# Algorithm User Guide

This document contains descriptions on how to compile and use the various graph colouring algorithm implementations used in the following publications:

- Lewis, R. (2015) *A Guide to Graph Colouring: Algorithms and Applications*. Berlin, Springer. ISBN: 978-3-319-25728-0. http://www.springer.com/us/book/9783319257280
- Lewis, R., J. Thompson, C. Mumford, and J. Gillard (2012) 'A Wide-Ranging Computational Comparison of High-Performance Graph Colouring Algorithms'. *Computers and Operations Research*, vol. 39(9), pp. 1933-1950.

These executables are downloadable from http://rhydlewis.eu/resources/gCol.zip

Once downloaded and unzipped, you will see that the directory contains a number of sub-directories. Each algorithm is contained within its own sub-directory. Specifically, these are:

| | |
|---|---|
| AntCol | The Ant Colony Optimisation-based algorithm for graph colouring |
| BacktrackingDSatur | The Backtracking algorithm based on the DSatur heuristic |
| DSatur | The DSatur algorithm |
| HillClimber | The hill-climbing algorithm |
| HybridEA | The hybrid evolutionary algorithm |
| PartialColAndTabuCol | The PartialCol and TabuCol algorithms |
| RLF | The recursive largest first (RLF) algorithm |
| SimpleGreedy | The Greedy algorithm, using a random permutation of the vertices |

All of these algorithms are programmed in C++. They have been successfully compiled in Windows using Microsoft Visual Studio 2010 and in Linux using the GNU Compiler g++. Instructions on how to do this now follow.

## Compilation in Microsoft Visual Studio

To compile and execute using Microsoft Visual Studio the following steps can be taken:

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.

## Compilation with g++

To compile the source code using g++ at the command line, navigate to each sub-directory in turn and use the following command:

```
g++ *.cpp -O3 -o myProgram
```

By default this will create a new executable program called **myProgram** that can then be run from the command line (you should choose your own name here). The optimisation option **-O3** ensures that the algorithms execute at maximum speed. A **makefile** is also supplied for compiling all programs in one go if preferred.

## Input

Input files must be in the DIMACS format, as described here: mat.gsia.cmu.edu/COLOR/general/ccformat.ps. An example input file called **graph.txt** has been included in all subdirectories.

The graph described in **graph.txt** is a random graph comprising 250 vertices, in which each pair of vertices has been joined by an edge with probability 0.55. In this case the graph has 17,083 edges, a minimum degree of 115, an average degree of 136.67 (standard deviation 7.52), and a maximum degree of 159.

Below are the first 20 lines of **graph.txt**. Lines beginning with "c" are comments that give the user textual information about the graph. They are ignored by the programs. The single line beginning with "p" 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. There are 17,803 lines beginning with "e" in total.

```
c DESCRIPTION: Quasi-random coloring problem
c CODE SOURCE: Joseph Culberson (joe@cs.ualberta.ca)
c Specifications:
c   Random seed: 5
c   No hidden coloring
c   Probability: 0.550000
c   random IID graph
c   Degree Information:
c   Min:115 Avg:136.664000 Max:159 Std:7.524301
c No verification required.
c Creation Date: Thu Sep 24 15:25:54 2009
p edge 250 17083
c no cheat
e 2 1
e 3 2
e 4 2
e 4 3
e 5 1
e 5 2
e 5 4
```

**Note to Linux/Unix users:** The file **graph.txt** was created on a DOS/Windows machine and therefore uses a carriage return and line feed for line endings. If you are using a Linux/Unix machine you may find that you first need to convert this file to the correct format for it to be read correctly by the implementations. The command **dos2unix** is suitable for this (https://en.wikipedia.org/wiki/Unix2dos).

## 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, useful 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 single-parse constructive algorithms DSatur, RLF and Greedy. 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 above publications. A full listing of all run time parameters for each algorithm is given below.

## Which Algorithm Should I Use?

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

- **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 literature cited at the beginning of the document. The default parameters for the algorithm are also good choices in most cases.
- **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 at the publications cited at the beginning of this document.

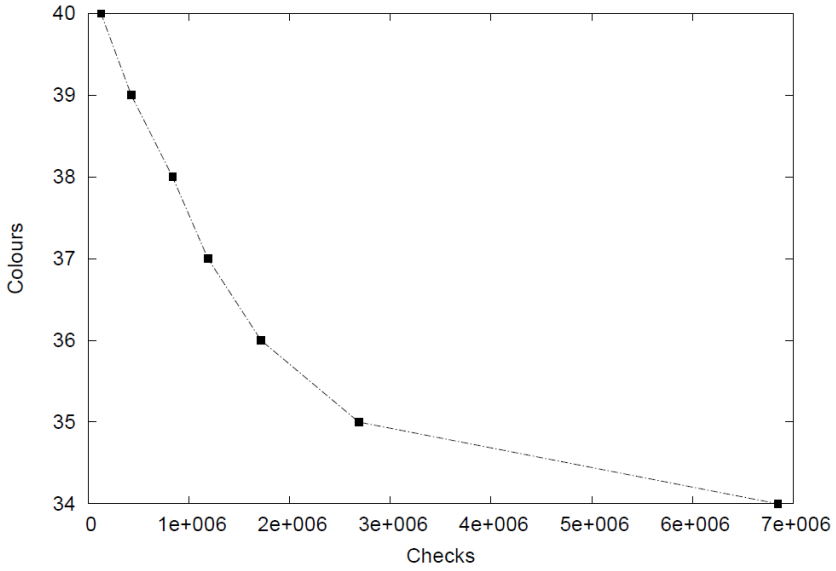
## Output

When a run of any of the programs is completed, three files are created: **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 last 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 with 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 of these files are useful for analysing algorithm speed and performance. For example, the computational effort file above can be used to generate the following plot:




Finally, the file **solution.txt** contains the best feasible solution (i.e. the solution with fewest colours) that was achieved during the run. The first line of this file gives the number of vertices **n**, and the remaining **n** lines then state the colour of each vertex, using labels **0, 1, 2,…**

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.

## Copyright notice

otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage. This software is supplied without any support services.

Please direct any queries/comments to Rhyd Lewis: web: www.rhydLewis.eu, email: LewisR9@cf.ac.uk

*Diolch!*

R. Lewis (Last updated Friday, 28 July 2017)

## Acknowledgement

R. Lewis 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.

## Execution Commands for the Algorithms

Below is a list of run-time parameters for each available algorithm. 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.)
****
```

```
DSatur Algorithm for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-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. DSsatur = 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. DSsatur = 1,
                 Greedy = 2. DEFAULT = 1.)
****
```

```
Recursive Largest First Algorithm for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-r <int>        (Random seed. DEFAULT = 1)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is
                 given.)
****
```

```
Greedy Algorithm for Graph Colouring

USAGE:
<InputFile>     (Required. File must be in DIMACS format)
-r <int>        (Random seed. DEFAULT = 1)
-v              (Verbosity. If present, output is sent to screen. If -v is repeated, more output is
                 given.)
****
```