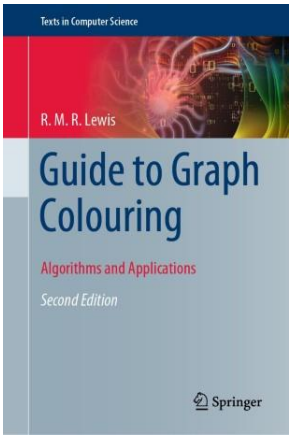
Graph Colouring Algorithm User Guide

R. M. R. Lewis, School of Mathematics, Cardiff University, Wales,
www.rhydlewis.eu, LewisR9@cardiff.ac.uk
Last Updated: Friday, 09 September 2022



1. Summary

This document contains descriptions on how to compile and use the graph colouring algorithm implementations described in the following publication:

Lewis, R. (2021) A Guide to Graph Colouring: Algorithms and Applications (second ed.). Springer, isbn: 978-3-030-81053-5, doi: 10.1007/978-3-030-81054-2

These implementations can be downloaded from http://rhydlewis.eu/resources/gCol.zip. Section 9 of this document also contains a short list of typos found in the above edition of this book. If you spot any more, please get in contact with the author.

Update Information. As of the above date, this resource has been updated to feature some new algorithms and to address some minor issues arising in earlier versions. As a result, the usage instructions described in this document are slightly different to those listed in the first (2015) edition of this book. A summary of these changes is as follows:

- All code has now been modified to comply with C++ 11 standards.
- Improved error checking/reporting and code layout.
- After all runs, each algorithm now also appends basic run information to a log file.
- Implementations of the constructive algorithms **Greedy**, **DSatur**, and **RLF** have now been altered to feature asymptotic runtimes of O(n + m), O((n + m) lg n), and O(nm) respectively (where n is the number of vertices, and m is the number of edges).
- Two new constructive algorithms have now also been added to the resource. The **Welsh-Powell** algorithm uses the **Greedy** algorithm, but first orders the vertices by degree (largest first). It therefore operates in O(n lg n + m) time. The algorithm **Greedy-IS** has now also been added, where IS stands for Independent Sets. This algorithm operates in the same manner as **RLF** except that vertices are chosen randomly instead of using heuristics based on vertex degrees. Like the **Greedy** algorithm, **Greedy-IS** operates in O(n + m) time.
- The five constructive algorithms, **Greedy**, **Welsh-Powell**, **DSatur**, **Greedy-IS** and **RLF**, are now implemented as part of the same C++ program.

2. Compilation Instructions

Once downloaded and unzipped, we see that this directory contains six sub-directories. Different algorithms are contained within each of these. Specifically,

- **AntCol:** The AntCol algorithm, based on ant colony optimisation.
- **BacktrackingDSatur:** The Backtracking algorithm based on the DSatur heuristic'
- **Constructive:** The Greedy, Welsh-Powell, DSatur, Greedy-IS and RFL algorithms.
- **HillClimber:** The hill-climbing algorithm.
- **HybridEA:** The hybrid evolutionary algorithm.
- **PartialColAndTabuCol:** The PartialCol and TabuCol algorithms.

Full descriptions of these algorithms can be found in the above publication. All algorithms are programmed in C++ and can be compiled in both Windows (using Microsoft Visual Studio) and Linux (using **g++**). Instructions on how to do this now follow.

2.1. Compilation in Microsoft Visual Studio

To compile and execute using Microsoft Visual Studio, use the following steps.

1. Open Visual Studio and click **File**, then **New**, and then **Project from Existing Code**.
2. In the dialogue box, select **Visual C++** and click **Next**.
3. Select one of the sub-directories above, give the project a name, and click **Next**.
4. Finally, select **Console Application Project** for the project type, and then click **Finish**.

The source code for the chosen algorithm can then be viewed and executed from the Visual Studio application. Release mode should be used during compilation to make the programs execute at maximum speed.

2.2. Compilation with g++

To compile the source code using **g++**, please use the included **makefiles**.

3. Algorithm Input

Input files are used to specify individual problem instances for the algorithms. These are simple undirected graphs and should be specified using the DIMACS format.

Below are the first few lines of the file **graph.txt**, which is included in each subdirectory as an example. The initial lines begin with the character "c". These are comments that give the user textual information about the graph, which are ignored by the programs. The single line beginning with "p edge" then specifies that edges will be used to specify the graph, and that it contains 250 vertices and 17,083 edges. Note that vertices are labelled from 1 upwards. Finally, lines beginning with "e" give the edges of the graph. As a result, there are 17,803 lines beginning with "e".

```
c Example graph. This is a random graph with 250 vertices and edge probability 0.55 (17083 edges)
c Min Degree = 115; Av. degree = 136.664 (SD = 7.524301); Max degree = 159
p edge 250 17083
e 2 1
e 3 2
e 4 2
e 4 3
e 5 1
e 5 2
e 5 4
e 6 1
e 6 3
...
```

4. Algorithm Usage

Once generated, the executable files in each sub-directory can be run from the command line. If the programs are called with no arguments, usage information is printed to the screen. For example, suppose we are using the executable file **HillClimber**. Running this program with no arguments from the command line gives the following output:

```
Hill Climbing Algorithm for Graph Colouring

USAGE:
<InputFile> (Required. File must be in DIMACS format)
-s <int>    (Stopping criteria expressed as number of constraint checks.
```

```
             Can be anything up to 9x10^18. DEFAULT = 100,000,000.)
-I <int>    (Number of iterations of local search per cycle. DEFAULT = 1000)
-r <int>    (Random seed. DEFAULT = 1)
-T <int>    (Target number of colours. Algorithm halts if this is reached. DEFAULT = 1.)
-v          (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
****
```

The input file should contain the graph colouring problem to be solved. This is the only mandatory argument. The remaining arguments for each of the programs are optional and are allocated default values if left unspecified. Here are some example commands using the **HillClimber** executable:

```
HillClimber graph.txt
```

This will execute the algorithm on the problem given in the file **graph.txt**, using the default of 1000 iterations of local search per cycle and a random seed of 1. The algorithm will halt when 100,000,000 constraint checks have been performed. No output will be written to the screen. Another example command is:

```
HillClimber graph.txt -r 1234 -T 50 -v -s 500000000000
```

This run will be similar to the previous but will use the random seed 1234 and will halt either when 500,000,000,000 constraint checks have been performed, or when a feasible solution using 50 or fewer colours has been found. The presence of **-v** means that output will be written to the screen. Including **-v** more than once will increase the amount of output.

The arguments **-r** and **-v** are used with all of the algorithms supplied here. Similarly, **-T** and **-s** are used with all algorithms except for the constructive algorithms. Descriptions of arguments particular to just one algorithm are found by typing the name of the program with no arguments, as described above. Interpretations of the run-time parameters for the various algorithms can be found by consulting the publication at the top of this document. A full listing of all run time parameters for each algorithm is given below.

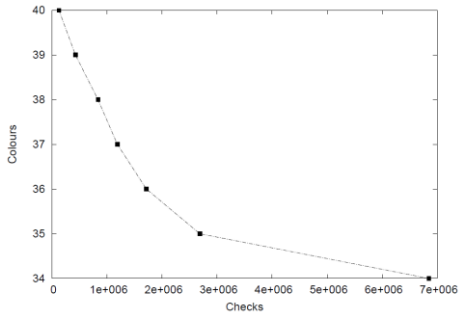
5. Algorithm Output

When a run of any of the programs is completed, three files are created, and one file is added to. The three created file are **ceffort.txt** (computational effort), **teffort.txt** (time effort) and **solution.txt**. The first two specify how long (in terms of constraint checks and milliseconds respectively) solutions with certain numbers of colours took to produce during the run. For example, we might get the following computational effort file:

```
40   126186
39   427143
38   835996
37   1187086
36   1714932
35   2685661
34   6849302
33   X
```

This file is interpreted as follows: The first feasible solution observed used 40 colours, and this took 126,186 constraint checks to achieve. A solution using 39 colours was then found after 427,143 constraint checks, and so on. To find a solution using 34 colours, a total of 6,849,302 constraint checks were required. Once a row with an X is encountered, this indicates that no further improvements were made – that is, no solution using fewer colours than that indicated in the previous row was achieved. Therefore, in this example, the best solution found used 34 colours. For consistency, the X is always present in a file, even if a specified target has been met.

The file **teffort.txt** is interpreted in the same way as **ceffort.txt**, with the right-hand column giving the time (in milliseconds) as opposed to the number of constraint checks. Both files are useful for analysing algorithm speed and performance. For example, the computational effort file above can be used to generate the following plot:



The file **solution.txt** contains the best feasible solution (i.e., the solution with the fewest colours) that was achieved during the run. The first line of this file gives the number of vertices n; the remaining n lines then state the colour of each vertex, using colour labels from zero upwards. For example, the following solution file:

```
5
0
2
1
0
1
```

is interpreted as follows: There are 5 vertices. The 1st and 4th vertices are assigned to colour 0, the 3rd and 5th vertices are assigned to colour 1, and the 2nd vertex is assigned to colour 2. Hence, three colours are used in total.

Finally, on completion of a run, information is also appended to the file **resultsLog.log**. This appears on a single line and contains the following pieces of information (in this order), separated by tabs.

1. Name of the algorithm;
2. Number of colours used in the best feasible solution observed during the run;
3. Total run time (in milliseconds);
4. Total number of constraint checks performed during the run.

If the run resulted in an error (due to unrecognised input parameters or an invalid input file), then appropriate single-line error messages are also appended to the file.

6. Which Algorithm Should I Use?

If you are interested in using one of these algorithms and you don't know which one to choose (or you don't really care how they work), here are some recommendations:

1. **Hybrid EA (HEA):** In general, this algorithm has been found to produce the best results (in terms of the number of colours used in its solutions) in the book cited at the beginning of the document. The default parameters for the algorithm are a reasonable choice in most cases.
2. **BacktrackingDSatur:** If you require a complete algorithm (i.e., you want to find the provably optimal solution to a particular problem), then use this algorithm. However, be warned that for larger problems, this algorithm may not be able to give you the proven optimal in reasonable time. Also, if halted early, it may not give you a solution of comparable quality to some of the other methods.

Further information on the relative performance of these algorithms can be found in the book cited at the start of this document.

7. Execution Commands for the Algorithms

Below is a list of run-time parameters for each available algorithm. As mentioned, this information is generated by the programs when they are executed with no arguments.

```
AntCol Algorithm for Graph Colouring

USAGE:
```

```
<InputFile>     (Required. File must be in DIMACS format)
-s <int>        (Stopping criteria expressed as number of constraint checks. Can be anything up to
         9x10^18. DEFAULT = 100,000,000.)
-I <int>        (Number of iterations of tabu search per cycle. This figure is multiplied by the graph
                 size |V|. DEFAULT = 2)
-r <int>        (Random seed. DEFAULT = 1)
-T <int>        (Target number of colours. Algorithm halts if this is reached. DEFAULT = 1.)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
****
```

```
Backtracking DSatur Algorithm for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-s <int>        (Stopping criteria expressed as number of constraint checks. Can be anything up to
                 9x10^18. DEFAULT = 100,000,000.)
-r <int>        (Random seed. DEFAULT = 1)
-T <int>        (Target number of colours. Algorithm halts if this is reached. DEFAULT = 1.)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
****
```

```
Constructive Algorithms for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-a <int>        (Algorithm choice: 1 = Greedy (random vertex permutation) (default)
                                   2 = Greedy (descending vertex degrees / Welsh-Powell algorithm)
                                   3 = DSatur
                                   4 = Greedy-IS (random vertex permutation)
                                   5 = RLF)
-r <int>        (Random seed. DEFAULT = 1)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
```

```
Hill Climbing Algorithm for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-s <int>        (Stopping criteria expressed as number of constraint checks. Can be anything up to
                 9x10^18. DEFAULT = 100,000,000.)
-I <int>        (Number of iterations of local search per cycle. DEFAULT = 1000)
-r <int>        (Random seed. DEFAULT = 1)
-T <int>        (Target number of colours. Algorithm halts if this is reached. DEFAULT = 1.)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
****
```

```
Hybrid EA for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-s <int>        (Stopping criteria expressed as number of constraint checks. Can be anything up to
                 9x10^18. DEFAULT = 100,000,000.)
-I <int>        (Number of iterations of tabu search per cycle. This figure is multiplied by the graph
                 size |V|. DEFAULT = 16)
-r <int>        (Random seed. DEFAULT = 1)
-T <int>        (Target number of colours. Algorithm halts if this is reached. DEFAULT = 1.)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
-p <int>        (Population Size. Should be 2 or more. DEFAULT = 10)
-a <int>        (Choice of construction algorithm to determine initial value for k. DSatur = 1,
                 Greedy = 2. DEFAULT = 1.)
-x <int>        (Crossover Operator. 1 = GPX (2 parents)
                                     2 = GPX (2 parents + Kempe chain mutation)
                                     3 = MPX (4 parent crossover with q=2)
                                     4 = GGA (2 parents)
                                     5 = nPoint (2 parents)
                 DEFAULT = 1)
-d              (If present population diversity is measured after each crossover)
****
```

```
PartialCol and TabuCol Algorithm for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-t              (If present, TabuCol is used. Else PartialCol is used.)
-tt             (If present, a dynamic tabu tenure is used (i.e. tabuTenure = (int)(0.6*nc) +
                 rand(0,9)). Otherwise a reactive tenure is used).
-s <int>        (Stopping criteria expressed as number of constraint checks. Can be anything up to
                 9x10^18. DEFAULT = 100,000,000.)
```

```
-r <int>        (Random seed. DEFAULT = 1)
-T <int>        (Target number of colours. Algorithm halts if this is reached. DEFAULT = 1.)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is given.)
-a <int>        (Choice of construction algorithm to determine initial value for k. DSatur = 1,
                 Greedy = 2. DEFAULT = 1.)
****
```

8. Copyright Notice

9. Errata

The following is a list of typos that have been identified in the 2021 edition of this book (second edition). If you spot any others, please send them to the author at LewisR9@cardiff.ac.uk and he will be happy to note them here.

- **Page 33, final paragraph of Section 2.4.** Should read "These results allow us to conclude that Problems 2 to **5** from Definition…"
- **Page 60:** Solution produced by the RLF algorithm should read {{v_1,v_6}, {v_2,v_4,v_8}, {v_3,v_5}, {$\mathbf{v_7}$}}.
- **Page 65, Fig. 3.17.** On Line (4) of the pseudocode, replace the *i* with a *j*.
- **Page 98, Fig. 4.10.** In Step (2), replace "come" with "some".
- **Pages 193, 195 and 196.** Replace occurrences of $G(V, E, w)$ with $G = (V, E, w)$.
- **Page 197, first line:** should read "expressed as **a** polynomial".
- **Page 200, Reference 7.** Year should be "**1977**", not "1997".
- **Page 206.** Please note that the wedding seat planner tool used in this section requires Adobe Flash player to operate. However, as of December 202 Flash Player is not supported by most web browsers. Indeed, Adobe strongly recommends all users immediately uninstall Flash Player to help protect their systems. As an alternative, a command-based C++ implementation of the wedding seat planner is available for download at http://www.rhydlewis.eu/resources/wsp.zip.

10. Acknowledgement

The author would like to give his thanks to Paulo Neis (email: neis@neis.com.br) from Brazil who produced the attached makefiles and who also helped to spot a small number of bugs in an earlier version of this code.