A Large Neighborhood Search Heuristic for Graph Coloring

Michael A. Trick and Hakan Yildiz

Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA USA, 15213

Abstract. We propose a new local search heuristic for graph coloring that searches very large neighborhoods. The heuristic is based on solving a MAX-CUT problem at each step. While the MAX-CUT problem is formally hard, fast heuristics that give "good" cuts are available to solve this. We provide computational results on benchmark instances. The proposed approach is based on similar heuristics used in computer vision.

1 Introduction

Graph Coloring Problem (GCP) is one of the central problems in graph theory, has direct applications in practice, and is related to many other problems such as computer register allocation, bandwidth allocation, and timetabling. Given an undirected graph $G=(V, E)$, a *coloring* f of G is an assignment of a color to each vertex. A *proper (or feasible) coloring* is a coloring such that for each edge $(i, j) \in E$, vertices i and j have different colors. A *conflict* is the situation when two adjacent vertices have the same color assigned to them. We say that a coloring is *improper (or infeasible)* if there exists at least one conflict. The *conflict graph* of a graph G is the graph induced by the vertices that are incident to the conflicts in G.

A *minimum coloring* of G is a feasible coloring with the fewest different colors. It is well known that graph coloring is a hard combinatorial optimization problem [12], and exact solutions can be obtained for only small instances [14]. Therefore, heuristic algorithms are used to solve large instances. In this paper we introduce a new local search algorithm that searches large neighborhoods, based on ideas introduced by Boykov et al.[2].

The rest of the paper is organized as follows. In Section 2 we shortly review the known neighborhood search methods and introduce our very large neighborhood approach. We describe our algorithm in Section 3 and present the experimental results in Section 4. The conclusion is given in Section 5.

2 Local Search for Graph Coloring

Local search is based on the concept of a neighborhood. A *neighborhood* of a solution S is a set of solutions that are in some sense close to S, i.e., they can be easily computed from S or they share a significant amount of structure with S.

Local search for the GCP starts at some initial, improper coloring and iteratively moves to neighboring solutions, trying to reduce the number of conflicts.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 346–360, 2007.
© Springer-Verlag Berlin Heidelberg 2007

It is clear that the larger the neighborhood, the better is the quality of the solutions that can be reached in one single move. At the same time, the larger the neighborhood, the longer it takes to search the neighborhood at each iteration.

In this paper, we investigate a new local search method that uses very large scale neighborhoods. This is one of the first attempts to solve the GCP using local search in large neighborhoods. The only other large neighborhood searches we are aware of are due to Chiarandini et al.[5] and Avanthay et al.[1].

To make the notion of *large neighborhood* clear, we'll first introduce the well known *1-exchange* and *2-exchange neighborhoods*, which are small polynomially sized neighborhoods. Given a coloring, a *1-exchange move* changes the color of exactly one node and a *2-exchange move* swaps the colors of two vertices. The corresponding neighborhoods for these moves are the set of colorings that can be obtained by performing a single move.

We consider the neighborhoods proposed by Boykov et al. [2] for energy minimization problems in computer vision, and use these neighborhoods to solve the GCP. In the following subsections, we formally describe these neighborhoods and the corresponding moves, which are explained best in terms of partitions. Then we describe how to find the optimal moves by using graph cuts. The structures of the graphs, the cuts on these graphs, and the properties of the cuts are also explained in detail.

2.1 Moves and Neighborhoods

The first neighborhood is the α-β-swap: for a pair of colors $\{\alpha, \beta\}$, this move exchanges the colors between an arbitrary set of vertices colored α and another arbitrary set colored β. The second neighborhood we consider is α-expansion: for a color α, this move assigns the color α to an arbitrary set of vertices.

The GCP can be represented as a partitioning problem, in which a feasible coloring f corresponds to a partition of the set of vertices into K sets such that no edge exists between two vertices from the same color class. Let $\mathbf{V} = \{V_l | l \in L\}$ be such a partition, where L is the set of colors and $V_l = \{v \in V | f(v) = l\}$ is the subset of vertices assigned color $l \in L$.

Given a pair of colors (α, β), a move from a partition $\mathbf{V}$ (coloring f) to a new partition $\mathbf{V}'$ (coloring f') is called an α-β-swap if $V_l = V_l'$ for any color $l \neq \alpha, \beta$. This means that the only difference between $\mathbf{V}$ and $\mathbf{V}'$ is that some vertices that were colored α in $\mathbf{V}$ are now colored β in $\mathbf{V}'$, and some vertices that were colored β in $\mathbf{V}$ are now colored α in $\mathbf{V}'$.

Given a color α, a move from a partition $\mathbf{V}$ (coloring f) to a new partition $\mathbf{V}'$ (coloring f') is called an α-expansion if $V_\alpha \subset V_\alpha'$ and $V_l' \subseteq V_l$ for any label $l \neq \alpha$. In other words, an α-expansion move allows any set of vertices to change their colors to α.

2.2 Size of the Neighborhoods

For an α-β-swap, each vertex either keeps its current color or switches to the other one. Since each partition has $\Omega(n)$ vertices, the possible solutions that can be reached by one swap move is $2^{\Omega(n)}$.

For an α-expansion, each vertex that is not colored α either keeps its old color or acquires the new color α. Since there are $\Omega(n)$ such vertices, the possible solutions that can be reached by one expansion move is $2^{\Omega(n)}$. These imply:

Lemma 1. *The size of both neighborhoods is* $2^{\Omega(n)}$.

2.3 Graph Cuts

The important part of the local search algorithms, which are presented in the following sections, is efficiently finding the best neighboring solution to the current solution by using graph cuts. Let $G = (V, E)$ be a connected and undirected edge weighted graph. A *cut* C of G is a minimal subset of E, which increases the number of connected components by exactly one. The *weight* (or *cost*) of a cut C is the sum of the weights of edges in the cut and is represented by $w(C)$. A *maximum cut* (or a *maxcut*) is then defined as a cut of maximum weight.

2.4 Finding the Optimal Swap Move

Given a coloring f and a pair of colors $\{\alpha, \beta\}$, we want to find a coloring $\hat{f}$ that minimizes the number of conflicts over all colorings within one α-β-swap of f. Our technique is based on computing a coloring that corresponds to a maximum cut on the subgraph $G^{\alpha\beta} = (V^{\alpha\beta}, E^{\alpha\beta})$, which is a clique over the vertices colored with α or β in f. For all $(i, j) \in E^{\alpha\beta}$, we assign a weight equal to one if $(i, j) \in E$ and a weight equal to zero if $(i, j) \notin E$. The latter ensures that $G^{\alpha\beta}$ is connected. The structure of the graph $G^{\alpha\beta}$ is illustrated in Fig. 1.

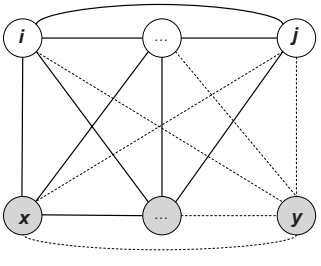

Fig. 1. An example of $G^{\alpha\beta}$. The set of vertices are $V^{\alpha\beta} = V_\alpha \cup V_\beta$ where $V_\alpha = \{i, ..., j\}$ and $V_\beta = \{x, ..., y\}$. Solid edges are induced edges and have weight 1. Dashed edges are artificial edges and have weight 0, which ensure that $G^{\alpha\beta}$ is connected.

Every edge, with a weight 1, between the vertices of the same color is a conflict. Every swap move defines a new bipartition of the vertex set $V^{\alpha\beta}$, possibly with a different number of conflicts. Notice that every cut in this graph defines a swap move that results a bipartition of the vertex set, thus a new coloring. In order to obtain the optimal swap move that results with minimum number of conflicts, we need to minimize the total weight of edges within the partitions. Notice that this is equivalent to maximizing the total weight of edges between

the two partitions, which is equivalent to solving a maxcut problem on $G^{\alpha\beta}$. After finding a maxcut, the vertices in one partition are going to be colored α, and the vertices in the other partition will be colored β. The selection of which partition will be colored α is arbitrary. This implies:

Theorem 1. *Let $G^{\alpha\beta}$ be constructed as described above for a given f and $\{\alpha, \beta\}$ and let T be the total weight of edges in $G^{\alpha\beta}$. A coloring f^C corresponding to a cut C on $G^{\alpha\beta}$ is one α-β-swap away from the initial coloring f. Moreover the optimal α-β-swap move is equivalent to a maxcut C^* in $G^{\alpha\beta}$ and the number of conflicts within $G^{\alpha\beta}$ for the new coloring f^{C^*} is x if $w(C^*) = T - x$.*

2.5 Finding the Optimal Expansion Move

Given an input coloring f and a color α, we want to find a coloring $\hat{f}$ that minimizes the number of conflicts over all colorings within one α-expansion of f. Our technique is based on computing a coloring that corresponds to a maximum cut on the graph $G^{\alpha} = (V^{\alpha}, E^{\alpha})$. The structure of this graph is determined by the current partition $\mathbf{V}$ and by the color α, so the graph dynamically changes after each iteration.

The structure of the graph is illustrated in Fig. 2. The set of vertices include all vertices $v \in V$. Moreover it includes two terminals α and $\overline{\alpha}$, which are auxiliary vertices representing the color α in consideration and the rest of the colors, respectively. In addition, we have six types of auxiliary vertices. For each edge that is incident to two vertices with color α, we create an auxiliary vertex of type A_1. For each edge that is incident to exactly one vertex with color α, we create an auxiliary vertex of type A_2. For each adjacent vertex pair such that neither vertex in the pair is colored with α, we create two auxiliary vertices of types B_1 and B_2 if the pair has different colors. If they are colored with the same color, say γ, we create two vertices of types D_1 and D_2.

We now explain the way we connect the vertices by edges with different wights. The weights assigned to these edges are summarized in Table 1. The two terminals are connected by an edge with a very high weight M to ensure that the maxcut that is found separates the two terminals α and $\overline{\alpha}$. Each vertex $v \in V$ is connected by an edge to the terminals α and $\overline{\alpha}$. Each pair of adjacent vertices $\{i, j\} \in V$ is connected by edges to the auxiliary vertices corresponding to that pair. Each pair of adjacent vertices such that neither of them are colored with α are connected by an edge. In addition to these, type $A1$, $A2$, B_1, and D_1 vertices are connected by an edge to the terminal α, and type B_2 and D_2 vertices are connected by an edge to the terminal $\overline{\alpha}$. As a result, each adjacent vertex pair and the auxiliary vertices corresponding to the pair and the edges incident to these vertices form four different structures, which we call as *gadgets*. The four different gadgets that correspond to four edge types of the original graph G are illustrated in Fig. 2. Formally, for an edge (i, j), there are four possible situations depending on the colors of the vertices incident to those edges:

1. $f(i) = f(j) = \alpha$
2. $f(i) = \alpha$, $f(j) \neq \alpha$, or $f(i) \neq \alpha$, $f(j) = \alpha$

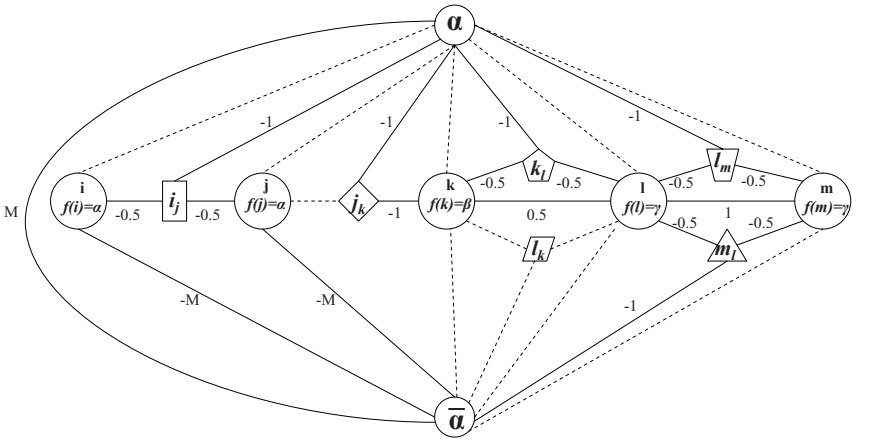

Fig. 2. An example of G^α. Dashed edges have weight 0. The type sets for auxiliary vertices: T=$\{\alpha, \overline{\alpha}\}$, A_1=$\{i_j\}$, A_2=$\{j_k\}$, B_1=$\{k_l\}$, B_2=$\{l_k\}$, D_1=$\{l_m\}$, D_2=$\{m_l\}$.

3. $f(i) \neq \alpha$, $f(j) \neq \alpha$, $f(i) = f(j)$
4. $f(i) \neq \alpha$, $f(j) \neq \alpha$, $f(i) \neq f(j)$

Any cut C on G^α, which separates the two terminals α and $\overline{\alpha}$, must include exactly one of the edges that connect $v \in V$ to the terminals. This defines a natural coloring f^C corresponding to a cut C on G^α. Formally,

$$f^C(v) = \begin{cases} \alpha, & \text{if } (v, \alpha) \in C \\ f(v), & \text{if } (v, \overline{\alpha}) \in C \end{cases}$$

In other words, a vertex v is assigned color α if the cut C separates v from the terminal α and, v is assigned its old color $f(v)$ if C separates v from $\overline{\alpha}$. Clearly this implies:

Lemma 2. *A coloring f^C corresponding to a cut C on G^α, which separates the two terminals α and $\overline{\alpha}$, is one α-expansion away from the initial coloring f. Also, an α-expansion move is equivalent to a cut C on G^α, which separates the two terminals α and $\overline{\alpha}$.*

For each of the four gadgets, a maxcut C on G^α severs some of the edges of the gadgets, and the sum of the weights of the edges severed is consistent with whether the corresponding edge (i, j) is a conflict or not in f^C.

Property 1. For any maxcut C and for any edge $(i, j) \in E$ such that at least one of i and j is colored with α in f

a) *If* (α, i), $(\alpha, j) \in C$ *then either* $(\alpha, i_j) \in C$ *or* (i, i_j), $(i_j, j) \in C$
b) *If* $(\overline{\alpha}, i)$, $(\overline{\alpha}, j) \in C$ *then no other edge from the gadget is in* C
c) *If* (α, i), $(\overline{\alpha}, j) \in C$ *then either* $(i, i_j) \in C$ *or* (α, i_j), $(i_j, j) \in C$
d) *If* (α, j), $(\overline{\alpha}, i) \in C$ *then either* $(i_j, j) \in C$ *or* (α, i_j), $(i, i_j) \in C$

Property 1 follows from the maximality of $w(C)$ and it is illustrated in Fig. 3.

Table 1. The weights assigned to the edges in the graph presented in Fig. 2

edge	weight	for	example(Fig. 2)
(v,α)	0	$v \in V$	(i,α),(j,α),(k,α),(l,α),(m,α)
$(v,\overline{\alpha})$	$-M$	$v \in V_\alpha$	$(i,\overline{\alpha})$,$(j,\overline{\alpha})$
$(v,\overline{\alpha})$	0	$v \in V \backslash V_\alpha$	$(k,\overline{\alpha})$,$(l,\overline{\alpha})$,$(m,\overline{\alpha})$
(a,α)	-1	$a \in A_1, A_2, B_1, D_1$	(i_j,α),(j_k,α),(k_l,α), (l_m,α)
$(b,\overline{\alpha})$	0	$b \in B_2$	$(l_k,\overline{\alpha})$
$(c,\overline{\alpha})$	-1	$c \in D_2$	$(m_l,\overline{\alpha})$
(v,a)	-0.5	$v \in V_\alpha$, $a \in A_1$	(i,i_j), (i_j,j)
(v,a)	-1	$v \in V \backslash V_\alpha$, $a \in A_2$	(j_k,k)
(v,a)	0	$v \in V_\alpha$, $a \in A_2$	(j,j_k)
(v,b)	-0.5	$v \in V \backslash V_\alpha$, $b \in B_1, D_1, D_2$	(k,k_l),(l,k_l),(l,l_m), (l,m_l),(m,l_m),(m,m_l)
(v,b)	0	$v \in V \backslash V_\alpha$, $b \in B_2$	(l_k,k), (l_k,l)
(v,w)	0.5	$v,w \in V \backslash V_\alpha$, $f(v) \neq f(w)$	(k,l)
(v,w)	1	$v,w \in V \backslash V_\alpha$, $f(v) = f(w)$	(l,m)
$(\alpha,\overline{\alpha})$	M	$(\alpha,\overline{\alpha})$	$(\alpha,\overline{\alpha})$

In the case that $f(i) = f(j) = \alpha$, since the weight of the edges that connect i and j to $\overline{\alpha}$ has weight $-M$, the only maxcut possible is one of the cuts described in Property 1(a). Since both of the cuts separate i and j from α, the colors of i and j stay unchanged: $f^C(i) = f^C(j) = \alpha$. If $(\alpha, i_j) \in C$, or if (i, i_j), $(i_j, j) \in C$ then the cost is -1. In both cases, the cost incurred is truly consistent with the fact that (i, j) is a conflict in f^C.

In the case that one of i and j is colored with α in f, assume w.l.o.g. $f(i) = \alpha$, $f(j) = \beta$, the possible cuts are the ones described in Property 1 (a) or (c). The cuts that sever $(i, \overline{\alpha})$ are not possible since the weight of $(i, \overline{\alpha})$ is $-M$. The cost of the cuts having Property 1(a) is -1, which is consistent with the fact that (i, j) is a conflict in f^C. The cost of the cuts having Property 1(c) is 0, which is consistent with the fact that (i, j) is not a conflict in f^C.

Property 2. For any maxcut C and for any edge $(k, l) \in E$ such that $f(k) \neq \alpha$, $f(l) \neq \alpha$:

a) *If* (α, k), $(\alpha, l) \in C$ *then either* $(\alpha, k_l) \in C$ *or* (k, k_l), $(k_l, l) \in C$
b) *If* $(\overline{\alpha}, k)$, $(\overline{\alpha}, l) \in C$ *then either* $(\overline{\alpha}, l_k) \in C$ *or* (k, l_k), $(l_k, l) \in C$
c) *If* (α, k), $(\overline{\alpha}, l) \in C$ *then* (k, k_l), $(l_k, l) \in C$
d) *If* (α, l), $(\overline{\alpha}, k) \in C$ *then* (k, l_k), $(k_l, l) \in C$

Property 2 follows from the maximality of $w(C)$ and from the fact that no subset of C is a cut. Property 2 is illustrated in Fig. 4.

In the case that $f(k) = f(l) = \beta$, all the cuts described in Property 2 are possible. The cost of the cuts that have Property 2(a) or (b) is -1, which is consistent with the fact that (k, l) is a conflict in f^C in both cases. The cost of the cuts that have Property 2(c) or (d) is 0, which is consistent with the fact that (k, l) is not a conflict in f^C in either case, since the color of exactly one of k, l is changed to α.

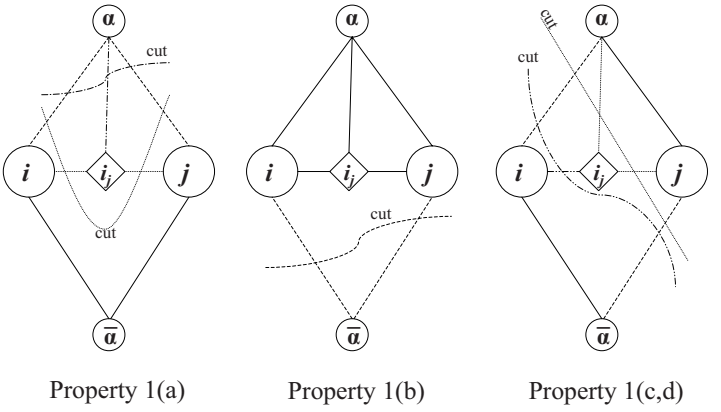

Fig. 3. Properties of a maxcut C on G^α for two vertices $i,j \in V$ such that at least one of them is colored with α

In the case that $f(k) = \beta$, $f(l) = \gamma$, all the cuts described in Property 2 are possible. The cost of the cuts that have Property 2(a) is -1, which is consistent with the fact that (k, l) is a conflict in f^C. The cost of the cuts that have Property 2(b), (c) or (d) is 0, which is consistent with the fact that (k, l) is not a conflict in f^C in those cases, since the color of at least one of k, l is not changed to α. Lemma 2, Property 1 and Property 2 implies:

Theorem 2. *A coloring f^{C^*} corresponding to a maxcut C^* on G^α is one α-expansion away from the initial coloring f. Moreover the optimal α-expansion move is equivalent to a maxcut C^* in G^α and the number of conflicts for the new coloring f^{C^*} is x if $w(C^*) = M - x$.*

3 Algorithms

In this section we first introduce swap-move, check-bipartite and expansion-move algorithms. These three algorithms are used as subroutines in the expansion-swap algorithm, which is the main algorithm.

3.1 Swap-Move Algorithm

Input: A coloring f of G with K colors.

1. Set Success:=0, Any_Imp(swap):=0
2. Set Improvement(swap):= 0
3. For each pair of colors $\{\alpha, \beta\} \subset L$
   3.1 Find $\hat{f}$ that minimizes the number of conflicts among all possible new colorings within one α-β-swap of f
   3.2 IF the number of conflicts is reduced, f:= $\hat{f}$, Improvement(swap):=1, Any_Imp(swap):=1

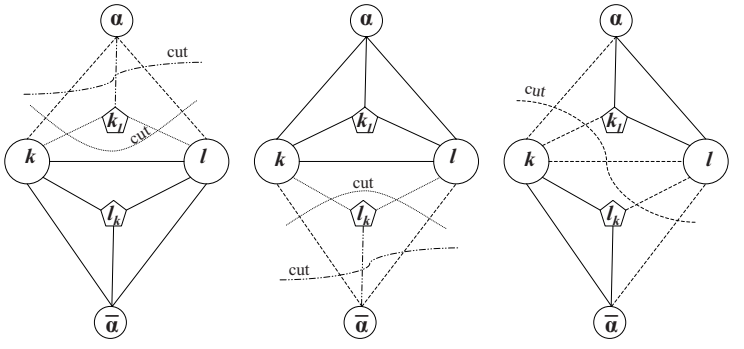

Fig. 4. Properties of a maxcut C on G^{α} for two vertices k,$l \in V$ such that none of them is colored with α

4. IF f has no conflicts, return f with Success := K
5. IF Improvement(swap) = 1, goto 2
6. Return f with Success:= 0 & Any_Imp(swap).

3.2 Expansion-Move Algorithm

Input: A coloring f of G with K colors.

1. Success:=0, Any_Imp(expansion):=0
2. Set Improvement(expansion):= 0
3. For each color $\alpha \in L$
   3.1 Find $\hat{f}$ that minimizes the number of conflicts among all possible new colorings within one α-expansion of f
   3.2 IF the number of conflicts is reduced, f:= $\hat{f}$, Improvement(expansion):=1, Any_Imp(expansion):=1
4. IF f has no conflicts, return f with Success:= K & Any_Imp(expansion)
5. IF Improvement(expansion)= 1, goto 2
6. Return f with Success:= 0 & Any_Imp(expansion).

3.3 Check-Bipartite Algorithm

If the conflict graph of an infeasible coloring f of G with K colors is bipartite, one can remove all conflicts without creating new conflicts and end up with a feasible $K+1$ coloring by coloring the vertices of one partition with a new color $K+1$. A short algorithm based on this observation is given below.

Input: A coloring f of G with K colors.

1. If f has conflicts, let $G^{'} = (V^{'},E^{'})$ be the subgraph induced by the conflicting vertices
   1.1 Starting from an arbitrary source vertex, color the vertices and their neighbors in alternation with colors α and β.
   1.2 If the resulting coloring is proper, then $G^{'}$ is bipartite. Introduce a new color $K+1$ to color the vertices in one partition, return f with Success:= $K+1$.

3.4 Expansion-Swap Algorithm

Expansion-swap is the main algorithm that takes advantage of both neighborhoods introduced and uses the swap-move, check-bipartite and expansion-move algorithms as subroutines. Namely, when the swap-move no longer improves the current solution, expansion-move tries to improve the current solution without starting from scratch. When expansion-move is unable to improve the solution, swap-move starts running again. When neither algorithm is able to improve the solution, first we check if the conflict graph is bipartite with the Check-bipartite algorithm. If it is not bipartite, then at Step 8, a new color $K+1$ is introduced by finding $\hat{f}$ that minimizes the number of conflicts among $f^{'}$ within one $(K+1)$-expansion of f.

Introducing a new color by an expansion move is a better approach than introducing it by finding an independent set on the conflict graph, since expansion allows for the creation of new conflicts in exchange for removing more current conflicts. However in the independent set case, coloring adjacent vertices with the new color is not permitted. This is easy to see when we assume that the conflict graph is a dense graph, such as a clique, where the maximum independent set is only one vertex.

Introducing the new color $K+1$ will definitely improve the solution if not actually remove all the conflicts. After this improvement, the swap-move phase will search for a better solution, and the whole cycle repeats until a feasible coloring is found.

Input: A graph G and an initial number of colors K.

1. Randomly color G with K colors, resulting in coloring f
2. Swap-move(G, f, K)
3. IF Success = 0, Expansion-move(G, f, K)
4. IF Success $>$ 0 return f
5. IF Any_Imp(expansion) = 1, Swap-move(G, f, K); ELSE goto 7
6. IF Success $>$ 0 return f;
   ELSE IF Success = 0 & Any_Imp(swap) = 1, goto 3
7. Check-bipartite(G, f, K)
8. IF Success = 0, find $\hat{f}$ that minimizes the number of conflicts among all possible new colorings within one $(K+1)$-expansion of f, set $f := \hat{f}$
9. IF f has no conflicts, return f with Success := $K+1$; ELSE set $K := K+1$, goto 2.

3.5 Finding a Maximum Cut

Given an undirected graph G with edge weights, the *MAX-CUT problem* consists of finding a maxcut of G. MAX-CUT is a well-known NP-Hard problem[12]. Since all our algorithms rely on solving MAX-CUT problems several times, the solution times for our algorithms can be expected to be out of our limits for a local search heuristic. So rather than determining the maxcut at each move, we can find a "good" cut, which has a weight that is close to the weight of a maxcut. This allows us to search the introduced neighborhood approximately and fast. However, since Theorem 2 is dependent on the cut found being a maxcut, we can not use the total weight of the cut found to calculate the number of conflicts. But notice that we can still use any cut to find a new coloring as Lemma 2 holds for any cut. For this reason, we use the cut obtained to define the new coloring and we calculate the actual number of conflicts by checking the adjacency matrix and the new coloring.

There are many heuristic and approximation algorithms that have been computationally tested and/or with theoretical performance guarantee. Goemans and Williamson [13] proposed a randomized algorithm that uses semidefinite programming to achieve a performance guarantee of 0.87856. More recent algorithms for solving the semidefinite programming relaxation are particularly efficient, because they exploit the structure of the MAX-CUT problem. Burer, Monteiro, and Zhang [4] proposed a rank-2 relaxation heuristic for MAX-CUT and described a computer code, called *Circut* [7], that produces better solutions in practice than the randomized algorithm of Goemans and Williamson. Circut does not assume the edge weights are positive. This property is necessary for our algorithm as the graphs created by our algorithm have edges with negative weights. Moreover, since the performance of Circut on many different problems has been shown to be very good, and the code is available for outside use, we decided to use Circut to solve MAX-CUT problems in our algorithms.

4 Experimental Results

4.1 Implementation Details

Our algorithm is implemented in C. Since the MAX-CUT solver code, Circut, is implemented in Fortran90, the input/output transaction between the main code and Circut is made through text files. For large size problems, writing into and reading from files takes very long times. This becomes a serious issue especially for the expansion graphs created for the expansion move since the size of the expansion graphs are much larger than the original graph. To overcome this disadvantage, we used the following strategy for large graphs: Expansion moves were only used for introducing a new color when swap-move is stuck with the current solution. We introduce the new color by finding the best expansion move on the conflict graph rather than the original graph, since the color of non-conflicting vertices do not change during an expansion move. The conflict graph

is smaller than the original graph in almost all cases, and becomes smaller as the number of conflicts is reduced during the execution of the algorithm. This observation made it possible for us, not fully but at least partially, to use the expansion move idea for large instances.

In addition to the modification described above, we have made two more changes in the original expansion-swap algorithm in order to decrease the execution time: First, we use the simple 1-exchange moves after Step 5 and Step 9 of the expansion-swap algorithm. Second, after introducing a new color at Step 9, we have looked at the best swap moves only between the new color and the old ones, but not between all the old colors.

4.2 Summary of Results

We run the expansion-swap algorithm on a 450 MHz Sun UltraSPARC-II workstation with 1024MB of RAM. We tested our algorithms on some of the benchmark instances proposed for COLOR02/03/04 [8].

Table 2 and Table 2 compare the results of the Expansion-Swap algorithm(ES) to the results of the heuristics proposed by Croitoru et al.(CL)[9], Galinier et al.(GH)[11], Bui and Patel(BP)[3], Phan and Skiena(PS)[15], Chiarandini and Stuetzle(CS)[6]. These results are summarized in [8]. Not all heuristics reported their results for all instances. Thus many cells in the table are empty.

Of the 68 test instances solved with the Expansion-Swap Algorithm, there are 18 instances with reported chromatic numbers [8]. Of these 18 instances our algorithm found the optimal solution for 10 of them.

The results of the ES algorithm for 32 instances are either equal to the optimal solution, or as good as the best result found by other heuristics listed. The results of ES for these instances are highlighted in bold in Tables 2 and 2. For 15 instances, ES either could not find the optimal solution or at least one of the other heuristics obtained a better solution. For 14 instances, ES obtained the worst results. And for the remaining 7 instances, we cannot make a comparison as we only have the results of ES but we see that these results are only one more than the clique number of 4 of these 7 instances.

In terms of instance types, we can say that our algorithm performed very well on myciel and FullIns instances, and on school1-nsh and mugg100-25. It also has a good performance on all other graphs except DSJ instances and latin-square-10. For DSJ and latin-square, the quality of the solutions are not as good, especially for the large and dense instances.

In terms of solution times, 16 instances are solved in less than 1 second, 39 instances are solved in more than 1 second but in less than 1 minute, 5 instances are solved in more than 1 minute but less than 4 minutes, 4 instances are solved in more than 4 minutes but in less than 8 minutes, and the remaining 4 instances are solved in more than 8 minutes but in less than 16 minutes.

Figure 5 presents the relationship of the solution time with the density of the graphs. As one would expect, the hardest instances are those with high density, though this does not fully explain the heuristic's running time since some high density instances can be solved quickly.

Table 2. Comparison of the results of the expansion-swap(ES) algorithm to the results of other heuristics. The columns in the table consist of the name of the graph, number of vertices (n), number of edges (m), density of the graph (d), clique number (cl), optimum solution(OPT), lower bound(LB), results due to (CL), (GH), (BP), (PS), (CS), results for expansion-swap (ES) algorithm, and time in seconds for ES (time).

Graph	n	m	d	cl.	OPT	LB	CL	GH	BP	PS	CS	ES	time
le450_5a.col	450	5714	6%	5	5	5			5	14		**5**	*3.6*
le450_5b.col	450	5734	6%	5	5	5			5	13		**5**	*8.6*
le450_5c.col	450	9803	10%	5								**5**	*4.2*
le450_5d.col	450	9757	10%	5	5	5				16		**5**	*4.1*
le450_15a.col	450	8168	8%	15		15	18		15	23	15	*18*	*51.2*
le450_15b.col	450	8169	8%	15	15	15	18		15	23	15	*18*	*46.4*
le450_15c.col	450	16680	17%	15		15	27	15		32	16	*25*	*90.8*
le450_15d.col	450	16750	17%	15		9		15		31	16	*26*	*36.5*
le450_25a.col	450	8260	8%	25								*26*	*21.3*
le450_25b.col	450	8263	8%	25								*26*	*19.8*
le450_25c.col	450	17343	17%	25		25		26		36	26	*32*	*29.3*
le450_25d.col	450	17425	17%	25		13		26		37	26	*31*	*100.9*
queen8_8.col	64	728	36%	9	9	9						*10*	*7.2*
queen8_12.col	96	1368	30%	12								*13*	*4.5*
queen9_9.col	81	2112	65%	10					10			*11*	*12.2*
queen10_10.col	100	2940	59%									*13*	*3.5*
queen11_11.col	121	3960	55%	11					12			*14*	*4.5*
queen12_12.col	144	5192	50%									*15*	*12.8*
queen13_13.col	169	6656	47%	13					14		14	*16*	*107.9*
queen14_14.col	196	8372	44%									*17*	*75.9*
queen15_15.col	225	10360	41%						17			*19*	*9.9*
queen16_16.col	256	12640	39%							21	18	*19*	*30.0*
myciel5.col	47	236	22%	6								**6**	*0.5*
myciel6.col	95	755	17%	7					7			**7**	*0.9*
myciel7.col	191	2360	13%	8					8			**8**	*1.4*
1-Insertions_4.col	67	232	10%	4		4	4		4			*5*	*0.4*
1-Insertions_5.col	202	1227	6%			4	6			6		**6**	*0.9*
1-Insertions_6.col	607	6337	3%				7			15		**7**	*5.1*
2-Insertions_3.col	37	72	11%	4								**4**	*0.2*
2-Insertions_4.col	149	541	5%	4	4	4	5		4	5	5	*5*	*0.4*
2-Insertions_5.col	597	3936	2%			4	6			11		**6**	*4.2*
3-Insertions_3.col	56	110	7%	4								**4**	*0.2*
3-Insertions_4.col	281	1046	3%			3	5			5	5	**5**	*1.2*
3-Insertions_5.col	1406	9695	1%				6			29	6	**6**	*35.4*
4-Insertions_3.col	79	156	5%	3		3			4			**4**	*0.2*
4-Insertions_4.col	475	1795	2%			3				7		**5**	*2.3*
1-FullIns_3.col	30	100	23%	4	4	4	4					**4**	*0.2*
1-FullIns_4.col	93	593	14%	5	5	5	5					**5**	*0.4*
1-FullIns_5.col	282	3247	8%	6	6	6	6			7		**6**	*1.1*
2-FullIns_3.col	52	201	15%	5		5	5					**5**	*0.3*

Table 2. *(continued)*

Graph	n	m	d	cl.	OPT	LB	CL	GH	BP	PS	CS	ES	time
2-FullIns_4.col	212	1621	7%			5	6			7		**6**	*0.7*
2-FullIns_5.col	852	12201	3%			6	7			23		**7**	*9.1*
3-FullIns_3.col	80	346	11%		5	5	6					**6**	*0.4*
3-FullIns_4.col	405	3524	4%			6	7			11	7	**7**	*4.6*
3-FullIns_5.col	2030	33751	2%			6	8			59	8	**8**	*53.7*
4-FullIns_3.col	114	541	8%	7	7	7	7					**7**	*1.0*
4-FullIns_4.col	690	6650	3%			7	8			19		**8**	*7.6*
4-FullIns_5.col	4146	77305	1%				9				9	*11*	*325.0*
5-FullIns_3.col	154	792	7%	8	8	8				8		**8**	*2.2*
5-FullIns_4.col	1085	11395	2%							27		**10**	*11.8*
DSJC125.1.col	125	736	9%	5	5	5			5	7		*6*	*1.0*
DSJC125.5.col	125	3891	50%	12		12	20		18	21		*21*	*11.8*
DSJC125.9.col	125	6961	90%	27		30			42	46		*48*	*50.2*
DSJC250.1.col	250	3218	10%		8	8			9			*10*	*3.3*
DSJC250.5.col	250	15668	50%			13	37		22		28	*36*	*46.9*
DSJC250.9.col	250	27897	90%			35			72	79		*82*	*230.4*
DSJC500.1.col	500	12458	10%			6	16	12		20	12	*15*	*42.0*
DSJC500.5.col	500	62624	50%			16	66	48	51		50	*61*	*256.7*
DSJC500.9.col	500	112437	90%	35		42		126			127	*156*	*838.3*
DSJR500.1.col	500	3555	3%	12	12	12	12					**12**	*5.3*
DSJR500.1c.col	500	121275	97%	63	63	63	56			105		*94*	*418.1*
DSJR500.5.col	500	58862	47%	26	26	26			129	155	124	*143*	*474.2*
DSJC1000.1.col	1000	49629	10%			6		20		41		*26*	*50.5*
DSJC1000.5.col	1000	249826	50%			17		84				*111*	*793.4*
DSJC1000.9.col	1000	449449	90%	37		54		224				*289*	*796.6*
latin_sq_10.col	900	307350	76%						101		99	*123*	*901.5*
school1_nsh.col	352	14612	24%	14	14	14			14	33		**14**	*29.3*
mugg100_25.col	100	166	3%	4								**4**	*0.3*

5 Conclusion

We studied a new local search algorithm using two very large-scale neighborhoods for the GCP. The first type of move allows us to swap the colors of sets of vertices. The second type of move allows any set of vertices to change their colors to a particular color. The algorithm proposed combines these two types of moves.

The key part of the algorithm is efficiently finding the best neighboring solution to the current solution by solving a MAX-CUT problem. Since MAX-CUT is a hard problem, we considered approximate algorithms that are able to find "good" solutions very fast. It is important to note that the success of the algorithms presented in this paper hinges on fast algorithms that can solve MAX-CUT problems optimally or approximately.

This study is one of the first attempts to solve the GCP using local search in very large neighborhoods. Although we could not fully take advantage of the

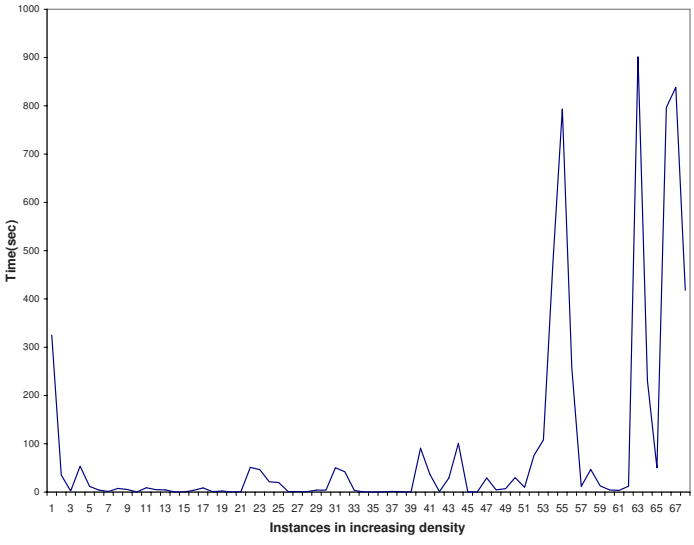

Fig. 5. The x-axis is the ordered list of 68 instances solved. The instances are ordered in ascending order of density. Instance number 1 is the least dense and instance number 68 is the most. The y-axis is the solution time.

neighborhoods by solving the MAX-CUT problems optimally, the results we present here are promising. However, further research efforts are still required to make large scale neighborhood techniques fully competitive.

One possible extension is to use exact or better approximate algorithms and to fully integrate them with the main code to solve the MAX-CUT problems. Another one is to investigate if a best-improvement variant of the Expansion-Swap algorithm would perform better than the current first-improvement search approach we use. That is instead of accepting the first improving move, using the move that gives the best improvement in conflicts.

References

1. Avanthay, C., Hertz, A., Zufferey, N.: A variable neighborhood search for graph coloring. European Journal of Operational Research, **151** (2003) 379–388.
2. Boykov, Y., Veksler, O., Zabih, R.: Fast approximate energy minimization via graph cuts. IEEE Transactions on Pattern Analysis and Machine Intelligence, **23(11)** (2001) 1222–1239
3. Bui, T.N., Patel, C.M.: An Ant System Algorithm for Coloring Graphs. In D.S. Johnson, A. Mehrotra, and M. Trick, editors, Pro- ceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
4. Burer, S., Monteiro, R.D.C., Zhang T.: Rank-two relaxation heuristics for MAX-CUT and other binary quadratic programs. SIAM Journal on Optimization, **12** (2001) 503-521

5. Chiarandini, M., Dumitrescu, I., Stuetzle, T.: Local search for the colouring graph problem. A computational study. Technical Report AIDA-03-01, FG Intellektik, TU Darmstadt (2003)
6. Chiarandini, M., Stuetzle, T.: An application of Iterated Local Search to Graph Coloring Problem. In D. S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
7. CirCut: A Fortran 90 Code for Max-Cut, Max-Bisection and More. http://www.caam.rice.edu/ zhang/circut/
8. COLOR02/03/04: Graph Coloring and its Generalizations. http://mat.gsia.cmu.edu/COLOR04
9. Croitoru, C., Luchian, H., Gheorghies, O., Apetrei, A.: A New Genetic Graph Coloring Heuristic. In D. S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
10. Galinier, P., Hertz, A.: A Survey of Local Search Methods for Graph Coloring. Computers & Operations Research **33** (2006) 2547–2562.
11. Galinier, P., Hertz, A., Zufferey, N.: Adaptive Memory Algorithms for Graph Coloring. In D.S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
12. Garey, M.R., Johnson, D.S.:Computers and Interactibility: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, USA, (1979).
13. Goemans, M.X., Williamson, D.P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of ACM **42** (1995) 1115-1145.
14. Mehrotra, A., Trick, M.: A column generation approach for graph coloring. INFORMS Journal On Computing **8(4)** (1996) 344-354
15. Phan, V., Skiena, S.: Coloring Graphs With a General Heuristic Search Engine. In D.S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
16. Thompson, P.M, Psaraftis, H.N.: Cyclic transfer algorithms for multivehicle routing and scheduling problems. Operations Research **41** (1993) 70-79