An Experimental Evaluation of List Coloring Algorithms

Patrick Healy and Andrew Ju

Computer Science Department,
University of Limerick,
Limerick, IRELAND
{patrick.healy, andrew.ju}@ul.ie

Abstract. The list coloring problem is a variant of vertex coloring where a vertex may be colored only a color from a prescribed set. Several applications of vertex coloring are more appropriately modelled as instances of list coloring and thus we argue that it is an important problem to consider. Regardless of the importance of list coloring, few published algorithms exist for it. In this paper we review the only two existing ones we could find and propose an exact branch and bound one. We conduct an experimental evaluation of the three algorithms.

Keywords: list coloring, branch and bound, exact graph algorithm

1 Introduction

In the classical vertex coloring problem one asks if one may color the vertices of a graph $G$ with one of $k$ colors so that no two adjacent vertices are similarly colored; the corresponding optimization problem seeks to find the minimum value $k$ for a graph that admits a legal $k$ coloring. In 1976 Vizing [13] proposed an additional restriction on the coloring by supplying for each vertex a list of permissible colors.

Many problems that rely on vertex coloring might be modelled more appropriately using list coloring. For example, exam timetabling is frequently modelled as a vertex coloring problem where graph edges represent subjects that may not be scheduled simultaneously. Other constraints, such as preference for the times an exam may be scheduled, are often considered to be soft constraints [11]. By supplying a list of (in)appropriate hours for each exam one may model the problem more accurately as an instance of list coloring. Similarly, the frequency assignment problem for cellular telephone networks and WLANs may be modelled more accurately by restricting the coloring of vertices (transmitters or routers) to a specified set [3].

Closely related to the list coloring problem is the weighted vertex coloring or vertex multicoloring problem [10,6]. In this problem each vertex has a weight associated with it and the graph $G$ must be multicolored so that vertex $v$ is assigned a set of colors $C_v$ where $C_u \cap C_v \forall (u, v) \in E(G)$; the objective is to find $\chi(G, w)$, the minimum number of colors required to color $G$ and that satisfies
the vertex weight requirement. When, further, each vertex is supplied with a list of permissible colors one arrives at the list multicoloring problem \[2\].

Clearly the list coloring problem is as hard as vertex coloring, for the latter reduces to the former (in polynomial time) through supplying, for every vertex, all colors as its permissible list. Few published algorithms exist for the list coloring problem in its most general form. In the context of frequency assignment Borndörfer et al. present several heuristics that incorporate problem-specific requirements. Likewise with Garg et al. \[8\] though their interest is in developing a distributed solution.

In this paper we investigate the performance of three list coloring algorithms. The first we consider is the greedy, random algorithm \(k\)-GL (Greedy List) proposed by Achlioptas and Molloy \[1\]. Our second algorithm from the literature is a maximal independent set-based heuristic algorithm \[12\]. Finally we propose a new exact branch-and-bound algorithm ELC. While the running time of the latter cannot be expected to be competitive with the former two it does provide a useful reference point against which one may consider their performance.

In the following section we describe the three algorithms we have implemented and the context of our experiments. Following that, in Section 3 we provide the outcome of our experiments. In Section 4 we conclude the paper and suggest areas of further research.

**2 Experimental Context**

We implemented two algorithms from the literature as well as a newly developed branch-and-bound algorithm. Of the two previously published algorithms one is randomised and the second is deterministic; both are heuristic. We describe these algorithms in the following sections.

**2.1 The \(k\)-GL algorithm**

Achlioptas and Molloy propose a greedy algorithm they call \(k\)-Greedy-List or \(k\)-GL \[1\]. Each vertex is supplied with a permissible color list \(L_v = \{1, 2, 3, \ldots, k\}\) (the contiguous sequence of integers between 1 and \(k\)) in order to facilitate their analysis. However, it would not be difficult to modify the algorithm in order to cater for a) lists of varying lengths and, b) non-contiguous sequences.

The algorithm proceeds by picking a vertex \(v\) that is deemed most critical as measured by the number of remaining colors on its permissible list, \(L_v\), with ties broken randomly. If set \(L_v\) is not empty, a color randomly chosen from it is assigned to \(v\) and since that color can no longer be used in \(v\)'s neighborhood it is removed from each neighbor's permissible list (line 10); if \(L_v\) is empty, then the algorithm fails in finding a solution.

The algorithm given below in Algorithm 1 is a modification of the original \[1\] so that it accepts lists of non-contiguous sequences. We discuss its performance in Section 3.1.
Algorithm 1 *k*-Greedy-List (*k*-GL)

1: Initialize: $G = (V, E)$, and $L_v = \{i_1, \ldots, i_k\}$ for every $v \in U$, $|L_v| = k$.

2: while $G \neq \emptyset$ do

3: Pick a vertex $v$ uniformly at random from $\{u \in V : |L_u| \text{ is smallest}\}$.

4: if $L_v \neq \emptyset$ then

5: assign a color $w$ chosen uniformly at random from $L_v$ to $v$.

6: else

7: exit(fail)

8: end if

9: for each $u \in V$ that is adjacent to $v$ do

10: set $L_u = L_u \setminus \{w\}$

11: end for

12: set $V = V \setminus \{v\}$

13: end while

14: return

2.2 The *LC* algorithm

Satratzemi and Tsouros propose a deterministic heuristic algorithm \cite{12} that centres around finding a maximal independent set at each iteration. Since the graph induced by an independent set (IS) is edgeless it may be colored with a single color.

The algorithm proceeds by determining $Q_i$, the set of vertices that can be colored $i$ ($v \in Q_i \Leftrightarrow i \in L_v$). Then amongst all those $Q_i$s a search is made for $S$, the largest maximal independent set (l. 10; code not shown); the vertices in $S \subseteq Q_j$ are colored $j$ and color $j$ is removed from $LL$ ($LL = \cup_{i=1}^{n} L_i$). Data structures are then updated appropriately. The algorithm fails in finding a solution if $LL$ is empty when there are still uncoloured vertices. The algorithm makes little effort to compute a “good” maximal independent set, simply performing a linear scan over the elements of $Q_i$. There are three nested loops (one not shown) with each iterating over, at worst, the set of vertices and set of colors, and thus the overall running time is $O(n^3)$.

The algorithm outline is given below in Algorithm 2. We report on its performance in Section 3.1.

2.3 ELC – An Exact Algorithm

We developed an exact branch-and-bound algorithm, ELC, as shown in Algorithm 3. With sufficient time ELC will find the minimum coloring subject to the supplied list constraints for each vertex. In previous work on the vertex coloring problem \cite{9} we have determined that it is advantageous to manage the vertices that remain to be colored in a priority queue for sparse graphs. Assuming that the priority queue $PQ$ has been initialised with all vertices the function $\text{color()}$ selects the “next up” (most critical) vertex and considers all of its permitted colors in turn. For each such assigned color the priority queue $PQ$ and associated data structures are updated and the process recurs on the remaining uncoloured
Algorithm 2 \textit{LC} – heuristic

1: Initialize: $G = (V, E)$, and $L_v = \{1, ..., k\}, |L_v| = k$ for every $v \in V$, $LL$ is list of all available colors.
2: $ncol \leftarrow z \leftarrow 0, G_w \leftarrow G$
3: while (true) do
4:   for (each $i \in LL$) do
5:     $Q_i = \{q_{i1}, q_{i2}, ...\}$ where $q_j$ such that $i \in L(q_j)$
6:     Order elements of $Q_i$ such that $deg_{G_w}(q_{ij}) \leq G_w(q_{ij+1})$
7:   end for
8:   $mc \leftarrow 0, S = \emptyset, LT \leftarrow LL$
9:   while ($LT \neq \emptyset$) do
10:      select $i \in LT$, form an independent set $KEEP$
11:      if ($|KEEP| > mc$) then
12:         $mc \leftarrow |KEEP|, S \leftarrow KEEP, m \leftarrow i$
13:      end if
14:      $LT \leftarrow LT - \{i\}$
15:   end while
16:   $z \leftarrow z + 1$
17:   assign color $m$ to nodes in set $S$
18:   $ncol \leftarrow ncol + |S|, LL \leftarrow LL - \{m\}$
19:   if ($ncol == |V|$) then
20:      print $z$
21:   end if
22: end while
23: Update $LL, Q_i, i \in LL, G_w$
24: if $LL == \emptyset$ then
25:   print NO SOLUTION
26: end if
27: end while

vertices. If $PQ$ is empty the search has reached a leaf node of the search tree, every vertex has been colored, and a feasible coloring has been found.

Otherwise, if the number of colors used to date exceeds that of a previous feasible solution (named UB) then the search path is abandoned and another color possibility is examined. If the number is less, then a new upper bound is determined. This upper bound is used to prune the search tree if the total colors used in the current coloring process is more than or equal to the current upper bound. This makes ELC a branch and bound algorithm.

In the following section we discuss the heuristics that we have investigated for determining proper upper bounds and also describe supporting data structures that we have determined to be useful in a branch-and-bound setting. We evaluated its performance in Section 3.2.

Branch-and-bound Issues At each step, ELC picks the “next up” vertex from the uncolored subgraph and colors it accordingly. It is clear that with a 'proper'
Algorithm 3 ELC algorithm

main()

Require: Graph $G = (V, E)$ and color availability list $LL$ as input
1: read $G$ and $LL$, and store in pset form, initialise $PQ$, $UB = |LL|$
2: call color() function
3: STOP

color()

1: if (current_color $\geq UB$) or
2: ($PQ$ is empty) or
3: (exceed time limitation) then
4: return
5: end if
6: set colorflag = 0
7: $v = PQ.delete_{\text{max}}()$ // the next up vertex
8: for each available color $i \in LL[v]$ do
9: if vertex $v$ can be assigned with color $i$ then
10: assign vertex $v$ with color $i$
11: update $PQ$
12: set colorflag = 1
13: if (thiscol $= ELC() < UB$) then
14: $UB = thiscol$ // new upper bound
15: end if
16: uncolor vertex $v$ and update $PQ$
17: if $UB \leq current\_color$ then
18: return
19: end if
20: end if
21: end for
22: if (colorflag == 0) then
23: return $-1$
24: else
25: return $UB$
26: end if

coloring order, the algorithm will arrive at a tighter UB earlier. Therefore, a proper strategy shall be deployed in $PQ$, so a more proper vertex could be always in the root node.

In the color() function of ELC, prior to implementing the actual exact algorithm, various heuristics for finding proper upper bounds were investigated. An initial thought would be to embed either the LC heuristic or the $k$-GL heuristic (both reported on in Section 3.1) in the branch-and-bound search algorithm. However, it is clear that the LC heuristic cannot be incorporated in the algorithm since, by its nature, it cannot guarantee all possible colorings have been considered. For $k$-GL, randomisation can be avoided and a systematic search conducted, if the selection of vertices is done lexicographically (line 3 in Algo-
Algorithm 4 DSATUR-h heuristic

1: Initialise: $G = (V, E)$ as input.
2: Pick a vertex $v \in V$ where $\text{deg}(v)$ is maximal;
3: Assign color 1 to vertex $v$; set $V = V \setminus \{v\}$.
4: Pick a vertex $u$ from $V$ where $\text{deg}_s(u)$ is maximal, pick the one with larger $\text{deg}_u(u)$ in case of a tie;
5: Assign the lowest numbered color to vertex $u$; set $V = V \setminus \{v\}$.
6: if $(V = \emptyset)$ then
7:    STOP
8: else
9:    return to step 4.
10: end if

We also considered a very effective yet simple heuristic for vertex coloring, DSATUR-h as given in Algorithm 4. We denote $\text{deg}(v)$ in the algorithm as the degree of vertex $v$ in graph $G$, $\text{deg}_s(v)$ as the saturation degree – the number of different colors to which vertex $v$ is adjacent – and $\text{deg}_u(v)$ as the number of vertices in the uncoloured subgraph of $G$ to which vertex $v$ is adjacent. Based on the DSATUR-h heuristic, we implemented a variant of ELC algorithm - ELC-d, and the performance is shown in Table 3.

Further, in the cases when algorithm LC does find a feasible solution, we have observed that the quality is very good. Thus, we embedded it in our ELC-d algorithm to initialise the UB, and to prune the search tree at each step once the total colors used in current coloring exceed this upper bound. We name this algorithm the ELC-l and its performance is shown in Table 3.

Supporting Data Structures

The pset

During the course of the algorithm as vertices get colored (and, when backtracking, uncolored) we need to know frequently the uncolored neighbors of an identified vertex; similarly we need to know the vertices that have been colored an identified color. The two data structures to represent sets like these that immediately come to mind are the adjacency list and adjacency vector. While a list supports easy insertion and efficient iteration over exactly those elements in the list, testing for membership is less efficient. Conversely, a vector permits efficient membership testing and insertion/removal though iteration over the elements is less efficient. Through the use of the pset we may achieve the positive characteristics of both the adjacency list and vector.

The pset is defined on the domain of integers $\{0, \ldots, n-1\}$. A pset comprises an array, $v$, of $n$ integers which at all times represents a permutation
of the integers \(\{0, \ldots, n-1\}\). A marker, \(m\), is maintained that specifies that the first \(m\) elements are in the partition while the remainder are not. A final array, called \(1\), that is indexed by the data in \(v\), (namely the values \(\{0, \ldots, n-1\}\)) indicates the location, or index, of a value in \(v\). Since the domain and range of both arrays are the same, then from the definition of \(1\) clearly the two arrays act as inverses of each other: \(v[1[i]] = i\) and \(1[v[j]] = j\). Given an element \(x\) of the set, \(x\) is then in the partition if and only if \(0 \leq 1[x] < m\), so membership can be tested in constant time. An element \(x\) can be removed from the partition by finding its location, \(i=1[x]\) in \(v\), swapping elements \(i\) and \(m-1\) of \(v\) (and \(1\)) and, finally, decreasing \(m\) by 1; insertions can be performed analogously. Thus, for a cost of \(2n+1\) units of storage per vertex we can maintain a data structure that permits constant-time insertions, removals and membership tests as well as efficiently iterating over the elements. (Although this is hardly new we have not been able to find a reference to a data structure of this type in the literature.)

This data structure can be used to maintain the (sub)set of feasible colors for a vertex or to maintain its uncolored neighborhood. As a generalisation of the latter it can also be used to simulate the adjacency list associated with a vertex.

**The Priority Queue**

One of the crucial issues for a branch-and-bound algorithm is efficient pruning of the search tree. In order to achieve this we must prioritise vertices that remain to be colored and at each call of the recursion efficiently retrieve the vertex of highest priority for coloring next; we call this vertex the ‘next up’. This suggests some type of ordered queue whereby we select the front element of the queue at each iteration. The priority queue abstract data type classically provides for the \(\text{insert()}\) and \(\text{delete}\_\text{max()}\) functions, each achievable in \(\mathcal{O}(\log n)\)-time [7]. Thus, this appears to be an attractive alternative to an iteration over all uncolored vertices. Our situation makes further demands, however. After making a next up selection and assigning it a color the uncolored degree of its uncolored neighbors will decrease by 1 and the saturation degree of each may increase by 1. Thus, each neighbor needs to be repositioned in the priority queue. Prior to being recolored, at the backtracking step this vertex will need to be uncolored, requiring the undoing of the above. With the appropriate underlying data structure the priority adjusting operations \(\text{decrease}\_\text{p()}\) and \(\text{increase}\_\text{p()}\) are still achievable in \(\mathcal{O}(\log n)\)-time [7]. We do this by representing the priority queue as a heap [7] with an additional array that provides us with a handle to the index of any vertex similar to the look-up array, \(1\), used in \text{pset}.

When a vertex is selected and assigned a color the priorities of all remaining uncolored vertices in the queue must be updated. It is clear that the set of vertices whose priorities might change is limited to those in \(N_U(v)\), the uncolored neighborhood of the selected vertex, \(v\). For sparse graphs it seems reasonable to make a separate call to \(\text{adjust}\_\text{p()}\) for each \(u \in N_U(v)\), adjusting its priority independently, for a total effort of \(\mathcal{O}(|N_U(v)| \log n)\).

Note that as a consequence of a vertex being selected and assigned a color it is possible either for an uncolored neighbor’s priority in the queue to increase or
decrease. For example, if \( v \) is the next up vertex and it is assigned color \( c \) then if there is an uncolored neighbor \( w \) of \( v \) that did not have \( c \) in its neighborhood the saturation degree of \( w \) will increase, leading to an increase in its priority. If, on the other hand, \( w \) already had \( c \) in its neighborhood prior to \( v \) then there will be no change to its saturation degree; but since its uncoloured neighborhood has been reduced its priority now decreases. However, the change in priority does not guarantee a change in its position in the queue.

3 Experimental Evaluation

To facilitate a preliminary experimental analysis of the algorithms it is reasonable that we focus on sparse graphs and small list lengths.

All tested graphs and the color lists were randomly generated. In our experiments we generated graphs with \( |V| = \{50, 100, 200\} \) vertices and for each we randomly generated edges (uniformly) between vertices so that the density of edges, \( d \) was one of \( \{0.05, 0.1, 0.2\} \). When assigning color restrictions to each vertex we considered lists restricted to three different color ranges. That is, the randomly assigned colors were drawn from the range of colors \([1, c|V|]\) where \( c = \{1.00, 0.50, 0.25\} \). Finally, for each color range we considered three list lengths \( k = \{5, 4, 3\} \).

Thus 81 problem instances were generated, each a unique \((|V|, d, c, k)\) tuple, and the three algorithms were run on each instance. For the sizes under consideration both of the heuristic algorithms ran in negligible time. We terminated the branch-and-bound algorithm after 10mins (approx.) of elapsed time. The solution quality of each are reported in the following Table 1.

The implementation of all algorithms were written in C++ and were complied using g++-4.2.1. All experiments were conducted on a Macbook Pro running OS X version 10.9, 2.5GHz Intel Core i7 processor with 8GB of DDR3 memory (clock speed 1333MHz).

3.1 Evaluation of the Heuristics

Table 1 below compares the solution quality of the two heuristic algorithms, \( k\)-GL and LC. In view of the performance of each algorithm, not surprisingly, \( k\)-GL provides a poor solution quality than \( LC \) if the latter finds a feasible solution. But it’s surprising to see that \( k\)-GL always succeed in finding a feasible solution over the 81 instances in a single run while \( LC \) fails with smaller color ranges. More surprising, though, is the poor showing of LC, the independent set-based heuristic. Of the 81 instances considered the algorithm failed to find a solution (shown as “–” below) 42 times. These failures, generally, correlate with smaller color ranges. This is discussed further in Section 3.1.

Satratzemi and Tsouroς [12] provided data on a set of experiments they ran and our figures correspond to theirs in the common cases generally. Also, our results confirm the trends they reported, namely
Table 1. Preliminary comparison of two heuristic algorithms LC and k-GL.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |

Table 1. Preliminary comparison of two heuristic algorithms LC and k-GL.

- for greater value of list cardinality, \( k \), a smaller number of colors are required to list color a graph, generally;
- for smaller graph densities, a smaller number of colors is needed to color the graph;
- in general fewer colors are needed when the color range becomes smaller.

We conducted another experiment to investigate these phenomena. The number of failures of LC to find a solution over 15 runs is illustrated in Table 2. We considered graphs with \(|V| = \{50, 100, 200\}\) vertices and for each we randomly generated edges (uniformly) between vertices so that the density of edges, \( d \) was one of \{0.01, 0.015, 0.02, 0.025, 0.03\}. When assigning color restrictions to each vertex we considered lists restricted to three different color ranges. That is, the randomly assigned colors were drawn from the range of colors \([1, c|V|]\) where \( c = \{1.00, 0.50, 0.25, 0.20\}\).

The results in the table suggest that for smaller values of list cardinality, there is a higher chance that the IS-based heuristic will fail in finding any feasible solution. Meanwhile from Table 1 we can see that LC fails in finding solutions
for 8 out of 9 instances in each category when $c = 0.25$. Therefore, in tandem with Table 2, we determine that $c = 0.2$ is the threshold of when the original $LC$ heuristic shall no longer be adopted.

| $c/|V|$ | 50  | 100 | 200 |
|--------|-----|-----|-----|
| 0.20   | 15/15 | 14/15 | 14/15 |
| 0.25   | 13/15 | 12/15 | 14/15 |
| 0.50   | 12/15 | 8/15  | 4/15  |
| 1.00   | 1/15  | 0/15  | 0/15  |

Table 2. When $LC$ fails to find a solution.

### 3.2 ELC Evaluation

Table 2 below compares the solution quality of the two variants of our exact ELC algorithm, ELC-$d$ and ELC-$l$. We considered graphs with $|V| = \{50, 100, 200\}$ vertices and for each we randomly generated edges (uniformly) between vertices so that the density of edges, $d$ was one of $\{0.05, 0.1, 0.2\}$. When assigning color restrictions to each vertex we considered lists restricted to three different color ranges. That is, the randomly assigned colors were drawn from the range of colors $[1, c|V|]$ where $c = \{1.00, 0.50, 0.25\}$. Finally, for each color range we considered three list lengths $k = \{5, 4, 3\}$.

Again, this gave rise to 81 problem instances being generated, each a unique $(|V|, d, c, k)$ tuple; the three algorithms were run on each instance. For the sizes under consideration both of the heuristic algorithms ran in negligible time. We terminated the branch-and-bound algorithm after 10mins (approx.) of elapsed time. The solution quality of each are reported in the following table.

The table also confirms the trends that appear in the heuristic algorithms in Table 1, namely

- for greater values of list cardinality, $k$, a smaller number of colors are required to list color a graph, generally;
- for smaller graph densities, a smaller number of colors is needed to color the graph;
- in general fewer colors are needed when the color range becomes smaller.

In the table, ELC-$l$ uses the LC heuristic first to initialise the UB. If LC fails in finding a heuristic solution, no tighter UB can be initialised and in these cases ELC-$l$ will perform the same as ELC-$d$ (shown as “–” below).

It is quite surprising to observe that neither ELC-$l$ nor ELC-$d$ can prove optimality even for the graphs with $|V| = 50$, and not even to mention the case of $|V| = 100$. To facilitate the improvement of exact algorithms, it is reasonable, \footnote{In Section 2.3 we mentioned a variant of ELC, ELC-$rgl$, whose performance is not shown in the table since in no instance was it a winner in comparison of the ELC-$d$.}
we suggest, that further research focus on $|V| = 50$ and use $|V| = 100$ as a threshold to narrow down the area.

### 4 Conclusions

We have implemented and investigated the only two existing list coloring algorithms known to us in the literature and have proposed an exact branch-and-bound algorithm with additional data structure support through the use of a priority queue. The performance of each has been investigated and reported on. Further algorithm tuning opportunities exist: with the use of the LC heuristic the ELC-l algorithm can get a tighter upper bound, but the performance cannot be improved significantly as expected; we suspect the branching (or vertex selection) strategy needs to be improved by conducting an initial partial coloring. This, and also identifying other opportunities for code optimization are our next priorities.

| $|V|$ | $d$ | $e$ | $k$ | ELC-l | LC | ELC-d | d | ELC-l | LC | ELC-d | d |
|------|-----|-----|-----|-------|-----|-------|---|-------|-----|-------|---|
| 50   | 0.05| 1.00| 5   | 11    | 14   | 0.1   | 14 | 22    | 0.2 | 16    | 17   | 21   |
|      | 4   |     |     | 13    | 24   |       | 16 |       |     | 16    |     |     |
|      | 3   |     |     | 16    | 18   |       | 17 |       |     | 19    |     |     |
| 0.50 | 5   |     |     | 7     | 8    | 11    | 8  | 9     | 13  | -     | -    | 18   |
|      | 4   |     |     | -     | -    | 12    | -  | 16    | 13  | 13    | 15   |      |
|      | 3   |     |     | 11    | 11   | 12    | -  | 12    | -   | -     | -    | 14   |
| 0.25 | 5   |     |     | -     | -    | 5     | -  | 5     | -   | -     | -    | 7    |
|      | 4   |     |     | -     | -    | 8     | -  | 6     | -   | -     | -    | 8    |
|      | 3   |     |     | -     | -    | 6     | 7  | 10    | 7   | -     | -    | 8    |
| 100  | 0.05| 1.00| 5   | 24    | 37   | 0.1   | 24 | 52    | 0.2 | 27    | 51   |      |
|      | 4   |     |     | 27    | 47   |       | 30 | 46    |     | 36    | 55   |      |
|      | 3   |     |     | 32    | 52   |       | 37 | 57    |     | 38    | 57   |      |
| 0.50 | 5   |     |     | -     | -    | 32    | -  | 39    | -   | -     | -    | 38   |
|      | 4   |     |     | 19    | 31   |       | 22 | 30    | -   | -     | -    | 43   |
|      | 3   |     |     | 22    | 32   |       | 25 | 31    |     | 31    | 45   |      |
| 0.25 | 5   |     |     | -     | -    | 18    | -  | 17    | -   | -     | -    | 25   |
|      | 4   |     |     | -     | -    | 21    | -  | 19    | -   | -     | -    | 23   |
|      | 3   |     |     | -     | -    | 18    | -  | 22    | -   | -     | -    | 25   |
| 200  | 0.05| 1.00| 5   | 49    | 102  | 0.1   | 50 | 118   | 0.2 | 60    | 121  |      |
|      | 4   |     |     | 56    | 103  |       | 60 | 118   |     | 68    | 125  |      |
|      | 3   |     |     | 68    | 123  |       | 71 | 113   | -   | -     | -    | 116  |
| 0.50 | 5   |     |     | -     | -    | 76    | -  | 65    | -   | -     | -    | 81   |
|      | 4   |     |     | -     | -    | 69    | 45 | 83    | -   | -     | -    | 90   |
|      | 3   |     |     | -     | -    | 73    | -  | 79    | -   | -     | -    | 83   |
| 0.25 | 5   |     |     | -     | -    | 35    | -  | 29    | 42  | -     | -    | 45   |
|      | 4   |     |     | -     | -    | 38    | -  | 50    | -   | -     | -    | 50   |
|      | 3   |     |     | -     | -    | 47    | -  | 50    | -   | -     | -    | 50   |

**Table 3.** Comparison of ELC-l and ELC-d.
Acknowledgments

The research is jointly funded by the Irish Research Council (previously known as Irish Research Council for Science, Engineering and Technology, RS/2011/5), and China Scholarship Council (CSC).

References

1. Dimitris Achlioptas and Michael Molloy. The analysis of a list-coloring algorithm on a random graph (extended abstract). In 38th Annual Symposium on Foundations of Computer Science, pages 204–212, 1997.
2. Yves Aubry, Jean-Christophe Godin, and Olivier Togni. Vectorial solutions to list multicoloring problems on graphs. CoRR, abs/1202.4842, 2012.
3. R. Borndörfer, A. Eisenblätter, M. Grötschel, and A. Martin. Frequency assignment in cellular phone networks. Annals of Operations Research, 76(0):73–93, 1998.
4. D. Brélaz. New methods to color the vertices of a graph. Communications of the ACM, 22(4):251–256, 1979.
5. Massimiliano Caramia and Paolo Dell’Olmo. Solving the minimum-weighted coloring problem. Networks, 38(2):88–101, 2001.
6. Massimiliano Caramia and Jirí Fiala. New lower bounds on the weighted chromatic number of a graph. Discussiones Mathematicae Graph Theory, 24(2):183–195, 2004.
7. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edition, 2001.
8. Naveen Garg, Marina Papatriantafilou, and Philippas Tsigas. Distributed list coloring: How to dynamically allocate frequencies to mobile base stations, 1996.
9. Patrick Healy and Andrew Ju. An experimental analysis of vertex coloring algorithms on sparse random graphs. In Proceedings ICAA 2014, 2014. (To appear.).
10. Anuj Mehrotra and Michael A. Trick. A branch-and-price approach for graph multicoloring. In Edward K. Baker, Anito Joseph, Anuj Mehrotra, and Michael A. Trick, editors, Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies, volume 37 of Operations Research/Computer Science Interfaces Series, pages 15–29. Springer US, 2007.
11. R. Qu, Edmund K. Burke, B. McCollum, L.T. Merlot, and S.Y. Lee. A survey of search methodologies and automated system development for examination timetabling. Journal of Scheduling, 12(1):55–89, 2009.
12. C. Tsouras and M Satratzemi. A heuristic algorithm for the list coloring of a random graph. In The 7th Balkan Conference on Operational Research, Constanta, Romania, 2005.
13. V. G. Vizing. Vertex colorings with given colors. Metody Diskret. Analiz. (in Russian), 29:3–10, 1976.