Problem Generator for the Maximum Happy Vertices Problem: User Guide

This document contains descriptions on how to compile and use the program for generating instances of the maximum happy vertices colouring problem, described in the following paper:

- Lewis, R., Thiruvady, D. and Morgan, K. "Finding Happiness: An Analysis of the Maximum Happy Vertices Problem" *Computers and Operations Research*, doi: 10.1016/j.cor.2018.11.015.

This program has been written in C++ (V11) and is available at http://rhydlewis.eu/resources/happyGen.zip. It has been successfully compiled in Windows using Microsoft Visual Studio 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 the subdirectory containing **gen.cpp** and click **Next**.
4. Finally, select **Console Application Project** for the project type, and then click **Finish**.

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

Compilation with g++

To compile the source code in Linux, please use the included **makefile**.

Usage

Once generated, the executable file should be run from the command line. If the program is called with no arguments, the following usage information will be printed to the screen.

```
Maximum Happy Vertices Problem Generator:

 USAGE:
 -------------
 PARAMETERS:
       -n <int>    (number of vertices. Default = 100)
       -p <double> (proportion of vertices to be preassigned to a colour. Default = 0.1)
       -k <int>    (number of different colours to be used in the preassignments. Default = 5)
 -------------
 GRAPH TYPE: (default = -R 0.5)
       -R <double> (produce a random graph with the stated edge probability (between 0 and 1).
       -S <int>    (produce a scale-free graph, adding the stated number of links for each new vertex (between 0 and n - 1).
       -D <int>    (produce a d-regular graph where each vertex has the same, specified degree (between 0 and n - 1).
 -------------
 OTHER OPTIONS:
       -s <int>    (random seed. Default = 1)
       -f <string> (name of output file. Default = graph.txt)
       -d <int>    (precolour certain vertices. 1 = random choice, 2 = highest degrees only, 3 = lowest degrees only. Default = 1)
       -A          (if present we do not allow adjacent vertices to be preassigned to different colours.)
 -------------
```

This provides the information needed to produce valid commands. Here are some examples:

```
gen –n 200
```

This will produce a problem instance, written to the file **graph.txt**, containing 200 vertices. Defaults values will be used for all other parameters, as specified above.

```
gen -n 1000 -p 0.2 -k 10 -R 0.3 -s 123 -f mygraph.txt
```

This produces a problem instance with 1000 vertices, of which 200 are precoloured (proportion of 0.2), using 10 colours. The graph is a random graph, where each pair of vertices is joined with probability 0.3. The random seed used to produce this instance is 123, and the instance has been written to the file **mygraph.txt**.

```
gen -n 1000 -p 0.25 -k 10 -S 5
```

The generated problem here is a scale-free graph, where 5 edges have been added for each new vertex, according to the Barabási–Albert method (see below). The problem contains 10 colours and 250 of the vertices have been precoloured.

How the problem instances are generated.

In this section we now describe in more detail how the problem instances are generated.

First, the appropriate graph needs to be formed.

- For random graphs, we go through each pair of vertices in order and add an edge with a fixed probability q (input by the user as stated above).
- For scale free graphs, we use the Barabási–Albert model, adding m edges at each iteration, where m is the parameter input by the user. To do this we start with a complete graph K_m. At each iteration we then add one new vertex to the graph, and give it exactly m new edges. The endpoints for these edges are chosen via roulette wheel selection using probabilities proportional to the degree of existing vertices (as advised in the Barabási–Albert model). We stop once we have a graph with n vertices.
- Random d-regular graphs are generated using the Steger and Wormald method as described in their paper "Generating random regular graphs quickly" (1999) Combinatorics Probability and Computing 8(4):377-396. http://www.math.uwaterloo.ca/~nwormald/papers/randgen.pdf. The degree of every vertex in the final graph will be d.

Having produced the appropriate graph, we can now precolour a proportion p of the vertices. The number of vertices to precolour, x, is simply the number of vertices multiplied by p, rounded down to the nearest integer (where p is specified using **-p** at the command line). If precolourings are to be assigned randomly (**-d 1** is specified at the command line) the vertices are now arranged in a random order. The first k vertices in this permutation are then assigned to colours 1 to k respectively. This ensures that each colour is used at least once in the graph. The next (x – k) vertices in this permutation are then each assigned to random colours from the set {1, 2, ..., k}. If options **-d 2** or **-d 3** are selected, the same process is followed, except that the permutation used contains the vertices in the desired order of degree.

Note that there is also an additional option in the program (**-A**) that ensures that adjacent vertices are never precoloured to different colours. For some graphs this might be difficult to do, or even impossible. The program therefore has 100 separate attempts at doing this and, if it fails in all cases, will not return a problem instance. In this case the output file will simply contains the text "**NULL**". This failure is also recorded in the log file **genLog.txt**, specified below.

Output Format:

Problem instances made by this generator are output in the DIMACS format. For ease of reading, the vertices are labelled in descending order of degree. Specifically,

- Lines beginning with a **c** contain comments and can be ignored;
- The single line beginning with **p edge** contains, respectfully, the number of vertices, the number of edges, and the number of vertices preassigned to a colour;
- Lines beginning with **e** state the graph adjacencies (i.e., the endpoints of each edge);

- Lines beginning with **n** state the precolourings by giving, respectively, the vertex number and its colour.

For reference, an example graph, **graph.txt**, is included with this resource.

Log file.

For book-keeping purposes, each time this program is executed, a single line of information is also added to the file **genLog.txt**. This contains the following information, separated by tabs:

- Name of the file created.
- Number of vertices.
- Number of colours.
- Number of precoloured vertices.
- Type of graph created, followed by its generating parameter (i.e., Random graph and its edge probability; Scale-free graph and the number of links per new node; d-regular graph with the stated degree).
- The resultant graph density.
- Random seed.
- Which precolouring option was used? (1/2/3).
- Are adjacent vertices allowed to be precoloured to different colours in the graph? (T/F).
- Comment: Was the instance successfully created?

Copyright notice

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

R. Lewis (Last updated Friday, 23 November 2018)