Distributed Coloration Neighborhood Search

CRAIG MORGENSTERN

ABSTRACT. This paper describes and evaluates distributed versions of the impasse class and s-impasse class coloration neighborhood search algorithms. The main result is that maintaining a population of the best colorations seen so far and repeatedly attempting to improve them in parallel is faster and produces better colorations than doing independent concurrent search. The results of experimentation with a variety of graphs are presented to illustrate this result.

1. Introduction

A coloration neighborhood is an implicit mapping $\mathcal{F} : \Pi \rightarrow 2^{\Pi}$, where Π is the set of all colorations of a graph, $G = (V, E)$. Coloration neighborhood search (CNS) algorithms traverse a neighborhood structure in search of an improved coloration by repeatedly sampling the neighborhood of the current coloration as shown in Figure 1. These algorithms are computation intensive, but usually produce much better colorations than those obtained by doing multiple runs, for the same length of time, using randomized versions of the known polynomial time approximation algorithms. Acceptance of a neighboring coloration is determined using an objective function that prescribes a value to each coloration. Smaller objective function values correspond to better colorations. To avoid local optimum traps, one must allow for the possibility of accepting a neighboring coloration that represents a move contrary to the direction of optimality. A well known and successful mechanism used in simulated annealing involves generating a random number, $0 \leq r < 1$, and then accepting a disimprovement only if $r \leq e^{-\Delta/T}$ where Δ is the magnitude of the disimprovement and T is a control parameter called the temperature. Though simulated annealing requires a temperature reduction schedule, previous experience indicates that for some

1991 *Mathematics Subject Classification.* Primary 90C35, 68R10; Secondary 68R10.

```
C_0 := initial coloration;
repeat
    select a C ∈ F(C_0);
    if C is acceptable then C_0 := C
until termination criterion satisfied;
```

FIGURE 1. Generic CNS Algorithm

neighborhoods, equivalent or improved coloring results can be obtained by running at a fixed optimized temperature [**8, 9**] (though multiple short runs are required to determine this temperature for a class of graphs).

In this paper, the problem of distributing a coloration neighborhood search amongst several processors is examined, where the target system is one that is commonly available—a cluster of workstations. The most obvious approach is to perform concurrent runs of the algorithm given in Figure 1. That is, to concurrently follow independent paths in the neighborhood structure, where each (partial) coloration spawns and is replaced by a single new (partial) coloration. However, when doing multiple runs with a tight target number of allowed colors (using the impasse class neighborhood described below with the Metropolis acceptance criterion and a fixed temperature), it was observed that allowing each search to run for increasing periods of time did not significantly increase the number of successful searches. This indicated that if a search was not successful within a certain time limit, then the remaining time could be better spent by starting a new run. The implication for the design of a distributed coloration neighborhood search algorithm was that improved results might be obtained by maintaining a population of the best partial colorations seen so far. These partial colorations are farmed out to individual processors, which try to improve them within a parameterized number of iterations. That is, rather than starting a completely new search when a partial coloration cannot be improved within the time bound, a search is restarted from one of the existing good solutions in the population. On the other hand, if a partial coloration is improved, then the improvement is inserted into the population and the worse member of the population is removed. Thus, multiple dependent search paths are concurrently explored in the neighborhood structure, where several new paths can be spawned from a single given partial coloration.

The coarse grain parallelism of this distributed design makes it attractive for implementation on a workstation cluster using a manager-worker model. The algorithms tested were implemented in C and SR [**1**], and distributed runs were performed on five identically configured, 80 MHz, Sun Sparcstation-2 workstations (running almost the speed of a Sparcstation-10). Section 2 describes the neighborhood structures used, the algorithms are presented in Section 3, the experimental results are given in Section 4, and we conclude in Section 5.

2. The Coloration Neighborhoods

Modified versions of two coloration neighborhoods proposed in [**8, 9**] are utilized by the search algorithms. The *impasse class* neighborhood structure is used to try to improve a partial k coloration into a complete k coloration, and the *s-chain* neighborhood is used to jump to a new impasse class solution when it is determined that the impasse class process has become stuck.

The *impasse class* neighborhood requires that a target value, k, be provided for the number of colors to be used. A solution of the impasse class neighborhood is a partition of V into $k+1$ color classes, $V_0, \ldots, V_k$, in which all classes except possibly V_k are proper. V_k initially contains all vertices (an empty coloration) and the objective is to make V_k empty (a complete coloration) by doing a sequence of i-swaps. V_k is called the impasse class and contains vertices that are "at impasse" with the k proper classes. Given impasse vertex $v \in V_k$ and $0 \leq j < k$, let

$$U_{v,j} = \{w \mid w \in V_j \text{ and } (v,w) \in E(G)\}.$$

An i-swap operation involving $v \in V_k$ and V_j then performs the following steps:

(i) remove all $w \in U_{v,j}$ from V_j and add them to V_k, and

(ii) remove v from V_k and add it to V_j.

Rather than minimizing $|V_k|$, the objective used was to minimize the value

$$d_k = \sum_{v \in V_k} d(v)$$

where $d(v)$ is the vertex degree of v. This forces vertices of small degree into the impasse class, and these vertices are more easily colored. The Metropolis method is used to search the impasse class neighborhood structure, but with a low fixed temperature and (re)starting from initially good partial colorations. To avoid the performance penalty associated with a low acceptance rate, we refined the impasse class neighborhood to use rejectionless Metropolis move selection [**3**]. In the refinement, a move is selected with respect to a given temperature T as follows (see Figure 2):

(i) Choose a vertex $v \in V_k$ uniformly at random, and let

$$\Delta_j = \sum_{w \in U_{v,j}} d(w).$$

(ii) If $\Delta_j = 0$ for some j, then insert v into proper class V_j.

(iii) Otherwise, perform an i-swap with proper class V_j, where j is selected with probability

$$P_j = \frac{e^{-\Delta_j/T}}{e^{-\Delta_0/T} + \cdots + e^{-\Delta_{k-1}/T}}.$$

An additional efficiency gain was obtained by maintaining a cache of Δ_j values, one value for each (v, V_j) pair, and by determining $e^{-\Delta_j/T}$ by table lookup (since the Δ's are all integer valued and since T is fixed).

```
function i-swap_search(C, T, I, B) returns new_C;
    • T is the temperature.
    • I is the max number of allowed i-swaps.
    • Try to improve d_k = Σ_{v∈V_k} d(v) below B in coloration C.
begin
  while I > 0 and d_k ≥ B do begin
    I := I − 1;
    randomly select a v ∈ V_k;
    if v not in conflict with some proper class V_j then
      move v from V_k to V_j
    else
      perform i-swap with proper class V_j where V_j is
      selected with probability P_j
  end;
  return(C);
end;
```

FIGURE 2. Performing a sequence of i-swaps

The routine given in Figure 2 is the main search component for all the methods presented. It is passed a coloration and a bound B, and performs a sequence of i-swaps to produce a new coloration which it returns. The new coloration is the first one encountered that has $d_k < B$ (an improvement), or the one that results after I iterations (no improvement).

When a sequence of i-swaps fails to improve a partial coloration, it is possible to take advantage of the fact that classes $V_0, \ldots, V_{k-1}$ are all proper and nonempty. We can perform a sequence of nontotal s-chain interchanges on a subset of these classes in order to move to a new solution in the impasse class neighborhood that has the same d_k value as the old. An s-chain is an ordered $(s+1)$-tuple, $(v, W_0, \ldots, W_{s-1})$, where all the classes, W_i, are distinct, proper and nonempty, and W_0 contains v. This tuple represents the set of vertices reachable from v in the digraph, D, given by

$$\begin{aligned} V(D) &= W_0 \cup \cdots \cup W_{s-1} \\ A(D) &= \{(u,w) \mid (u,w) \in E(G),\ u \in W_i \text{ and } w \in W_{(i+1) \bmod s}\}. \end{aligned}$$

An s-chain interchange involves reassigning each chain vertex (a vertex reachable from v) in class W_i to class $W_{(i+1) \bmod s}$. An s-chain is said to be total when all vertices in $V(D)$ are reachable from v, since in this case the s-chain interchange is just a relabeling of color classes. For example, in Figure 3 the 3-chain given by (v, W_0, W_1, W_2) is total while the 3-chain given by (w, W_0, W_1, W_2) is not total. Several efficiency measures are mentioned in [**8, 9**] for finding nontotal s-chains. Finally, the number of vertices moved by an s-chain interchange is taken to be

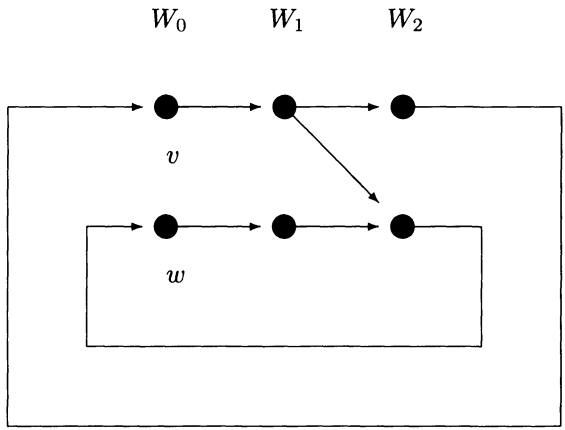

FIGURE 3. Example of 3-chaining

the minimum of the number of vertices reachable from v versus the number of vertices not reachable from v in D. This is because an interchange can also be performed by moving each nonchain vertex in class W_i to class $W_{(i-1) \bmod s}$.

3. The Search Algorithms

The coloration algorithms all utilize the routine shown in Figure 4. This routine attempts to improve coloration C by performing L alternating sequences of i-swaps and s-chain interchanges. Each i-swap sequence terminates after I consecutive i-swaps have been performed without improving the best solution seen so far. When this occurs, random s-chain interchanges are performed on the current solution until a total of D vertices have been moved (we guard against the unlikely case that all chains are total). The altered solution is then passed back to the i-swap search process, which again tries to improve the solution.

3.1. Sequential Algorithms. The CNS routine given in Figure 4 can be used to support several different types of searches. Making I large and setting $L = 1$ results in a search that does a single long sequence of i-swaps. Smaller I and $L > 1$ results in a search that alternates between sequences of i-swaps and sequences of s-chain interchanges. Finally, since the CNS routine returns the best solution found, we have the option of restarting the search anew from this best solution, called a *restart*. The resulting four combinations are invoked as follows:

no restarts, no chaining: $\boxed{C := \text{CNS}(C, T, \infty, 0, 1);}$

This algorithm is a pure i-swap search, and is denoted as S_0 (single run, no chaining).

```
function CNS(C, T, I, D, L) returns new_C;
        • C is the coloration to try to improve.
        • T is the temperature.
        • I is the max number of i-swaps/sequence that can be performed.
        • Perform s-chain interchanges until D vertices have been moved.
        • L is the number of i-swap and s-chain sequences to try.
        • d_k = Σ_{v∈V_k} d(v) in the current coloration.
begin
    best_C := C;   best_B := d_k;
    while L > 0 do begin
        L := L − 1;
        repeat
            improved := false;
            C := i-swap_search(C, T, I, best_B);
            if d_k < best_B then begin
                if d_k = 0 then return(C);
                best_C := C;   best_B := d_k;
                improved := true;
            end;
        until not improved;
        if L > 0 then C := s-chain(C, D);
    end;
    return(best_C);
end;
```

FIGURE 4. Performing sequences of i-swaps and s-chain interchanges

no restarts, chaining: $\boxed{C := \text{CNS}(C, T, I, D, \infty);}$
This algorithm will be denoted as S_1 (single run, chaining). It alternates between sequences of i-swaps and sequences of s-chain interchanges and never restarts. We set $D = |V|$ unless otherwise noted.

restarts, no chaining: $\boxed{\text{repeat } C := \text{CNS}(C, T, I, 0, 1) \text{ until done;}}$
Denoted as R_0 (restarts from best solution only, no chaining), this algorithm restarts the i-swap search from the current best solution after an i-swap sequence fails to find an improvement.

restarts, chaining: $\boxed{\text{repeat } C := \text{CNS}(C, T, I, D, L) \text{ until done;}}$
This is algorithm R_1 (restarts from best solution only, chaining). We set $D = |V|$ and $L = 2$ unless otherwise noted. Thus, we do i-swap sequences until no improvement is made, perform an s-chain interchange sequence and then try once more to (further) improve with i-swap sequences. This process is then restarted from the current best solution.

3.2. Distributed Algorithms. The distributed search algorithms use a manager-worker model. The manager maintains a pool of the best P colorations found by the workers, and a single search process is associated with each worker. Initially, the pool contains empty colorations. The workers each request a coloration from the manager. When a worker receives a coloration to try to improve, it invokes the CNS search routine of Figure 4 (and so performs a restart on this coloration). If an improved coloration results, then the improvement is passed back to the manager for insertion into the pool. In any case, a new coloration is requested from the manager by the worker after the worker's CNS routine returns. The pool scheme allows for multiple dependent search paths through the neighborhood since several restarts may be performed on the same coloration resulting in several improved colorations. That is, new colorations are generated by selecting a "parent" coloration from the pool and then passing a copy of the parent to the requesting search process. The parent coloration remains in the pool, and if the search process produces an improved "child" coloration, then the child is inserted into the pool. Only the current best P colorations are in the pool (can produce children) and a parent coloration may produce several children before being forced from the pool. There are two versions of the distributed algorithm, since restarts are always performed:

no chaining: The workers invoke $C := \mathrm{CNS}(C, T, I, 0, 1)$ which does i-swaps till a sequence of I of them have been performed without improvement. This is the distributed counterpart to R_0 and is denoted by M_0 (many parallel restarts from a pool of solutions).

chaining: The workers invoke $C := \mathrm{CNS}(C, T, I, D, L)$ where $D = |V|$ and $L = 2$ unless otherwise noted. This is the distributed counterpart to R_1 and is denoted as M_1 (many parallel restarts from a pool of solutions with chaining allowed).

Other implementation details are:

- The pool is kept sorted by decreasing d_k value; the coloration that has the smallest degree sum taken over the impasse class vertices is at the top of the pool. After each insertion of a coloration into the pool, the member with largest d_k value is removed.
- An attempt is made to avoid inserting a coloration into the pool that is isomorphic to an existing pool coloration. The manager uses a simple $O(V + k^2 \log k)$ greedy algorithm to check coloration isomorphism—the algorithm is always correct if it answers yes, but may not be correct if it answers no.
- Selection of colorations from the pool proceeds in a round-robin fashion. It is possible that a random or biased selection of pool colorations could give better results, but we were unable to experiment with this idea.
- The workers use their own random number streams by dividing up a random number sequence as described in [**6**, p. 250]. The period of the random number sequence used is 2^{32}, and every 2^{20} value was saved in a

static table. The master is given a position in the sequence at run time. The worker's initial seeds are then determined relative to this position (utilizing the static table) so that the sequence is divided evenly amongst the workers.

- Color classes are represented as linked lists and the graph is represented with an adjacency matrix. Each worker has its own copy of the graph.

3.3. Hybrid XRLF Algorithms. The results of Johnson *et al.* [**4**], Bollobás and Thomason [**2**], and Morgenstern [**8, 9**] demonstrate that the known coloration neighborhood search methods are outperformed by independent set selection algorithms on random graphs with edge density near 1/2. On these graphs, methods such as XRLF (Johnson *et al.* [**4**]) or Bollobás and Thomason's [**2**] semi-exhaustive search are the most competitive. For example, coloring $G_{1000,0.5}$ graphs with a tuned distributed CNS search on five Sparcstation-2's for 24 hours can occasionally produce an 88 coloration. Independent set selection methods can consistently achieve colorations in the range of 86 to 87 colors on a single machine in under an hour. However, as noted in [**8, 9**], coloration neighborhood search is effective in improving the colorings produced by independent set selection methods. A distributed implementation of the XRLF algorithm was used as an alternative method to initially populating the pool (instead of populating with empty colorations). For a given target value k, the workers use distributed XRLF to generate k disjoint, proper classes. All vertices not assigned to a proper class are then placed into the impasse class, and the resulting impasse class neighborhood solution is inserted into the pool. This is repeated until the pool contains the desired number of XRLF generated initial solutions.

4. Experimental Results

In addition to the DIMACS benchmark instances, the algorithms were tested on the four 25 chromatic Leighton graphs [**7**] and the following additional random graphs used in Johnson *et al.* [**4**] study: $G_{500,0.1}$, $G_{1000,0.1}$, $G_{500,0.9}$, $G_{1000,0.9}$, and the B, D, E, and F $G_{1000,0.5}$ instances.

The time values that are in cpu seconds do not include communication time. Communication time includes the time required to send a copy of the graph to each worker, and could only be measured as a component of wall clock time. Total cpu time is given in seconds, and is the sum of each worker's cpu time and the manager's cpu time. It is an accurate measure of the time that would be required by a sequential simulation of the distributed algorithms. Wall clock time is also given and care was taken to perform the runs when the systems were lightly loaded. However, since the runs were performed in a multi-user environment, wall clock time gives an upper bound on the actual running time required by the distributed algorithms.

In addition to cpu time, the number of color class scans required by the i-swap_search routine is given as a measure of efficiency. Color classes are main-

		$(P=1)$		$P=5$		$P=15$		$P=25$	
I	S_1	R_0	R_1	M_0	M_1	M_0	M_1	M_0	M_1
6250	10	0	0	0	0	1	3	1	4
12500	10	3	0	0	2	4	7	5	8
25000	10	1	3	1	7	6	10	10	10
50000	10	5	5	5	9	10	10	10	10
100000	10	5	6	10	10	10	10	10	10
200000	10	8	9	10	10	10	10	10	10

TABLE 1. Number of successful 28-colorings

tained as linked lists and class lists are scanned to (re)compute a Δ_j value (when the value in the cache is incorrect) and to perform an i-swap.

The algorithms tested were implemented in C and SR [**1**]. The distributed runs were performed with five worker processes, each running on a 80 MHz Sun Sparcstation-2 workstation. The manager process shared a machine with a worker process. Because of the coarse grain parallelism of the distributed algorithms, the manager was almost always idle—the manager required on the order of a few seconds of cpu time per worker's hour of cpu time.

4.1. Effects of Varying I and P on a $G_{250,0.5}$ Graph. The $G_{250,0.5}$ random graph of the DIMACS benchmark suite (graph `DSJC250_5.col`) was used in the Johnson *et al.* [**4**] study. Random $G_{250,0.5}$ graphs have an expected chromatic number of 27. The results of attempting to 28-color this graph are given to illustrate the effects of altering I (largest i-swap sequence allowed) and P (pool size) with respect to a fixed number of colors and temperature. The temperature for all the runs on this graph was fixed at an optimized value of 35 using the technique described in § 4.2. Algorithm S_0 was run 50 times and successfully 28-colored the graph each time. Algorithms R_0 and R_1 approximate a pool of size 1 since they always restart from the best solution and not from any other solution. Table 1 shows the number of successful 28-colorings out of 10 attempts per I and P value. The R_0, R_1, M_0 and M_1 runs terminated with failure when 64 consecutive restarts were performed without an improvement. Table 1 shows that a smaller pool size requires a larger maximum i-swap sequence size, and that restarting from the best solution only is not effective with the (smaller) values of I that work for the distributed algorithms.

Table 2 shows the efficiency of the algorithms for selected parameter values which exhibited no 28-coloring failures. As I increases, the efficiency of S_1 approaches that of S_0. With small I, S_1 performs more s-chain searches and interchanges. Both M_1 and S_1 must also flush the Δ_j cache after each sequence of s-chain interchanges, which increases the number of class scans. For this graph, there does not appear to be a need to use chaining to escape locally optimal traps, so the increased overhead associated with chaining does not pay for itself. On the other hand, Table 2 shows that a sequential simulation of M_0

			total class scans		total cpu time	
	I	P	mean	std dev	mean	std dev
S_0	—	—	1.8×10^8	1.3×10^8	988	699
S_1	6250	—	7.6×10^8	5.5×10^8	4312	3146
	200000	—	2.2×10^8	1.6×10^8	1201	830
M_0	100000	5	6.5×10^7	3.2×10^7	406	244
	200000	5	1.1×10^8	9.7×10^7	652	643
	50000	15	1.0×10^8	1.8×10^7	591	137
	200000	15	1.6×10^8	1.1×10^8	878	657
	25000	25	1.3×10^8	4.2×10^7	865	366
M_1	100000	5	1.2×10^8	5.5×10^7	675	336
	200000	5	1.6×10^8	1.1×10^8	919	700
	25000	15	1.0×10^8	4.0×10^7	659	313
	200000	15	3.0×10^8	2.3×10^8	1696	1361

TABLE 2. Successful 28-coloring efficiency

will outperform S_0 over a wide range of parameters. The smallest value of I which exhibits no failures for a given P gives the best results in terms of average speed and stability.

The S_0 runs were performed in 10 groups of 5 runs each to simulate 10 runs of 5 concurrent independent searches of the neighborhood. The running time of each set of concurrent runs is then the minimum time required by each of the 5 independent searches. The mean of the 10 minimum cpu times was 238 seconds, with a standard deviation of 194. Even with this view of the data, M_0 still outperformed S_0. With $I = 100000$ and $P = 5$, the mean M_0 wall clock time was 113 seconds with a standard deviation of 51. For $I = 50000$ and $P = 15$, the mean M_0 wall clock time was 146 seconds with a standard deviation of 27. Not only is M_0 up to twice as fast as independent, concurrent S_0 search on average, but it is considerably more stable.

Not surprisingly, this example indicates that for a given P, the best M_0 and M_1 results are obtained by making I as small as possible while still avoiding failures. On the other hand, if I is too large then the number of failures does not increase; only the running time and variance increases.

4.2. Effects of Varying T and k on $G_{500,p}$ Graphs. The $G_{500,0.1}$, $G_{500,0.5}$, and $G_{500,0.9}$ random graphs are all from the Johnson *et al.* [**4**] study and have expected chromatic numbers of 11, 46, and 122 respectively. The $G_{500,0.5}$ graph is part of the DIMACS benchmark suite and is identified as graph `DSJC500_5.col`. These graphs are used to illustrate a procedure for determining an optimized temperature, T, and a coloration target value, k. Since rejectionless Metropolis move selection [**3**] was applied, T was not determined by specifying an acceptance rate (see Laarhoven [**5**, p. 32]). Our best results were obtained using a

k	T	k	T	k	T	k	T	k	T
≥ 65	2	62	14	59	24	56	30	53	54
64	4	61	16	58	26	55	38	52	64
63	4	60	24	57	28	54	50		

TABLE 3. Determining a crude k and T for $G_{500,0.5}$

temperature that corresponds to an extremely low acceptance rate under normal Metropolis move selection. This temperature was determined by a procedure that is based on the following heuristics.

HEURISTIC 4.1. *Let $k_0 < k_1$, and C_k be the empty impasse class solution with k proper classes. For reasonable values of I and T, on average, the impasse class neighborhood colorations produced by CNS$(C_{k_0}, T, I, 0, 1)$ and CNS$(C_{k_1}, T, I, 0, 1)$ have $d_{k_0} \geq d_{k_1}$.*

Reasonable values of I and T means that T should be small enough and I large enough so that the impasse class size, d_k, reaches quasi-equilibrium [**5**, p. 30]. This heuristic is intuitive since C_{k_1} has at least one more proper class than C_{k_0}. Thus, for fixed I and T, the quasi-equilibrium colorations produced from C_{k_1} should have fewer vertices in the impasse class than the ones produced from C_{k_0}. Heuristic 4.2 follows from Heuristic 4.1, and it means that the optimal temperature range for a large enough k contains the optimal temperature range for targets smaller than k.

HEURISTIC 4.2. *Let $L_k < T < H_k$ be the temperature range such that, on average, CNS$(C_k, T, I, 0, 1)$ produces impasse class neighborhood solutions with $d_k \leq b$ for a fixed I and fixed bound b. Then on average, for $k_0 < k_1$, $L_{k_1} \leq L_{k_0}$ and $H_{k_0} \leq H_{k_1}$.*

The procedure for determining a target k and associated optimized T proceeds in two parts. The first part is concerned with quickly getting crude estimates. Set $I \approx 10^5$ and start with $T = 2$ (this worked well for the graphs we tested). The initial k should be large enough so that k-colorations are "easily" obtained. Such an initial k can be determined by using the greedy coloring algorithm if little is known about the graph. Raise T by increments of t until CNS$(C_k, T, I, 0, 1)$ consistently produces complete k-colorations ($b = 0$), then using this value of T, decrease k by 1 and repeat. Table 3 shows the estimates obtained with $t = 2$ and $I = 100000$ on the $G_{500,0.5}$ graph. For each value of k, the T given is the smallest one for which CNS$(C_k, T, I, 0, 1)$ produced 10 complete k-colorations out of 10 attempts. Eventually, k will become so small that complete colorations are not produced. This condition is identifiable when increasing T results in CNS producing colorations whose average d_k also increases (as illustrated in Table 4 when T increases beyond 64).

T	60	62	64	66	68	70
$k = 51$	609	285	242	453	1148	1540
$k = 50$	1792	1966	1904	4353	4211	5405
$k = 49$	5691	5732	7429	7962	8530	8921

TABLE 4. $G_{500.0.5}$ d_k chart for optimizing T

Average d_k over 10 runs per T and k						
	$I = 10^5$			$I = 10^6$		
T	$k = 130$	$k = 131$	$k = 132$	$k = 128$	$k = 129$	$k = 130$
60	661.2	131.8	75.7	266.5	218.3	0.0
76	397.3	130.9	0.0	176.4	44.6	0.0
86	353.1	0.0	0.0	1023.3	44.2	0.0
96	1617.7	581.5	0.0	3145.5	1468.9	176.0
Average cpu time over 10 runs per T and k						
76	38.1	22.2	12.9	279.6	108.0	40.2
86	33.4	26.6	13.4	373.7	148.0	57.4
96	46.3	37.2	20.1	340.2	410.8	196.3

TABLE 5. $G_{500,0.9}$ d_k chart

Table 4 contains the average d_k value over 10 CNS runs per T and k value, with $I = 100000$. This table represents the effort needed for the second part of the temperature selection procedure. For values of k just below the estimated target k determined in first part, tally the average d_k values produced by $\text{CNS}(C_k, T, I, 0, 1)$. Use values of T centered on the estimated target temperature. Then when increasing T, d_k will undergo a relatively large jump. The optimized temperature for these values of k and the given I is one that is slightly smaller than the temperature at which the jump occurs. Thus, Table 4 shows that $T = 63$ should be used to 49-color the $G_{500,0.5}$ graph with $I \approx 100000$. Tables 5, 6, and 7 also illustrate the temperature setting procedure and the relatively large jump that d_k undergoes when increasing T. This jump is seen in Table 7 for $k = 12$ as T goes from 22 to 24, in Table 6 for $k = 50$ as T goes from 64 to 68, and for several values of k in Table 5. Additionally, the following trends have been observed and are indicated by these tables:

(i) Smaller values of I result in larger target k values.

(ii) Small I and large t values give a quick estimate of k and T. These estimates can then be improved by increasing I and decreasing t.

(iii) The closer k is to the chromatic number of the graph, the more exact T must be. A larger temperature increment, t, was used when determining the temperature for dense graphs than was used for the sparse graphs. This is because we were able to color closer to the expected chromatic number of the sparse graphs than that of the dense graphs.

Average d_k over 10 runs per T and k						
	$I = 10^4$			$I = 10^6$		
T	$k = 53$	$k = 54$	$k = 55$	$k = 50$	$k = 51$	$k = 52$
56	415.3	92.1	24.8	477.4	171.9	45.9
60	503.6	0.0	0.0	380.5	24.4	0.0
64	259.7	0.0	0.0	531.4	0.0	0.0
68	93.2	46.2	0.0	1690.1	0.0	0.0
72	1050.8	269.8	0.0	3797.6	432.1	0.0
76	1064.0	221.2	0.0	6197.7	2663.6	125.6
Average cpu time over 10 runs per T and k						
56	4.3	2.6	1.7	180.9	133.3	63.4
64	3.4	2.5	1.3	307.1	51.7	14.4
76	5.5	3.8	2.1	446.0	315.9	175.4

TABLE 6. $G_{500,0.5}$ d_k chart

Average d_k over 10 runs per T and k						
	$I = 10^4$			$I = 10^6$		
T	$k = 12$	$k = 13$	$k = 14$	$k = 12$	$k = 13$	$k = 14$
10	810.9	152.4	4.8	436.9	3.6	0.0
18	605.6	4.6	0.0	164.1	0.0	0.0
20	841.7	0.0	0.0	108.5	0.0	0.0
22	1424.7	42.1	0.0	260.9	0.0	0.0
24	2256.8	73.6	0.0	1143.1	0.0	0.0
30	3301.9	1408.3	39.1	2875.7	556.8	0.0
Average cpu time over 10 runs per T and k						
10	4.3	2.0	0.4	195.6	26.1	0.4
20	4.9	1.5	0.3	215.4	1.6	0.3
30	2.9	3.0	1.3	215.1	273.0	1.3

TABLE 7. $G_{500,0.1}$ d_k chart

T	S_0	S_1	R_0	R_1	M_0	M_1
20	3	1	0	0	3	8
22	2	0	0	0	8	10
24	0	0	0	0	1	3

TABLE 8. Number of successful 12-colorings of $G_{500,0.1}$

		total cpu time		wall clock time		total class scans	
T	49-col	mean	std dev	mean	std dev	mean	std dev
61	8	24097	7831	5643	1923	3.3×10^9	6.6×10^8
63	10	17852	6031	3724	1201	3.1×10^9	9.7×10^8
65	9	28283	12004	6730	3458	4.9×10^9	1.7×10^9

TABLE 9. 49-coloring $G_{500,0.5}$ with M_0

4.3. $G_{n,p}$ **Graph Results.** The results of coloring random graphs of the following types are presented: $G_{500,0.1}$, $G_{500,0.5}$, $G_{500,0.9}$, $G_{1000,0.1}$, $G_{1000,0.5}$, $G_{1000,0.9}$, and $G_{2000,0.5}$. The $G_{n,0.5}$ graphs are part of the DIMACS benchmark suite, and all but the $G_{2000,0.5}$ graph are taken from the Johnson *et al.* [4] study. A pool size of $P = 15$ was used for all the M_0 and M_1 runs. The S_1, R_1, and M_1 runs all used $D = |V|$ and $L = 2$. R_0, R_1, M_0 and M_1 all terminated with failure when 64 consecutive restarts were attempted without any improvement. S_0 and S_1 terminated with failure when the number of class scans exceeded a bound.

4.3.1. $G_{500,0.1}$ *coloring results.* This graph has an expected chromatic number of 11. As indicated by Table 7, 13-colorings are easily obtainable, and the optimal temperature for $k = 12$ is in the range $20 \leq T < 24$ for $10^4 \leq I \leq 10^6$. Table 8 gives the number of successful 12-colorings obtained out of 10 attempts per temperature and method with $I = 10^5$. S_0 and S_1 failed when more than 4×10^9 class scans were performed, which corresponded to 10 hours of cpu time. The 10 successful M_1 runs at $T = 22$ had the following statistics: the mean total cpu time was 5452 seconds with a standard deviation of 3087, the mean wall clock time was 1153 seconds with a standard deviation of 624, and the mean number of color class scans was 5.4×10^8 with a standard deviation of 2.9×10^8.

4.3.2. $G_{500,0.5}$ *coloring results.* This graph has an expected chromatic number of 46. As indicated by Table 6, 51-colorings are easily obtainable, and the optimal temperature for $k = 50$ is in the range $60 \leq T < 68$ for $10^4 \leq I \leq 10^6$. In fact with $T = 65$ and $I = 10^5$, each of the methods successfully 50-colored the graph 10 times out of 10 attempts. There was no speed advantage in using the distributed methods to 50-color the graph. In the `49-col` column, Table 9 gives the number of successful 49-colorings produced by M_0 out of 10 attempts per temperature with $I = 10^5$. The cpu times and class scans are over the successful

T	S_0	S_1	R_0	R_1	M_0	M_1
75	7	4	8	7	10	10
80	4	3	6	6	10	10
85	0	0	1	3	10	10

TABLE 10. Number of successful 127-colorings of $G_{500,0.9}$

		total cpu time		wall clock time		total class scans	
	T	mean	std dev	mean	std dev	mean	std dev
R_0	75	2248	1048	2279	1043	2.7×10^8	1.4×10^7
S_0	75	5399	4333	5473	4364	7.1×10^8	5.8×10^8
M_0	75	9071	1482	2024	274	1.2×10^9	1.8×10^8
	80	12578	5866	2734	1170	1.8×10^9	8.1×10^8
	85	20826	7359	4265	1257	3.1×10^9	9.7×10^8

TABLE 11. 127-coloring $G_{500,0.9}$ with M_0, S_0 and R_0

runs, while the wall clock values are taken over all runs. As for the $G_{250.5}$ graph of § 4.1, there was no advantage to performing chaining with M_1. S_0 and S_1 were each run 5 times for 37 hours and 58 hours of cpu time respectively without successfully 49-coloring the graph (with $T = 63$). R_0 and R_1 were each run 10 times, and each run terminated with failure after 64 consecutive restarts occurred without any improvement.

4.3.3. $G_{500,0.9}$ *coloring results.* This graph has an expected chromatic number of 122. As indicated by Table 5, 130-colorings are easily obtainable, and the optimal temperature for $k < 130$ is roughly in the range $75 \leq T \leq 85$ for $10^5 \leq I \leq 10^6$. Table 10 gives the number of successful 127-colorings obtained out of 10 attempts per temperature and method with $I = 5 \times 10^5$. S_0 and S_1 failed when more than 4×10^9 class scans were performed. Table 11 contains the running times and class scans needed by M_0 to 127-color the graph. For $T = 75$, the successful S_0 and R_0 run statistics are also included in the table. With $T = 75$ and $I = 10^6$, M_0 successfully 126-colored this graph 4 times out of 10 attempts. The successful runs required an average of 159872 seconds of total cpu time with a standard deviation of 81915, and the average wall clock time over all 10 runs was 36008 seconds with a standard deviation of 12207.

4.3.4. $G_{1000,0.1}$ *coloring results.* The $G_{1000,0.1}$ graph has an expected chromatic number of 19. All the methods can quickly 21 color this graph (the longest run required 420 seconds cpu time) with $I = 10^5$ and $31 \leq T \leq 38$. The optimal temperature for $k = 20$ and $10^5 \leq I \leq 10^6$ is $T = 35$. All attempts to 20 color this graph were unsuccessful. We attempted 5 M_1 runs with $P = 60$, $I = 10^6$ and $T = 35$, each for over 48 hours of wall clock time. The best result was 5 vertices left in the impasse class when the run was terminated.

	total cpu time		wall clock time		total class scans	
k	mean	std dev	mean	std dev	mean	std dev
230	3971	608	929	132	4.5×10^8	6.4×10^7
228	8870	1700	1933	329	9.0×10^8	1.6×10^8
226	65774	10135	13531	2053	6.9×10^9	6.3×10^9

TABLE 12. Coloring $G_{1000,0.9}$ with M_0

4.3.5. $G_{1000,0.5}$ *coloring results.* The $G_{1000,0.5}$ graph has an expected chromatic number of 80. All methods can consistently 91 color this graph with $T = 120$ and $I = 10^5$. The sequential methods start to fail when 90 colorings are attempted, and only the distributed methods were successful in 89 coloring the graph. M_0 required an average of 91336 seconds total cpu time (standard deviation of 3184) over 10 successful runs to 89 color the graph. M_0 was successful in one run out of 10 attempts to prescribe an 88 coloration, each run requiring over 24 hours of wall clock time. All the $k = 89$ and $k = 88$ M_0 runs used $T = 120$ and $I = 10^6$.

4.3.6. $G_{1000,0.9}$ *coloring results.* This graph has an expected chromatic number of 217. For $226 \leq k \leq 230$ and $10^5 \leq I \leq 10^6$, $T = 140$ is an optimized temperature. Table 12 gives the M_0 coloring results with $T = 140$ and $I = 10^5$ for $k = 226$, $k = 228$ and $k = 230$. M_0 was successful 10 times out of 10 attempts for each value of k.

4.3.7. $G_{2000,0.5}$ *coloring results.* This graph has an expected chromatic number of 142. For $10^5 \leq I \leq 10^6$ and $k \approx 165$, $T = 220$ is roughly optimal. Since the trend indicated from the runs on the smaller $G_{n,0.5}$ graphs is that M_0 has the best performance, only M_0 was tested on this graph (with $I = 250000$). M_0 required an average of 973 seconds wall clock time to 167 color the graph and an average of 6750 seconds wall clock time to prescribe a 165 coloring. All averages were over 5 runs.

4.4. XRLF Hybrid Results. A distributed implementation of Johnson's XRLF algorithm [4], denoted by dXRLF, was used to generate a pool population of good initial solutions. Then M_0 was applied to these colorations to obtain solutions that are better than M_0 or XRLF can produce individually. The resulting combination is denoted as M_0/dXRLF. The dXRLF parameters (*setlim*, *candnum*, and *trialnum*) have exactly the same meanings as they do in [4]. Johnson *et al.* fixed *setlim* and *candnum* to 63 and 50 respectively and varied *trialnum*. In this study, both *candnum* and *trialnum* are varied and *setlim* remains fixed at 63. The dXRLF implementation divides *trialnum* iterations evenly amongst the workers and does not perform an exhaustive exact-coloring search of the final 70 vertices (as is done by Johnson *et al.* [4]).

k	trialnum	candnum	$\overline{\lvert V_k \rvert}$	mean total cpu time dXRLF	mean total cpu time M_0/dXRLF	wall clock time mean	wall clock time std dev
	100	50	8.66	603	710	195	47
49	50	100	8.23	1152	1207	304	16
48	300	250	11.44	11812	37259	8144	2373

TABLE 13. Coloring $G_{500,0.5}$ with M_0/dXRLF

Graph B			Graph C			Graph D		
	wall time			wall time			wall time	
$\lvert V_k \rvert$	dXRLF	M_0	$\lvert V_k \rvert$	dXRLF	M_0	$\lvert V_k \rvert$	dXRLF	M_0
7	379	238	9	377	61	7	382	27
8	381	1257	9	380	948	9	410	643
9	385	123	10	372	1232	10	370	1652
9	416	19	12	366	2133	10	381	(1741)
10	419	1600	12	376	14	11	476	(4488)

TABLE 14. 85-coloring $G_{1000,0.5}$ with M_0/dXRLF

4.4.1. $G_{500,0.5}$ *coloring results.* In [**4**], XRLF with a final exact-coloring search produces 50 colorations of this graph. dXRLF requires ≥ 51 colors over a wide range of parameters. Table 13 contains the results of applying the M_0/dXRLF hybrid algorithm to this graph. Each $k = 49$ row contains the statistics of 10 successful runs out of 10 attempts. Out of 10 attempts to 48-color the graph, 8 were successful. The cpu time statistics given are over the 8 successful runs, and the remaining statistics (including wall clock time) are over all 10 runs. M_0 ran with $T = 63$ and $P = 15$ in all cases. The 49-coloring runs populated the pool with 3 dXRLF solutions and then ran M_0 with $I = 250000$. The 48-coloring runs populated the pool with 10 dXRLF solutions and then ran M_0 with $I = 500000$. M_0/dXRLF is an order of magnitude faster than M_0 in finding 49-colorings for this graph. The 48-colorings were possible only after experimentation showed that M_0 was most effective in improving on dXRLF solutions when *candnum* was increased. We were unable to 48-color this graph when leaving *candnum* fixed at 50 even though *trialnum* was raised as high as 2500.

4.4.2. $G_{1000,0.5}$ *coloring results.* In [**4**], XRLF with a final exact-coloring found 86 colorations on the C graph (the DIMACS benchmark $G_{1000,0.5}$ graph), while dXRLF required ≥ 88 colors over 5 trials. Also reported in [**4**], XRLF used either 85 or 86 colors for the other four $G_{1000,0.5}$ graphs that were tested. On these four graphs, dXRLF needed $87 \leq k \leq 89$ colors over 5 trials per graph. Table 14 indicates the effort needed to 85-color the B, C, and D graphs using M_0/dXRLF. The pool was populated with a single dXRLF solution, and M_0 was run with $T = 120$, $I = 150000$ and $P = 15$. The dXRLF parameters used were the same as the XRLF parameters given in [**4**] (a *setlim* of 63, a *candnum*

graph	$\overline{\lvert V_k \rvert}$	mean total cpu time dXRLF	mean total cpu time M_0/dXRLF	wall clock time mean	wall clock time std dev
B	8.90	33605	36196	8078	1129
C	10.18	32935	85658	18235	12529
D	10.86	32990	132352	27706	13777
E	10.40	32930	70558	15117	11507
F	8.94	33315	34118	7459	702

TABLE 15. 84-coloring $G_{1000,0.5}$ with M_0/dXRLF

of 50, and a *trialnum* of 1260). On the D instance, M_0 twice failed to improve the dXRLF solution to a complete 85-coloring; the times for these failed runs are given in ()'s. Table 15 shows the effort required to 84-color all 5 graphs using M_0/dXRLF. The statistics are taken over 10 successful runs out of 10 attempts per graph. The pool was populated with 5 dXRLF solutions generated with a *setlim* of 63, a *candnum* of 500, and a *trialnum* of 300. Again, dXRLF experimentation indicated that an increased *candnum* was more important than a large *trialnum* for the hybrid algorithm. M_0 was applied with $T = 120$, $I = 10^6$, and $P = 15$.

4.4.3. $G_{2000,0.5}$ *coloring results.* Using a *setlim* of 63, a *candnum* of 1000, and a *trialnum* of 250, dXRLF was able to find a 152 coloration 8 times in 10 attempts, requiring an average of 40874 seconds of total cpu time. The M_0/dXRLF approach was able to find a 150 coloration 10 times out of 10 attempts when the pool was populated with a single dXRLF solution (using the same parameters as the $k = 152$ XRLF runs). M_0 was applied with $T = 220$, $I = 500000$ and $p = 15$. These runs required a mean total cpu time of 41087 seconds with a standard deviation of 309. The mean wall clock time was 9461 seconds with a standard deviation of 621. Using these same parameters, M_0/dXRLF was unsuccessful in trying to 148-color this graph in 10 attempts.

4.5. Structured Graph Results. All the Leighton graphs [**7**] except for the the c and d 25-colorable and 15-colorable ones have proven to be extremely easy to color optimally by a wide variety of methods. The results of coloring the hard Leighton graphs are given in Tables 16 and 17. Generally, the methods without chaining could not perform nearly as well as those that used chaining. The `25-col` and `15-col` columns give the number of successful coloring out of 10 attempted. Methods not listed were unsuccessful more than 50% of the time. All runs used $T = 22$ and $I = 10^5$.

The flat graphs in the DIMACS benchmark suite represent an anomaly in that both restarts and chaining are completely ineffective. Method S_0 exhibited the best performance by far. The `flat300_20_0.col` and `flat300_26_0.col` graphs are optimally colored in less than a minute over a wide range of temperatures ($20 \leq T \leq 45$). The `flat1000_50_0.col` and `flat1000_60_0.col` graphs were

		total cpu time		wall clock time		total class scans	
	25-col	mean	std dev	mean	std dev	mean	std dev
Graph le450_25c							
S_0	8	2821	1521			3.7×10^8	1.8×10^8
S_1	10	5035	2981			6.9×10^8	4.2×10^8
M_0	6	2612	1177	802	354	3.4×10^8	1.4×10^8
M_1	10	3435	2023	780	427	4.5×10^8	2.3×10^8
Graph le450_25d							
S_0	10	5971	4295			8.3×10^8	6.1×10^8
S_1	10	4472	3230			6.1×10^8	4.4×10^8
R_1	6	1628	1387			1.9×10^8	1.6×10^8
M_0	7	2133	2284	783	645	2.9×10^8	2.8×10^8
M_1	10	3810	1205	854	256	5.1×10^8	1.4×10^8

TABLE 16. Coloring the 25-chromatic Leighton Graphs

		total cpu time		wall clock time		total class scans	
	15-col	mean	std dev	mean	std dev	mean	std dev
Graph le450_15c							
S_1	10	126	99			1.1×10^7	7.9×10^6
R_1	10	258	221			2.3×10^7	2.1×10^7
M_0	9	443	426	171	183	4.6×10^7	4.3×10^7
M_1	10	522	482	142	103	5.2×10^7	4.7×10^7
Graph le450_15d							
S_1	10	80	34			7.0×10^6	3.2×10^6
R_1	10	104	36			9.7×10^6	3.9×10^6
M_0	6	1029	877	344	192	9.9×10^7	8.1×10^7
M_1	10	566	365	151	77	5.7×10^7	3.7×10^7

TABLE 17. Coloring the 15-chromatic Leighton Graphs

optimally colored in under a minute of cpu time with $T = 120$. What is of prime importance is that I be relatively large when compared to the values of I used for other graphs of this size. If $I < 10^6$ then all coloring attempts failed. With large $I > 10^6$, the coloring was always found in the first i-swap sequence. Though there was little hope of optimally coloring the `flat1000_76_0.col` graph, attempts (unsuccessful) were made to optimally color the `flat300_28_0.col` graph with four 36 hour runs.

M_0 was able to 101-color the `latin_square_10.col` graph in 194 seconds, 100-color it in 539 seconds, and 99-color it in 1107 seconds. The temperature used was $T = 280$ and iswap sequence size was $I = 10^5$. Other temperatures in the range $240 \leq T \leq 320$ were equally effective. In order to 98-color this graph, an anomaly was observed similar to the flat graphs. It was necessary to use $I = 5 \times 10^6$ in order for M_0 to produce 10 98-colorations out of 10 attempts. The mean total cpu time was 11223 seconds with a standard deviation of 6073 seconds. The mean wall clock time was 3122 seconds with a standard deviation of 1283 seconds.

The geometric graphs used in [**4**] were all colored with $I = 10^5$. The $U_{500,0.5}$ graph could be 124-colored in under a minute, and 123-colored in just under 3 minutes using S_0 with $T = 20$. The $U_{500,0.1}$ graph required a few seconds to 12-color using S_0 with $T = 20$, and an 85-coloration was obtained for the $\overline{U}_{500,0.1}$ graph in under a minute also using S_0, but with a temperature of 200.

5. Commentary

The main result of this study is that doing repeated, limited searches for improvements from a collection of the best colorations seen so far can be more effective than doing independent concurrent searches. We have also given an effective implementation for searching the impasse class neighborhood using rejection free Metropolis move selection.

The poor performance of the R_0 and R_1 methods shows that the good performance of the M_0 and M_1 methods depends very much on maintaining a pool of colorations; i.e., doing repeated restarts from the same coloration often results in the search becoming trapped. S_0 and S_1 did not exhibit this behavior and appeared to be able to move out of local traps when given enough time. When a pool of colorations is maintained, those that represent locally optimal traps (cannot be easily improved within the i-swap sequence length) get eliminated from the pool to be replaced by colorations that are more easily improved. Thus, the rate of coloration improvement is as big a factor as is the magnitude of coloration improvement. When we traced the lineage of the colorations after a run, very seldom was more than a single family alive. Though we have not been able to investigate this in any detail, we conjecture that those colorations that are most easily able to produce new improved colorations early in the run are the ones that will produce a complete coloration the fastest.

The most critical parameters are the temperature and i-swap sequence length.

Setting I to be larger than necessary seems to only affect running time and does not hurt a method's ability to find a k-coloration. A graph that has little variation in vertex degree, such as the flat graphs or the latin square graph, require a large value for I relative to the value used for other graphs of the same size. This may be because the Δ_j values are all very close in such graphs and so move selection approaches random permutation. The fact that the same performance is obtained for these graphs across a wide range of temperatures is evidence for this conjecture.

As k approaches the chromatic number of the graph, the range of values that can be used for T narrows. The procedure used to determine T and k for a family of graphs can double or even triple the coloring time of the first graph from the family. However, when large amounts of cpu time are available, the methods presented appear to be competitive when compared to the results given in [**2, 4, 8, 9**] and other studies.

Finally, in order to obtain good performance across the entire test bed, we found it necessary to incorporate the XRLF [**4**] algorithm to give us good initial colorations on large $G_{n,0.5}$ graphs. Perhaps the ultimate test of a coloring algorithm is its performance when passed a graph of unknown origin. An algorithm that combines several coloring techniques can outperform its components and may be able to pass this test. We believe that using a coloration pool with i-swaps, s-chaining and XRLF represents a step in this direction. Future work will involve self-configuration techniques and will likely use the survival-extinction mechanism of the coloration pool.

REFERENCES

1. F. Andrews and R. Olsson. 1993. *The SR Programming Language*, Benjamin/Cummings, Redwood City, CA.
2. B. Bollobás and A. Thomason. 1985. Random Graphs of Small Order. *Ann. Discrete Math.*, **28**, 47–97.
3. J. Greene and K. Supowit. 1986. Simulated Annealing Without Rejected Moves. *IEEE Trans. Computer-Aided Design*, **5**, 221–228.
4. D. Johnson, C. Aragon, L. McGeoch and C. Schevon. 1991. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, **39**, 378–406.
5. P.J.M. van Laarhoven. 1988. *Theoretical and Computational Aspects of Simulated Annealing*, CWI Tracts, Amsterdam.
6. S. Lavenberg, editor. 1983. *Computer Performance Modeling Handbook*, Academic Press, New York, New York.
7. F. T. Leighton. 1979. A Graph Coloring Algorithm for Large Scheduling Problems. *J. Res. Nat. Bur. Standards* **84**, 489–506.
8. C. Morgenstern. 1990. Algorithms for General Graph Coloring. Doctoral Dissertation, Department of Computer Science, University of New Mexico, Albuquerque, New Mexico.
9. C. Morgenstern and H. Shapiro. 1990. Coloration Neighborhood Structures for General Graph Coloring. First Annual ACM–SIAM Symposium on Discrete Algorithms.

Second DIMACS Challenge

Coloring Benchmark Results

GENERAL INFORMATION

Author: Craig Morgenstern

Title: Distributed Coloration Neighborhood Search

Name of Algorithms: Denoted as M_0, S_0 and S_1.

Description of Algorithm: Rejectionless Metropolis move selection.

Type of Machine: Five identically configured 80 MHz Sun Sparcstation-2's

Compiler and flags used: `gcc -O` and `sr -O`.

MACHINE BENCHMARKS

User time for instances:

r100.5	r200.5	r300.5	r400.5	r500.5
0.04	0.86	7.75	49.41	189.26

ALGORITHM BENCHMARKS

Authors' Comments: The M_0 distributed algorithm takes a target number of color classes, k, and maintains a pool of the best partial k-colorations seen so far. The coloring processes try to improve pool colorations to a complete k-coloration by doing sequences of what are called i-swap operations. A sequence is terminated when I consecutive i-swaps are performed without any improvement being found. The S_0 algorithm is a sequential coloring process that does an "unbounded" sequence of i-swaps (I is usually very large). S_1 is a sequential coloring algorithm that interleaves a small sequence of random s-chain interchanges between an unlimited number of "bounded" i-swap sequences. The coloring processes do rejectionless Metropolis move selection and run with with a fixed (optimized) temperature, T. M_0 used a pool of size 15 for all its runs. The data given is for the smallest values of k for which no coloring failures occurred in any of the runs. The number of runs performed per graph is given in column n, and the standard deviation for the total cpu times is given in column s.

The M_0 times reported are total cpu seconds, which is the sum of the cpu time taken by each of the five worker coloring processes and the manager process. This is an accurate measure of the time that would be required by a sequential simulation of M_0. Because of the coarse grain parallelism of coloring processes, M_0 communication time was negligible. The hidden cost not reflected by these times is the effort required to find the values for k, I and T. This hidden cost doubles or even triples the actual coloring time. On the other hand, this cost is removed for a class of graphs once the parameters have been determined. Some graphs are more easily colored by other sequential and distributed versions of M_0, and better results are possible if optimized values for pool size and i-swap sequence length are used. Much better results are obtained on the `DSJC500.5`, `DSJC1000.5`, and `C2000.5` graphs by using M_0 to improve on colorations produced by the XRLF independent set selection method.

Results on Benchmark Instances

Instance			Time (total cpu seconds)				Parameters		
Name	Alg	n	min	avg	s	max	k	I	T
DSJC125.5	M_0	10	1	14	12	32	17	2.5e3	20
DSJC250.5	M_0	10	386	591	137	838	28	5.0e4	35
DSJC500.5	M_0	10	10156	17852	6031	28878	49	5.0e5	63
DSJC1000.5	M_0	10	87887	91336	3184	94764	89	1.0e6	120
C2000.5	M_0	5	17893	31036	10461	43535	165	2.5e5	220
C4000.5		0							
R125.1	M_0	10	< 1	< 1	< 1	< 1	5	1.0e5	20
R125.1c	M_0	10	< 1	< 1	< 1	< 1	46	1.0e5	20
R125.5	M_0	10	< 1	< 1	< 1	< 1	36	1.0e5	20
R250.1	M_0	10	< 1	< 1	< 1	< 1	8	1.0e5	10
R250.1c	M_0	10	< 1	1	1	3	64	1.0e5	75
R250.5	M_0	10	26	181	202	509	65	1.0e5	15
DSJR500.1	M_0	10	< 1	< 1	< 1	< 1	12	1.0e5	15
DSJR500.1c	M_0	10	9	130	123	294	85	1.0e5	200
DSJR500.5	M_0	10	24	386	397	967	123	1.0e5	20
R1000.1	M_0	10	9	18	7	25	20	1.0e5	20
R1000.1c	M_0	10	825	1240	549	2143	98	1.0e5	500
R1000.5	M_0	10	430	2078	1158	3130	241	1.0e5	40
flat300_20	S_0	5	< 1	< 1	< 1	< 1	20	∞	35
flat300_26	S_0	5	14	22	10	40	26	∞	35
flat300_28	S_0	5	640	4214	2893	8012	31	∞	35
flat1000_50	S_0	5	< 1	< 1	< 1	< 1	50	∞	120
flat1000_60	S_0	5	< 1	< 1	< 1	< 1	60	∞	120
flat1000_76	S_0	5	2146	24291	19484	47380	89	∞	120
latin_sqr_10	M_0	10	1050	11223	6073	19707	98	5.0e6	280
le450_15a	S_1	10	< 1	< 1	< 1	< 1	5	1.0e5	22
le450_15b	S_1	10	< 1	< 1	< 1	< 1	5	1.0e5	22
le450_15c	S_1	10	27	126	99	280	15	1.0e5	22
le450_15d	S_1	10	43	80	34	127	15	1.0e5	22
mulsol.i.1	S_0	5	< 1	< 1	< 1	< 1	20	1.0e5	49
school1	S_0	5	< 1	< 1	< 1	< 1	20	1.0e5	20
school1_nsh	S_0	5	< 1	< 1	< 1	< 1	20	1.0e5	20

Department of Computer Science, Texas Christian University, Fort Worth, TX, USA 76129

E-mail address: morgenst@riogrande.cs.tcu.edu

This page intentionally left blank