuristics. The basic observation is that for $k \leq 55$, the performance is relatively immune to small changes in the parameter settings. However, for $k = 60$ variations up to a factor of 2 are not uncommon.

If we compare Table 6 to Table 7, we find that the averages for graphs with k up to 55 are indeed similar, but at $k = 60$ there is a dramatic switch. The experiments in Table 6 used the heuristic mix (Largest First 50), (Reverse 50), (Random 30). For Table 7 the heuristic proportions (Largest First 50), (Reverse 20), (Random 5), (Largest Degree 5), and (Smallest First 5) were used. These were proportions that had been found to work well in earlier experiments.

Earlier tests indicated that the use of degree sequence was not particularly helpful. So the question arises, what causes the difference? Some additional tests were run, and the results are summarized in Table 8, including the previous results. From this it seems probable that the primary consideration is the proportion of **Largest first**. At least one earlier test indicated that 70:50:30 was superior to 50:50:30 at $k = 60$ although this is not evident from the example set in Table 8. The only thing that is clear is that there is high variability as we near the ridge and we should not put too much emphasis on any results that differ by a small constant factor. However, for $k = 60$ it does seem that increased ratios of **Largest First** are more efficient.

In subsequent sections, we typically use the 50:50:30 or 70:50:30 mixes, because large numbers of tests support their use for most ranges where k is far from the boundary and we want results that are comparable over different experiments. In particular, when we compare IG to mixed strategies involving

Color	Min	Max	Mean	Dev	95% Confidence
20	12	26	18.9	3.5	2.5
25	34	53	41.5	5.9	4.2
30	62	96	77.9	12.9	9.2
35	104	202	166.5	30.0	21.5
40	282	551	377.2	78.1	55.9
45	737	1370	942.5	221.9	158.7
50	1370	3364	2256.2	667.1	477.2
55	5285	10945	7399.1	1750.0	1251.8
60	47128	104562	75762.8	21126.0	15111.6

TABLE 6. Equi-partite Iterations using 50:50:30

Color	Min	Max	Mean	Dev	95% Confidence
20	15	30	21.6	4.7	3.3
25	30	60	46.0	8.9	6.4
30	60	127	90.9	23.0	16.4
35	146	223	189.9	23.4	16.7
40	301	504	406.7	65.9	47.1
45	636	1370	1043.5	220.5	157.7
50	2237	3041	2693.6	250.1	178.9
55	5034	8148	6462.5	1064.1	761.2
60	25792	39106	31535.9	4311.7	3084.2

TABLE 7. Equi-partite Iterations using Mixed Heuristics

	Largest	Reverse	Random	Smallest	Larg. Deg.
Mix 1	50	50	30		
Mix 2	70	50	30		
Mix 3	50	20	10		
Mix 4	50	20	5	5	5
Mix 5	50	20	5	5	
Mix 6	50	20	5	15	

	Min	Max	Mean	Dev
Mix 1	47128	104562	75762.8	21126.0
Mix 2	38243	118061	72793.7	22596.7
Mix 3	19733	48126	34417.7	8589.3
Mix 4	25792	39106	31535.9	4311.7
Mix 5	22769	55206	32499.7	10550.3
Mix 6	27009	58320	44175.6	12635.1

TABLE 8. Iterations Required on $\mathcal{G}_{1000,\frac{1}{2},60}$ with Various Heuristics

TABU and MAXIS in the following sections, we use the same heuristic mix in all experiments. This should isolate the effects of TABU and MAXIS on IG. It is possible that with different heuristic mixes the paradoxical results we find in those cases could change. However, those results are broadly consistent with additional experiments and for TABU are consistent with the experiments reported in [**5**].

In [**5**], it is shown that DSATUR seems to offer little long term advantage in the classes and ranges of probabilities we are considering. The results are similar to the results on random graphs, in that basically the DSATUR algorithm being a type of one-shot greedy algorithm, only replaces the first few iterations of greedy where the coloring is still well above the plateau.

10.3. TABU on the Landscape. In this section we will look at applying the TABU algorithm and mixed approaches to the k-colorable graphs. We consider mainly equi-partite graphs.

Although the TABU algorithm will color most equi-partite graphs, the time required is much greater than for the IG algorithm. In Figure 8 we present a plot of iterations versus conflicts for representative graphs for $k = 40, 45, 50$, where we chose the graph of rank 5 out of ten experiments in each case. TABU was started in these cases without any greedy coloring. Figure 9 displays the distribution of the ten runs on $k = 50$. Notice in particular the large variance. For 55-colorable equi-partite graphs, on a set of ten experiments where the number of iterations without improvement was set to 100,000, TABU succeed in finding 55-colorings only three times. The successes required from 600,000 to 1.23 million iterations, the latter requiring nearly 5 hours to complete. No successes have been recorded for TABU on 60-colorable graphs.

Table 9 contains CPU times from ten graphs for both TABU and IG for $k = 40$ to 50.

	IG CPU		TABU CPU	
k	Average	Dev.	Average	Dev
40	23.4	4.6	274.7	132.6
45	53.7	18.5	1127.2	383.5
50	143.3	35.9	2889.2	1594.6

TABLE 9. Comparison of running times of IG and TABU on $\mathcal{G}_{1000,\frac{1}{2},k}$

TABU seems even less effective on flat graphs. Table 10 displays some indication of the number of iterations used from a sample of five graphs. At $k = 35$ the $*$ indicates that only four of the five were successful. The fifth failed after 296000 iterations. The average is taken over the successful attempts. For $k \geq 40$ TABU has not been successful.

These results for TABU on k-colorable graphs contrast with the comparative performance on random graphs. In the section on random graphs it was noted

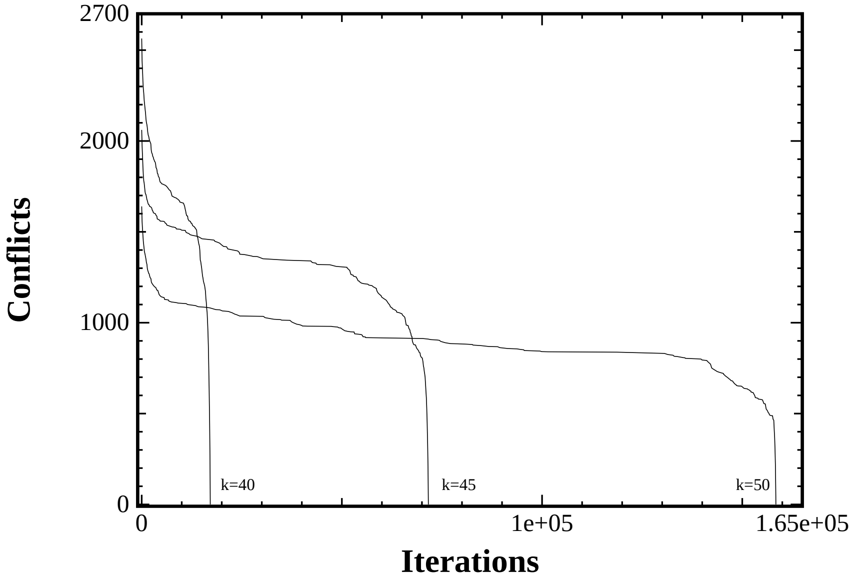


FIGURE 8. Conflict reduction by TABU on $\mathcal{G}_{1000,\frac{1}{2},k}$

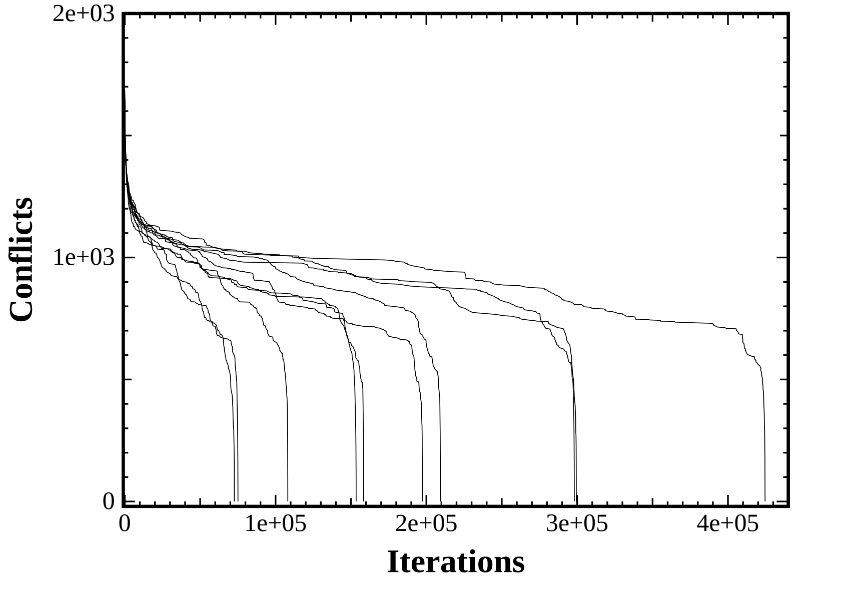


FIGURE 9. TABU on $\mathcal{G}_{1000,\frac{1}{2},50}$

	Iterations of TABU		
k	Average	Max.	Min
20	1547	1703	1369
25	4978	7919	3323
30	14181	27845	7942
35*	138492	271403	83928

TABLE 10. TABU on flat graphs of 1000 vertices

that a mixture of IG followed by TABU improved the performance over either algorithm alone. In [**5**] it was noted that this mixture does not help on k-colorable graphs. Various attempts have been made to use TABU to reach the middle or bottom of the plateau, and then have IG finish the job by reducing the coloring to a k-coloring.

The results are paradoxical. The use of TABU is problematical on these graphs. Tables 11 to 13 display typical results for this approach. For each $k = 50$ to 60, ten experiments were run. Initially IG was used to quickly reduce the coloring to 105–107, by letting it run until 100 iterations without improvement were made. Then a switch was made to TABU which searched for and found a 96-coloring. Then IG completed the coloring in each case.[1]

As expected the time to find the 96-coloring was greatly reduced when TABU was used. Note however, that for $k = 50$ and 55 the average number of iterations of IG to find the k-coloring actually increased when TABU was used. Also, this number does not include the iterations of IG before the switch to TABU is made. Thus, TABU was a hindrance on average. This is consistent with many attempts to use TABU on these graphs.

The results for $k = 60$ are the first that show a consistent improvement with TABU on equi-partite graphs. In all ten cases, the time to completion was reduced by about one half. However, the reader is reminded of the variations that can occur on 60-colorable graphs discussed in section 10.2, with particular reference to Table 8. To see if TABU could be of further assistance ten experiments were run in which TABU was used to find a 93-coloring. An average of 325 seconds was used to obtain the 93-coloring(std 192). (One instance required 772 seconds, the rest were in the 200–390 range). One of these experiments was terminated when IG was unable to reduce the 93-coloring after more than 110000 iterations. On the remaining instances IG used an average of 71648 iterations(std 20784). Thus, increased use of TABU apparently *increased* the cost to IG.

Clearly, if TABU was able to reduce the coloring far enough (in the extreme to the specified coloring) then the final search by IG will be reduced (in the extreme to zero iterations). With this thought, TABU was set to search for

[1]The *-entries indicates that in the $k = 50$ case, in 2 instances the initial IG attempt actually completed the 50-coloring. Only the 8 instances in which the switch to TABU was made are included in the analysis. Some of the timings came from earlier experiments, when for unexplained reasons the CPU time was frequently increased by up to 25%.

Graph	Pure IG		IG/TABU	
k-color	Seconds	Dev.	Seconds	Dev.
50*	143.3	35.9	58.2	17.3
55	278.0	56.5	89.4	29.5
60	2942.1	1032.7	68.5	21.5

TABLE 11. Time to obtain a 96-coloring

Graph	Pure IG		IG/TABU	
k-color	Iterations	Dev	Iterations	Dev
50*	2430	599	3390	1771
55	7224	1528	10321	3698
60	72794	22597	39083	11487

TABLE 12. IG Iterations to obtain specified coloring: final phase

Graph	Pure IG		IG/TABU	
k-color	Seconds	Dev	Seconds	Dev
50*	143.3	35.9	253.1	108.9
55	423.7	89.5	684.8	240.4
60	4250.1	1327.7	1837.8	516.5

TABLE 13. Total time to obtain specified coloring

a 90-coloring, with 100000 iterations allowed from the last improvement before incrementing the coloring. IG was allowed up to 70000 iterations from the last improvement. (Recall that 50000 were sufficient in all cases when TABU was not used). The attempt was made on the same set of graphs, but the experiment was terminated early before the fifth graph completed. Of these, TABU failed to achieve a 90-coloring in two cases, producing instead a 91-coloring which required 3883 seconds in the first graph and 2674 seconds in the fifth graph. The 90-colorings required from 2958 to 4133 seconds. IG reduced the first 91-coloring to a 60-coloring after 183259 iterations (12018 seconds total). It took 181606 iterations to reduce the 91-coloring to a 90-coloring, but then very quickly completed the reduction. IG failed to improve *any* of the 90-colorings, although it used 176182 iterations in one attempt. The fifth search was terminated after 70000 iterations failed to reduce the 91-coloring.

It is not yet clear why TABU is having such a negative effect on IG on this class of graphs.

11. MAXIS on the k-colorable Graphs

As was indicated in section 8, MAXIS can be used to good effect on k-colorable graphs. However, paradoxical effects are also found when this algorithm is combined with IG, and these effects are particularly noticeable on flat graphs.

For small k relative to the boundary $n/\mathcal{I}(n)$ MAXIS can successfully find the k-coloring alone given sufficient time. For example, with cutoff $(0, 6)$, on

$\mathcal{G}_{1000,\frac{1}{2},40}$ it found a 40-coloring 7 out of 10 times requiring 1607 seconds on average with standard deviation of 39.8. In the other three cases it found a 41-coloring. For larger k the results become more problematic, although in a full MAXIS search, that is one with unlimited backtracking, it is highly probable that the specified sets would be found since they are likely the largest, given that $k < n/\mathcal{I}(n)$.

However, it is generally much more effective to use a lighter search with MAXIS, and then allow IG to finish the coloring. For example, using $(0,3)$ on the same graphs, MAXIS found colorings in the range 42–49. IG used from 1 to 5 iterations to reduce these to 40-colorings, with an average total time of 75.1 seconds and standard deviation of 2.1. IG alone solved these in an average of 23.4 seconds. However, as k is increased the benefits of MAXIS increases, if MAXIS is allowed sufficient time.

For $k = 50$, MAXIS with $(0,3)$ found colorings in the range 64–77. IG used from 3 to 39 iterations to reduce these to 50-colorings, with total time averaging 81.4 seconds (deviation of 4.2). IG alone required 143.4 seconds(35.9 std).

For $k = 55$ the results become more interesting. Using MAXIS with $(0,3)$, colorings in the range 76–89 were obtained. IG was allowed 10000 iterations after the last improvement before failing. On nine of the ten graphs IG succeeded in reducing to a 55-coloring, using from 31 to 13846 iterations. In the remaining instance IG managed to reduce the 88-coloring handed to it by MAXIS to an 87-coloring in 5 iterations, but could not reduce this further in 11286 iterations. The switch was made to TABU which eventually reduced the coloring to 85. IG then took over and failed to reduce this further in over 13000 iterations. The nine successes averaged 222.0 seconds(217.5 std). In contrast pure IG which was also allowed 10000 iterations after the last improvement colored all ten graphs but required an average of 423.7 seconds(89.4 std).

By increasing the MAXIS search to $(0,4)$, MAXIS found 64–83 colorings, and IG made the reductions in under 20 iterations in all but one instance which required 457 iterations. The average time used was 367.7 seconds(25.6 std).

On $k = 60$, using $(0,4)$ and letting IG use 50000 iterations after the last improvement before failing, 8/10 graphs were successfully colored. Note that pure IG always succeeded with this setting. On the successes using MAXIS, from 207 to 50628 IG iterations were required, with an average of 25736(18702 std). This compares to pure IG which used an average 72793 iterations(22596 std).

Increasing the MAXIS search on $k = 60$ to $(0,5)$, all graphs were successfully colored, with IG using from 7 to 1406 iterations. The average total time was 1269 seconds(70 std). The average time for the pure IG approach was 4250 seconds(1327 std).

In summary, MAXIS is useful on equi-partite k-colorable graphs, but requires sufficient search time, or else the results may have negative impact on IG. If too much time is allocated to MAXIS, it may require more time than IG alone.

The problem is that on k-colorable graphs, if we do not know k in advance then the required tuning is not likely possible. The really interesting and as yet unanswered question is why using MAXIS with too little search time has a negative impact on IG on these graphs.

On flat graphs these negative effects are even more pronounced. In general, MAXIS seems unable to improve the search time on these graphs. All our results here are on 1000 vertex $p = \frac{1}{2}$ flat graphs with flatness zero.

For $k = 50$, MAXIS with $(0, 4)$ followed by IG fails. With cutoff $(0, 5)$ 4 out of 5 graphs were 50-colored, using from 147 to 12778 iterations of IG. With cutoff $(0, 6)$ all 5 graphs were successfully colored with IG using from 4 to 157 iterations. However, the average time was 2721 seconds(296 std), which is much larger than the average of 519 seconds(123 std) used by IG alone.

On $k = 55$ flat graphs using MAXIS with $(0, 6)$ failed to yield any color better than 85, even though IG was allowed 50000 iterations before failure. For $k = 60$, testing on the DIMACS graph with MAXIS using $(0, 8)$ also failed, providing only an 85-coloring. This graph was solved 5/5 times using pure IG and 50000 iterations before failure, using the same 70:50:30 heuristic ratio. In an earlier test, it was solved 5/7 times using the 50:50:30 heuristic ratio.

In [**5**] it was also observed that as the value of p decreased below 0.5 the tuning of MAXIS had to be more precise for any class of graphs. As p decreases the number and size of maximal independent sets increases, which makes it ever more difficult to use MAXIS effectively. For $p > 0.5$ MAXIS becomes more effective.

12. The Ridge

In the preceding sections we have discussed k-colorable graphs of various classes. We noted that as we increased k for fixed p and n the time to find the k-coloring increased apparently exponentially. This holds for IG, TABU or MAXIS and any mixture of them we have tried.

On the other hand, if k is made larger than the expected coloring obtained by greedy on a random graph $\mathcal{G}_{n,p}$, it is clear that even simple greedy will find a k-coloring or better with high probability. In between the small and large values of k, there are ranges where none of our algorithms find a k-coloring. As explained earlier, this region is near $k = n/\mathcal{I}(n)$.

Figure 10 shows the region where IG, even when working in combination with a number of other algorithms, fails to obtain the specified coloring on graphs with 300 vertices. For each combination of $.05 \le p \le 0.95$ in steps of 0.05 and for each k, $2 \le k \le 105$ (over the ranges indicated by the horizontal lines) ten graphs were generated. The black blobs indicate by height how many graphs were not colored with the specified number of colors. These tests were run on a variety of machines, ranging in speed from about 8 to 22 MIPS. Up to an hour was allowed in some cases. Checking the output, which includes measures of progress for TABU and IG, it appeared that no further progress was being made.

Above and to the left of the ridge IG is able to find the k-coloring unaided. To the right of the ridge, a coloring equal to the specified coloring is also easy to find, although in this case it is unlikely to be either optimal or to partition the vertices into the specified partition. In general, the further we are from the ridge, the easier it is to find the specified coloring.

For lower densities, the search was started with an iteration of DSATUR [**4**], then followed by a mixed IG and TABU [**9**] process. For higher densities, MAXIS was used for the initial coloring. Various points on the ridge with low densities were also retested using the MAXIS approach. All of these approaches failed consistently on the ridge. More detail can be found in [**5**].

The flat graphs also exhibit this ridge effect. The 300 vertex flat graphs submitted to DIMACS had hidden colorings of 20, 26 and 28. IG easily solves the 20-coloring instance, as shown in the appendix. It solved 2 out of 5 attempts on the 26-colorable graph, when 50000 iterations from the last improvement were allowed, using the 70:50:30 heuristic mix. It used 160700 iterations in one case and 175676 in the other. Starting with MAXIS using $(0, 4)$ one instance out of five was solved, when IG was allowed 50000 iterations.

However, no success was recorded on the 28-colorable instance. The preceding sections detail many attempts on the 1000 vertex graphs. No attempt was successful on the 76-colorable 1000 vertex flat graph.

Recent results [**1, 16**] in complexity indicate in the worst case no polynomial time algorithm can guarantee an approximation better than $O(n^\epsilon)$. Kučera [**14**] shows that the greedy algorithm will use $O(n/\log n)$ colors with high probability when $k = \lfloor n^\epsilon \rfloor$. This leaves us with the question of how well iterated algorithms may do in practice. It is perhaps fitting that we choose an iterated version of the greedy algorithm to explore this question.

Assuming that $n/\mathcal{I}(n)$ is a reasonable approximation to the boundary of the ridge, we let

$$k = k_r(n) = \frac{n}{r\mathcal{I}(n)}$$

for $r = 1.5, 2.0, 2.5, 3.0, 3.5$. We color equi-partite graphs for values of n up to 10000. The results are illustrated in Figure 11, where each plotted point is the median of ten runs. Various parameters to IG were tried, and a sample of typical results for $r = 3.0$ and $r = 3.5$ can be found in Table 14 and Table 15.

We see that as r is increased (decreasing k) the cost of coloring is reduced sharply. Perhaps not evident from the graph is the fact that even for $r = 3.0$ the number of iterations is approximately doubling for each increase of 1000 vertices and for $r = 3.5$ it doubles about every 1500 vertices. See Table 14 and Table 15. Thus, we have not escaped the exponential growth inherent in Kučera's result [**14**]. The 10000 vertex graphs required from 3 to 6 hours each[2].

[2]These tests were originally run on a machine other than the one we are using for the bench

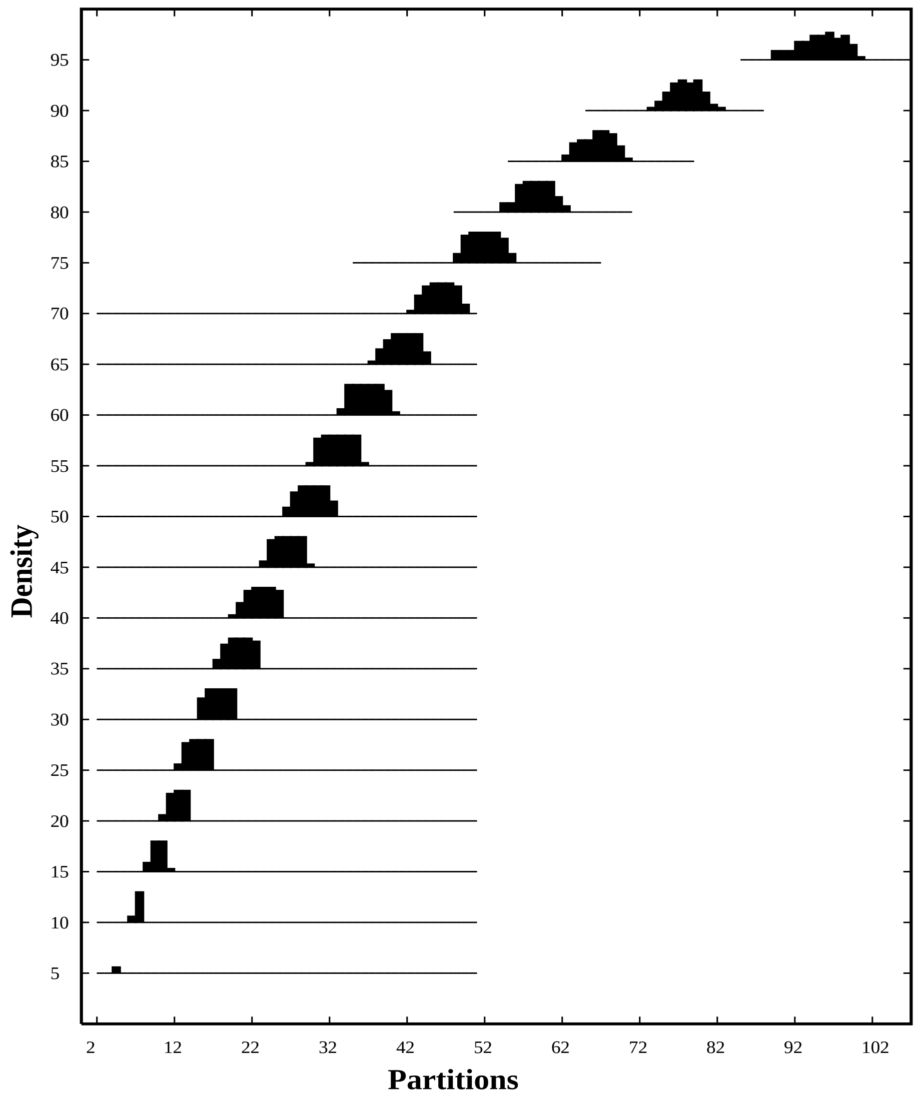


FIGURE 10. The Ridge of Missed Graphs on $\mathcal{G}_{300,\frac{1}{2},k}$

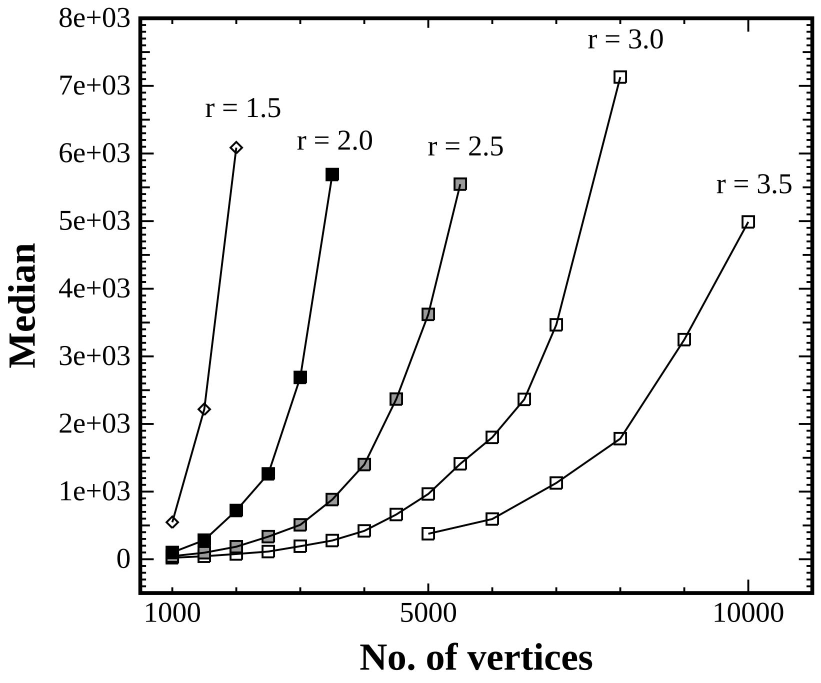


FIGURE 11. Difficulty of Coloring near the Ridge

n	$x = 90$		$x = 50$			$x = 40$	
	mean	deviation	mean	median	deviation	mean	deviation
1000	25.8	7.8	21.4	22.0	4.1	20.2	2.8
1500	52.4	8.6	44.7	44.0	7.4	38.2	5.8
2000	85.9	15.0	78.6	77.5	9.8	72.6	12.0
2500	136.3	21.1	109.7	113.5	15.6	120.5	16.2
3000	217.6	25.9	195.3	192.5	26.4	174.5	11.9
3500	339.9	59.4	278.1	276.5	39.2	270.9	46.3
4000	485.7	103.3	440.6	419.0	69.2	398.9	64.3
4500	775.9	130.8	623.1	661.5	96.3	575.2	90.9
5000	1093.2	148.6	967.0	965.0	166.5	855.3	83.5
5500			1380.1	1411.0	157.2		
6000			1866.0	1802.0	390.0		
6500			2379.1	2364.0	430.6		
7000			3383.3	3466.5	429.4		
8000			6827.6	7130.5	1283.8		

TABLE 14. Following the Boundary: $r = 3.0$, Parameter ratio (x:50:30)

n	(50:50:30)		
	mean	median	deviation
5000	371.5	376.5	58.0
6500	594.9	594.5	57.6
7000	1061.0	1127.5	88.5
8000	1820.9	1783.0	311.8
9000	3247.8	3247.5	619.2
10000	5026.3	4988.5	853.8

TABLE 15. Following the Boundary: $r = 3.5$

It seems that fixing k to be a constant induces an (almost) constant number of iterations. From Table 16 it can be seen that the rate of increase in iterations for $k = \sqrt{n}$ seems to be almost linear in n. If so, then this would show that Kučera's result does not extend to all values of ϵ for iterated algorithms. However, the range is too small, and the variability too great to draw firm conclusions. Ten trials were run at each n. Extending these observations, we wonder whether

		Iterations				Fraction Increase	
n	k	Min	Max	Mean	Dev	In n	Iterations
1024	32	63	130	99.4	20.2	–	–
2500	50	140	214	170.5	25.2	2.441	1.715
4900	70	216	306	259.2	35.1	1.960	1.520
10000	100	396	653	536.0	77.2	2.041	2.068

TABLE 16. Iteration Increase for $\mathcal{G}_{n,\frac{1}{2},k}$ with $k = n^{\frac{1}{2}}$

letting $r = r(n) = O(\log(n))$ might result in a non-exponential growth rate. If so, then for this class of graphs the chromatic number can be approximated on average within $O(\log n)$ in polynomial time. Unfortunately, it is unlikely that we will get any satisfactory answer to this question through empirical means, since the graphs required to make the distinction would be enormous.

Finally, referring back to Table 5 it seems likely that our ridge following approach would have more difficulty if we also allowed p to vary. With smaller p all of our algorithms face increased difficulty on approaching the ridge. Allowing both p and k to vary with n it seems likely there is a curve along which the growth rate is greater than shown in Figure 11.

13. Summary

The IG algorithm is robust over many classes of k-colorable graphs and over small perturbations in the parameters to the algorithm. Little performance im-

mark times. All times have been converted to what would be expected on our bench mark SUN SPARC 10 based on the running time of one 10000 vertex instance that was rerun on it provement seems to be obtained by tuning the parameters of the program for different regions of any class of graphs. Of course, knowing k does indicate when we can quit! For k-colorable graph classes IG also provides a machine independent measure of difficulty — the number of iterations until success. Finally, experiments with TABU and a semi-exhaustive maximal independent set search, plus other algorithms, all combined with IG, indicate that difficult graphs for one approach are difficult for all.

Thus, IG makes a good tool for an initial exploration of the complexity landscape of k-colorable graphs. Of course, it is not a universal algorithm. It does not do as well as the other approaches on $\mathcal{G}_{n,p}$. Using the independent set approach to find an initial coloring will often speed up the search, if the density is not too low [**5**]. In the appendix, we see that IG is not very successful on many of the graphs, such as Leighton graphs or the geometric graphs. Other programs often outperform IG on those graphs.

On equi-partite graphs using TABU almost always hinders IG. Using MAXIS in a too restricted mode has similar negative effects, especially on flat graphs. These paradoxical results are good candidates for further research.

In attempting to make graphs which are not on the ridge of difficulty, but which fool IG, we have tried varying the distribution of the partition elements. We also attempted to vary the probability distribution of edges between the partition elements. Neither variation created more difficult classes of graphs. Moving in the opposite direction, by reducing the variance in the degree sequence, the flat graph approach produces k-colorable graphs that seem to be harder to color. However, the boundary value of $k = n/\mathcal{I}(n)$ where IG fails seems to be the same as for equi-partite graphs, at least on 300 and 1000 vertex graphs. We wonder whether it is possible to substantially move this boundary.

References

1. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *33rd Annual Symposium on Foundations of Computer Science*, pp. 14–23. IEEE Computer Society Press, Los Alamitos, CA, 1992.
2. A. Blum. An $\tilde{O}(n^{0.4})$-approximation algorithm for 3-coloring (and improved approximation algorithms for k-coloring). In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pp. 535–542, 1989.
3. B. Bollobás and A. Thomason. Random graphs of small order. In M. Karoński and A. Ruciński (eds.), *Random Graphs '83*, volume 28 of *Annals of Discrete Mathematics*, pp. 47–97. North-Holland Publishing Co., Amsterdam, 1985.
4. D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4) pp. 251–256, April 1979.
5. J. C. Culberson. Iterated greedy graph coloring and the difficulty landscape. Technical Report TR 92-07, University of Alberta Department of Computing Science, Edmonton, Alberta, Canada T6G 2H1, 1992. ftp ftp.cs.ualberta.ca pub/TechReports.
6. J. A. Ellis and P. M. Lepolesa. A Las Vegas graph coloring algorithm. *The Computer Journal*, 32(5):474–476, 1989.
7. F. Glover. Tabu search–Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.
8. G. R. Grimmett and C. J. H. McDiarmid. On colouring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77:313–324, 1975.

9. A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.
10. D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
11. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May-June 1991.
12. A. Johri and D. W. Matula. Probabilistic bounds and heuristic 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.
13. L. Kučera. Graphs with small chromatic numbers are easy to color. *Information Processing Letters*, 30(5):233–236, 1989.
14. L. Kučera. The greedy coloring is a bad probabilistic algorithm. *Journal of Algorithms*, 12:674–684, 1991.
15. F. T. Leighton. A graph colouring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, 84(6):489–503, 1979.
16. C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, 1993.
17. B. Manvel. Extremely greedy coloring algorithms. In Frank Harary and John S. Maybee, editors, *Graphs and Applications (Proceedings of the First Colorado Symposium on Graph Theory, 1982)*, pp. 257–270. John Wiley & Sons, Inc., New York, 1985.
18. D. W. Matula. The employee party problem. *Notices of the American Math Society*, 19:A–382, 1972.
19. C. McDiarmid. Colouring random graphs badly. In *Graph theory and Combinatorics (Proc. Conf., Open Univ., Milton Keynes, 1978)*, volume 34 of *Research Notes in Mathematics*, pp. 76–86. Pitman, San Francisco, CA, 1979.
20. C. A. Morgenstern. *Algorithms for General Graph Coloring*. PhD thesis, Department of Computing Science, University of New Mexico, Albuquerque, New Mexico, 1989.
21. E. M. Palmer. *Graphical Evolution An Introduction to the Theory of Random Graphs*. Wiley-Interscience Series. John Wiley & Sons, Inc., 1985.
22. J. P. Spinrad and G. Vijayan. Worst case analysis of a graph coloring algorithm. *Discrete Applied Mathematics*, 12(1):89–92, 1985.
23. J. S. Turner. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

Second DIMACS Challenge

Coloring Benchmark Results

GENERAL INFORMATION Authors: Joseph C. Culberson and Feng Luo

Title: Exploring the k-colorable Landscape with Iterated Greedy.

Name of Algorithm: Iterated Greedy

Brief Description of Algorithm:

Heuristic: repeatedly applies the greedy algorithm to reorderings of the vertices that respect previous colorings.

Type of Machine: SUN SPARC 10/40
Compiler and flags used: GNU C: gcc -O2

MACHINE BENCHMARKS

User time for instances:

r100.5	r200.5	r300.5	r400.5	r500.5
0.03	0.92	7.99	49.68	192.60

ALGORITHM BENCHMARKS

Authors' Comments: The algorithm was set to look for a very small coloring number, which it could never find. It was instructed to terminate after 1000 iterations without improvement. Thus, there is no notion of "Fail"; or rather the program was forced to fail on every trial.

The parameter setting was the same for all graphs: reverse 50, largest first 50, random 30. The program started with a different random permutation for each of 10 runs. The initial coloring was obtained by greedy on that permutation. The program was not allowed to switch to TABU.

Better results can be obtained in some cases by careful tuning, or by extending the search time. For example, IG can optimally color 60-colorable flat graphs as described in the body of the paper.

In this table, the time is the total time including the time to read in an ASCII version of the DIMACS benchmark graph. Reading time for C4000 was 115 seconds, which agrees with the time to convert the file to binary using the program supplied by DIMACS (118 seconds). Reading time for C2000 was 28.5 seconds. It required less than 7 seconds to read DSJC1000.5.RES.

Results on Benchmark Instances

Name	Runs	Time Min	Time Avg. (Std. Dev.)	Time Max	Solution Min	Solution Avg (Std. Dev.)	Solution Max
`DSJC125.5.col`	10	2.90	5.62 (2.79)	12.20	18	18.90 (0.57)	20
`DSJC250.5.col`	10	7.00	15.31 (4.43)	20.00	32	32.80 (0.79)	34
`DSJC500.5.col`	10	24.00	40.91 (16.37)	74.20	57	58.60 (1.07)	60
`DSJC1000.5.col`	10	71.30	151.49 (47.19)	224.90	102	104.20 (1.23)	106
`C2000.5.col`	10	438.70	610.29 (190.62)	942.00	188	190.00 (1.15)	192
`C4000.5.col`	10	1058.90	2361.61 (821.99)	3544.00	345	346.90 (1.52)	350
`R125.1.col`	10	3.20	4.54 (1.29)	6.40	5	5.00 (0.00)	5
`R125.1c.col`	10	2.30	2.46 (0.11)	2.60	46	46.00 (0.00)	46
`R125.5.col`	10	2.70	4.23 (2.11)	9.20	36	36.90 (0.32)	37
`R250.1.col`	10	8.60	15.67 (7.60)	28.60	8	8.00 (0.00)	8
`R250.1c.col`	10	6.80	10.24 (3.63)	17.30	64	64.00 (0.00)	64
`R250.5.col`	10	9.60	18.60 (8.63)	34.90	68	68.40 (0.52)	69
`DSJR500.1.col`	10	28.80	47.35 (19.32)	81.60	12	12.00 (0.00)	12
`DSJR500.1c.col`	10	20.70	32.64 (7.16)	44.00	85	85.00 (0.00)	85
`DSJR500.5.col`	10	38.60	58.54 (14.31)	80.20	128	129.60 (0.84)	131
`R1000.1.col`	10	118.10	195.33 (63.62)	283.90	20	20.60 (0.52)	21
`R1000.1c.col`	10	61.80	109.04 (33.62)	166.20	98	98.80 (0.63)	100
`R1000.5.col`	10	157.60	230.43 (60.17)	338.50	252	253.20 (0.79)	254

Name	Runs	Time Min	Time Avg. (Std. Dev.)	Time Max	Solution Min	Solution Avg (Std. Dev.)	Solution Max
`flat300_20_0.col`	10	7.10	8.45 (0.79)	9.90	20	20.20 (0.63)	22
`flat300_26_0.col`	10	10.30	17.17 (6.31)	30.20	36	37.10 (0.74)	38
`flat300_28_0.col`	10	9.40	21.43 (7.60)	36.20	35	37.00 (0.94)	38
`flat1000_50_0.col`	10	82.40	327.81 (131.48)	500.00	50	65.60 (25.15)	104
`flat1000_60_0.col`	10	98.70	195.64 (65.03)	327.30	100	102.50 (1.35)	105
`flat1000_76_0.col`	10	124.30	178.30 (56.66)	311.60	102	103.60 (1.26)	106
`latin_square_10.col`	10	73.40	133.80 (46.79)	214.70	105	106.70 (1.42)	109
`le450_15a.col`	10	22.70	38.00 (16.70)	80.60	17	17.90 (0.32)	18
`le450_15b.col`	10	19.70	36.39 (10.73)	50.50	17	17.90 (0.32)	18
`le450_15c.col`	10	22.70	32.44 (8.73)	44.30	25	25.60 (0.52)	26
`le450_15d.col`	10	17.00	30.16 (12.26)	50.50	25	25.80 (0.42)	26
`mulsol.i.1.col`	10	6.80	9.34 (2.67)	15.90	49	49.00 (0.00)	49
`school1.col`	10	15.70	23.53 (5.57)	32.80	14	14.00 (0.00)	14
`school1_nsh.col`	10	14.30	20.01 (6.16)	33.40	14	14.10 (0.32)	15

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. T6G 2H1
E-mail address: joe@cs.ualberta.ca

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. T6G 2H1
E-mail address: feng@cs.ualberta.ca