Graph Coloring Algorithms*

Walter Klotz

Abstract

Deterministic graph coloring algorithms of contraction and sequential type are investigated. Sequential algorithms can be extended by backtracking to relatively effective algorithms for the chromatic number of a graph. Incomplete backtracking leads to new heuristics for graph coloring.

Keywords: graph coloring, chromatic number.

2000 Mathematics Subject Classification: 05C15, 05C85.

1 Introduction

Graph coloring serves as a model for conflict resolution in problems of the following type. Suppose that in a set V certain pairs of elements are incompatible. The problem is to find a partition of V into a minimal number of subsets of mutually compatible elements. The situation is described by a graph $G = (V, E)$ with vertex set V and edge set E formed by all pairs of incompatible Elements. Partitioning of V into k subsets is equivalent to coloring the vertices of G with k colors.

A *(proper) coloring* of the vertices of a graph $G = (V, E)$ is a map $F : V \to \mathbb{N}$, where adjacent vertices receive distinct colors in $\mathbb{N}$; that is, if $uv \in E$, then $F(u) \neq F(v)$. The *chromatic number* $\chi(G)$ is the minimum number of colors needed for a coloring of G. A graph G is $k-chromatic$, if $\chi(G) = k$, and G is $k-colorable$, if $\chi(G) \leq k$. A *color class* of a coloring F contains all vertices of the same color. The color classes of F form a partition of V into *independent* subsets, i.e. subsets of pairwise non-adjacent vertices.

*Mathematik-Bericht 5 (2002), 1-9, TU Clausthal

The problem of determining the chromatic number $\chi(G)$ is NP-complete. Therefore, no polynomial time algorithm is known for $\chi(G)$. In this paper we investigate several heuristics for graph coloring and a (sequential) backtracking algorithm for the exact value of $\chi(G)$. All heuristics considered are deterministic, they contain no random elements. For a survey of other heuristics see de Werra[7].

2 Contraction Algorithms

Contraction algorithms for graph coloring originate in a theorem of Zykov[8]. For non-adjacent vertices x and y we denote by $G/x, y$ the graph resulting from G by contracting y into x, i.e. y is deleteted and the neighborset $N_G(x)$ of x becomes $N_G(x) \cup N_G(y)$. Moreover, $G + xy$ is the graph with the additional edge xy.

Theorem 1.

$$\chi(G) \;=\; min\;\{\chi(G/x,y),\; \chi(G{+}xy)\}\;\; for\; non{-}adjacent\; vertices\; x\; and\; y\;.$$

Proof. Let $C(G)$ be the set of (proper) colorings of G and $|F|$ the number of colors used by $F \in C(G)$. Then we have

$$\begin{aligned} \chi(G) &= min\;\{|F| : \; F \in C(G)\} \\ &= min\;\{min\;\{|F| : \; F(x) = F(y)\},\; min\;\{|F| : \; F(x) \neq F(y)\}\} \\ &= min\;\{\chi(G/x,y),\; \chi(G + xy)\}. \end{aligned}$$ □

By multiple application of Theorem 1 graph G becomes the root of a binary tree of graphs, the *Zykov tree*. The construction of the Zykov tree is finished, when no further reduction according to Theorem 1 is possible. So the leaves of the Zykov tree are complete Graphs $G_i = (V_i, E_i)$. Theorem 1 implies $\chi(G) = min\;|V_i|$. This is the basis of an algorithm of Corneil and Graham[4] for $\chi(G)$, which searches through the Zykov tree in a depth-first manner. But despite some technical refinements this algorithm is inferior to the sequential backtracking algorithm described in section 3.

The Zykov tree is not uniquely determined. It depends on the order, in which non-adjacent pairs x, y are chosen. Each Zykov tree has exactly one branch which is exclusively generated by contractions. Let G_c be the complete graph at the end of this branch. The number of vertices, $|V(G_c)|$, of this graph is an upper bound for $\chi(G)$. The longer the branch to G_c is, the better will be this bound. So a good strategy for the selection of a non-adjacent pair $x,\ y$ to be contracted might be to retain as many non-adjacent pairs of vertices as possible. Equivalently, the resulting graph $G/x, y$ should have as few edges as possible. The difference of the number of edges $|E(G)| - |E(G/x, y)|$ equals the number $cn(x, y)$ of common neighbors of x and y. In the algorithm of Brigham and Dutton[2] non-adjacent vertices with a maximal number of common neighbors are contracted until a complete graph is achieved.

More effective and yielding better results than the algorithm of Brigham and Dutton is the algorithm RLF (Recursive-Largest-First) of Leighton[6], see also Hertz[5]. Here x is a fixed vertex of maximal degree. Non-neighbors y with a maximal number of common neighbors with x are contracted into x until x is adjacent to every other vertex. To increase effectiveness, x is then removed and a new vertex of maximal degree in the residual graph is chosen. Vertex x and all vertices contracted into x constitute a color class. If x has a non-neighbor, but no non-adjacent vertex with a common neighbor, then the graph is unconnected and x is adjacent to all other vertices of its own component. To enforce the RLF-principle consequently, we must continue the construction of the color class of x with a vertex of maximal degree in an other component. This has been taken into account in our implementation of the algorithm. The (worst-case time-)complexity of RLF is $O(|V|^3)$. One factor $|V|$ is due to the determination of a vertex x of maximal degree. Traversing the non-neighbors of x in search for a vertex y with a maximal number of common neighbors with x may cost another $O(|V|^2)$ elementary operations. We propose the following informal code for the RLF-algorithm.

Given a graph $G = (V, E)$ with vertex set $V = V(G)$ and edge set E.
A coloring $F : V \longrightarrow \mathbb{N}$ is established and will be returned.

$colornumber = 0$; //number of used colors

while ($|V(G)| > 0$){

  determine a vertex x of maximal degree in G;
  $colornumber = colornumber + 1$;
  $F(x) = colornumber$;
  NN= set of non-neighbors of x;
  while ($|NN| > 0$){ //find $y \in NN$ to be contracted into x
    $maxcn = -1$; //becomes the maximal number of common neighbors
    $ydegree = -1$; //becomes $degree(y)$
    for every vertex $z \in NN${
      cn=number of common neighbors of z and x;
      if ($cn > maxcn$ or ($cn == maxcn$ and $degree(z) < ydegree$)){
        $y = z$;

```
            ydegree = degree(y);
            maxcn = cn;
         }
      }
      if (maxcn == 0){                         //in this case G is unconnected
         y=vertex of maximal degree in NN;
      }
      F(y) = colornumber;
      contract y into x;
      update the set NN of non-neighbors of x;
   }
   G = G − x;                                              //remove x from G
}
return F.
```

3 Sequential Coloring

One of the simplest coloring methods is sequential coloring $SC(O)$ according to a given order $O = [a_0, ..., a_{n-1}]$ of the vertices:

$F(a_0) = 1$.
If $a_1, ..., a_{i-1}$ $(i \geq 1)$ have already received colors, let $F(a_i)$ be the smallest color not yet used in the neighborhood of a_i.

The LF-algorithm (Largest First) is based on an ordering of the vertices of the graph $G = (V, E)$ according to non-increasing degrees. Its complexity is $O(|V|^2)$.

The algorithm DSATUR (Degree of Saturation) of Brèlaz[1] is a sequential coloring algorithm with a dynammically established order of the vertices. Suppose F is a partial coloring of the vertices of G. The degree of saturation of a vertex x, $deg_s(x)$, is the number of different colors at the vertices adjacent to x. DSATUR starts by assigning color 1 to a vertex of maximal degree. The vertex to be colored next in the sequential coloring procedure of DSATUR is a vertex x with maximal $deg_s(x)$. The complexity of DSATUR is $O(|V|^3)$.

Definition 1. *Colorings F_1, F_2 of $G = (V, E)$ are equivalent, if they induce the same partition of the vertex set V into color classes.*

Definition 2. *A coloring F of the vertices $a_0, ..., a_{n-1}$ of the graph G is tight with respect to the given order, if*

$$F(a_i) \leq colors(i-1) + 1 \; for \; all \; i = 0, 1, ..., n-1,$$

where $colors(j)$ denotes the number of different colors at the vertices $a_0, ..., a_j$, resp. $colors(-1) = 0$.

A backtracking sequential coloring algorithm, which returns the exact value of $\chi(G)$, was first developed by Brown[3]. Many unnecessary branches of the search tree can be avoided by the following theorem.

Theorem 2. *Every coloring $F : V \longrightarrow \mathbb{N}$ of the vertices $a_0, ..., a_{n-1}$ of the graph $G = (V, E)$ is equivalent to a tight coloring with respect to the given order of vertices.*

Proof. We change F to an equivalent, tight coloring by renaming colors.

Suppose $F(a_0) = k_1 > 1$. If no vertex has color 1, we change color k_1 to color 1, else we interchange colors k_1 and 1.

Assume that we have already made F tight for all vertices a_0, ... , a_{i-1}, $i \geq 1$. Then exactly the colors $1, 2, ..., l = colors(i-1)$ appear at the vertices $a_0, ..., a_{i-1}$.

Suppose $F(a_i) = k > l+1$. If no vertex has color $l+1$, we change color k to color $l+1$, else we interchange colors k and $l+1$. □

The original algorithm of Brown[3] was improved by Brèlaz[1]. The vertices of the graph are stored in an array A. Initially they are ordered according to non-increasing degrees. The order is dynammically changed. Suppose $A[0], ..., A[i-1]$ have already been colored. The number of different colors at these vertices is $colors(i-1) = l_i$. The set of free colors at $x = A[i]$, $U = freeColors(x)$, is the subset of colors in $\{1, 2, ..., l_i + 1\}$, which are not present in the neighborhood of x. If an upperbound $optColorNumber$, $\chi(G) \leq optColorNumber$, has been established by a coloring F, all colors $\geq optColorNumber$ can be removed from U. The vertex to be colored next is as in DSATUR a vertex of maximal degree of saturation. It is colored with the smallest color in U. If U is empty, a backtrack is executed.

BSC (Backtracking Sequential Coloring) algorithm for the determination of the chromatic number $\chi(G)$ of the graph $G = (V, E)$.

```
Find an ordering A = [a_0, ..., a_{n-1}] of the vertices acording to non-increasing
degrees;
start = 0;                                                    //starting index
optColorNumber = |V| + 1;                           //optimal number of colors
x = A[0];                                        //current vertex to be colored
colors(-1) = 0;            //colors(j) =number of colors at A[0], ..., A[j]
U = [1];                   //Variable for the set(sequence) of free colors
freeColors(x) = U;                                   //set of free colors of x
while (start ≥ 0){
                       //x is colored in the following for-loop. Backtracking is
              //necessary, if U = ∅ or if an improved coloring has been found
  back = false;                            //boolean variable for backtracking
  for (i = start; i < |V|; i++){
    if (i > start){              //for i = start x and U are already available
      find an uncolored vertex x of maximal degree of saturation;
      U =set of free colors of x, which are < optColorNumber;
      sort U non-decreasing;
    }
    if (|U| > 0){
      k = U[0];                                          //selected free color
      F(x) = k;                                             //current coloring
      remove k from U;
      freeColors(x) = U;
      l = colors(i - 1);
      colors(i) = max{k, l};
    }
    else{                                       //U = ∅, backtrack one position
      start = i - 1;
      back = true;
      break;                                            //leaving the for-loop
    }
  }
  if (back){
    if (start ≥ 0){
      x = A[start];                                      //new starting vertex
      uncolor x;
```

```
            U = freeColors(x);
        }
    }
    else{   //in this case the above for-loop has been passed without a break
        Fopt = F;                              //storing the currently optimal coloring
        optColorNumber = colors(|V| − 1);
        i =least index with F(A[i]) = optColorNumber;
        start = i − 1;
        if (start < 0){
            break;                                              //leaving the while-loop
        }
        uncolor all vertices A[i] with i ≥ start;
        for (i = 0; i ≤ start; i + +){
            x = A[i];
            U = freeColors(x);
            remove from U all colors ≥ optColorNumber;
            freeColors(x) = U;          //the current coloring is to be improved
        }               //notice: here we have x = A[start], U = freeColors(x)
    }
}
return Fopt.
```

The algorithm can be improved by some refinements. If we start the ordering A of the vertices with a clique on the vertices $A[0], ..., A[r-1]$, these vertices receive fixed colors 1,...,r. The algorithm can be stopped, if backtracking leads to an index $< r$. A lower bound $lb \leq \chi(G)$ may be established by applying BSC to a dense subgraph. The algorithm may then be finished, if a coloring is found, which uses no more than lb colors. We call a vertex x a *merging vertex* with respect to a free color k of x, if all uncolored neighbors of x are adjacent to a vertex with color k. In this case we may reduce $freeColors(x)$ to $\{k\}$.

From BSC we gain the heuristic IBSC(k) by restricted backtracking (Incomplete Backtracking Sequential Coloring), where each vertex may only be k-times the new starting vertex after backtracking. For $k \geq 1$ the complexity of IBSC(k) is $O(k|V|^4)$.

4 Runtime Comparisons

We have generated 100 random graphs on 60 vertices for each edge-density 0.1, 0.3, 0.5, 0.7 resp. 0.9. For the algorithms LF, DSATUR, RLF, IBSC(1), IBSC(2), IBSC($|V|$) and BSC we display the average number of colors used and (in brackets) the average running time in ms.

edge-density	0.1		0.3		0.5	
LF	4.86	(0.94)	9.09	(1.32)	13.59	(1.92)
DSATUR	4.18	(1.84)	8.21	(2.89)	12.50	(4.62)
RLF	4.17	(10.13)	7.96	(11.95)	11.94	(12.06)
IBSC(1)	4.00	(2.99)	7.57	(27.38)	11.52	(45.68)
IBSC(2)	3.99	(3.00)	7.39	(43.04)	11.39	(72.17)
IBSC($\|V\|$)	3.99	(3.14)	7.10	(267.76)	11.00	(1107.75)
BSC	3.99	(5.14)	7.02	(485.65)	10.67	(19632.51)

edge-density	0.7		0.9	
LF	19.24	(1.86)	28.52	(2.03)
DSATUR	18.08	(5.82)	27.49	(8.25)
RLF	17.18	(13.05)	27.26	(13.26)
IBSC(1)	16.83	(53.87)	26.23	(43.32)
IBSC(2)	16.66	(83.95)	26.20	(61.63)
IBSC($\|V\|$)	16.08	(1382.24)	25.84	(424.18)
BSC	15.53	(39996.20)	25.80	(1154.61)

The results show that LF, the fastest algorithm on our list, yields poor quality. Its application makes sense only for very large graphs. RLF should be applied only for dense graphs, otherwise too many contractions consume too much time. Incomplete backtracking in a simple form, e.g. IBSC(1), may be a good alternative to RLF or DSATUR.

In some situations the different performances of the algorithms can be made more striking, if the coincidences of the number of colors used with the chromatic number are counted. We applied the algorithms to 100 random graphs on 120 vertices with edge-density 0.1. The number of direct hits of the chromatic number is shown.

LF: 0, DSATUR: 5, RLF: 4, IBSC(1): 16, IBSC(2): 19, IBSC($|V|$): 61.

References

[1] Brèlaz, D., *New methods to color the vertices of a graph*, Communications of the Assoc. of Comput. Machinery 22 (1979), 251-256.

[2] Brigham, R. D. and Dutton, R. D., *A new graph coloring algorithm*, The Computer Journal 24 (1981), 85-86.

[3] Brown, J. R., *Chromatic scheduling and the chromatic number problem*, Management Science 19 (1972), 456-463.

[4] Corneil, D. G. and Graham, D., *An algorithm for the chromatic number of a graph*, SIAM J. Comput. 2 (1973), 311-318.

[5] Hertz, A., *A fast algorithm for coloring Meyniel graphs*, Journal of Combinatorial Theory B 50 (1990), 231-240.

[6] Leighton, F. T., *A graph coloring algorithm for large scheduling problems*, Journal of Research of the National Bureau of Standards 84 (1979), 489-503.

[7] de Werra, D., *Heuristics for Graph Coloring*, Computational Graph Theory, Comput. Suppl. 7, Springer, Vienna (1990), 191-208.

[8] Zykov, A. A., *On some properties of linear complexes*, Amer. Math. Soc. Translations 79 (1952), p. 81.