Short manual for using minibaum.c

Minibaum.c is written for generating all connected simple cubic graphs with a fixed number of vertices, but can be restricted to generating only graphs that have a fixed minimal girth or are bipartite. It is designed to run on Unix machines.
It must be called **minibaum v g** to generate all cubic graphs on up to v vertices and girth at least g.
So **minibaum 20 3** generates all cubic graphs with 4, 6, ..., 20 vertices.
The standard way to store the graphs is to write them into files named `Codes_12.03.05` e.g. which means: 12 vertices, 3-regular, girth at least 5. For exceptions see the options.
The coding of the graphs is as follows: The vertices of a graph with n vertices are numbered 1 to n. Vertex 1 is always adjacent to vertex 2, 3 and 4, so there is no need to store this. For every vertex, all the adjacent vertices with a number larger than the vertex itself, are stored with one byte per entry all in a row starting with vertex 2, then 3, ...
So if one wants to reconstruct the graph from the code, one first has to fill in the edges (1,2),(1,3) and (1,4). Then one has to fill up the list of vertex 2: Vertex 2 is already adjacent to 1, so two edges are missing. This means that the next two entries in the code are the missing neighbours of 2.
In general: If one has already filled up the adjacency list of vertex 1 to $k-1$ and the valence of vertex k is m, then the next $3-m$ entries give the missing neighbours of vertex k. It is easily seen, that coded this way every graph needs $3 \cdot (n/2) - 3$ bytes, with n the number of vertices.

- Examples:
  - The Code of the $K_{3,3}$: 5 6 5 6 5 6
  - The Code of the K_4: 3 4 4

Options:

All the options must be given after the number of vertices and the minimal girth.
So e.g. **minibaum 30 6 b r c1** or **minibaum 30 6 r c1 b** are correct – **minibaum 30 b 6 r c1** is not and will definitely lead to a program mistake.

Possible options:

b: This option must be used if only bipartite graphs shall be generated. The output files get an additional ".b" to mark that they only contain bipartite graphs.

c: This option must be used if a given graph shall be part of every graph generated. The form of this graph is by no means arbitrary:

It must be a maximal (!) graph with the properties of the graphs that shall be generated (minimal girth **g** and eventually bipartite) so that there is a number k satisfying:

1. all vertices with a number less than k have valence 3
2. all edges contain at least one vertex with a number less than or equal to k.

Given a graph to start with, all specified maximal graphs that contain this graph as a part and contain only further edges that are lexicographically larger than the ones already inserted are generated.

Please note:

1. Not all the graphs that have a subgraph isomorphic to the startgraph are generated, but only maximal graphs that really contain an identical copy of the startgraph.
2. If k is as above and vertex k has valence 2 and is adjacent to one vertex with a number larger than k, then there might be maximal graphs that really contain a copy of the startgraph, but also contain edges smaller than one of the inserted. They are not generated.
   e.g. (1,2)(1,3)(1,4)(2,3)(2,5)(3,6)(4,7) is part of a lot of cubic maximal graphs containing the edge (4,5) and fulfills all the requirements for a startgraph.

The default startgraph is (1,2)(1,3)(1,4) and these edges should be part of every startgraph.

The *c* option (c for case) must always be used together with a string denoting the case – e.g. **minibaum 20 4 c3a**.

In this case the program reads the startgraph out of a file named `startgraph_20.03.04.c3a`. In the file the edges that shall be inserted have to be put in ASCII-Code. The entries must be separated by white space characters. The edges are assumed to be in lexicographic order.

- e.g.: If the contents of `startgraph_20.03.04.c3a` is:
  1 2
  1 3
  1 4
  Then the same graphs are generated using **minibaum 20 4 c3a** as using **minibaum 20 4** – the only difference is that all the output files get a **.c3a** as the ending.

The contents 1 2 1 3 1 4 is also correct, the contents 1 3 1 2 1 4 is not.

If the option **b** is used, the option must occur in the startgraph after the numbers and before the case number, so e.g. `startgraph_20.03.04.b.c3`

After the generation, the startgraphfile is removed unless you also used option **m**, which should be done only exceptionally and only makes sense if the value of the variable *splitlevel* in the program is larger than the number of edges in the startfile.

Compare options **p** and **m**.

g: use graph6 output format and write to stdout.

i: i stands for independence. Using this option you can give the maximal size of a maximal independent set. In the case of girth at least 4, the presence of a data-file named *MINIMALWERTE* is expected. If you don't have that file, contact the author. This option works efficiently (in particular better than a simple filter) only when the given number is close to the possible minimum, e.g. *minibaum 14 4 i5.*

m: This stands for modulo. It is another way to split the generation into parts. The whole generation can be split e.g. using first *m 0 2* and then *m 1 2.* As long as it is not split into too many parts, the single parts should take about equal time and produce about equally many output. The normal rules for modulo calculation are obeyed, so *m 1 2* together with *m 0 4* and *m 2 4* would also give a complete generation. Some amount of work is done

once for every step, so the generation times will be slightly larger than with e.g. the option **c**.

Since a large number of files is expected, the option **ni** is set.

ni: This stands for *no init*. It says that the Code-files shall not be initialized as empty files. If Code-files for the output exist, a warning is printed and the user is asked whether it is OK to go on.

nocycX: Only graphs without cycles of length X are generated. If this option is chosen, the options o and s are set automatically. Option g can be used to use graph6 output instead. The option may be used for various values of X simultaneaously, but is only (fairly) efficient for small values of X (it is especially **NOT** efficient to avoid hamiltonian cycles).

o: Write the output for the graphs with the largest number of vertices not to a file but to *stdout*. The output for the other graphs is still directed to the files, unless option **s** is used. No lastgraph files to recover are saved, since in case of a system crash graphs may have been lost in the pipe.

O: Observe whether the process is still running. This option uses the unix command "*at*", so it only works, if this command is available for the user. The program generates some files, all starting with "observer..." that are used for testing in 4 hour intervals whether the started process is still running, ready or has been killed (maybe by a computer breakdown). In the last case the process is restarted automatically with the same options (plus option **r** — see option *r* for details).

p: p stands for *partial* and is used if the generation shall be split into several parts. Here a startgraph and an endgraph are needed, both satisfying the same conditions as under **c** except that the endgraph may be empty. The names of the corresponding files have to be `startgraph_20.03.04.p7` and `endgraph_20.03.04.p7` e.g. if a part – called part number 7 by the user – of all cubic graphs with up to 20 vertices and girth at least 4 shall be generated. For the names the same rules apply as under **c**.

Using this option, all the graphs with the pregiven properties (minimal girth,...) are generated that have a code smaller than the startgraph and at most as large as the endgraph. Different from the effect of **c** the generated graphs need not contain the startgraph as a part and the startgraph – even if cubic – is not generated itself.

This option can be very useful, if one e.g. has already generated all graphs with girth **g** and **v** vertices (say there are 1000) and wants to generate all graphs with girth **g** and **v+2** vertices. Then one could take the defaultgraph as the first startgraph, graph no.100 as the first endgraph and the second startgraph, graph no.200 as ..... graph no.900 as the 9th endgraph and the 10th startgraph and the empty graph as the 10th endgraph.

Indeed it would be better not to distribute the intermediate graphs as equally as described, but take longer intervals in the beginning and shorter ones at the end, but that will have to be decided each case on its merits.

P: The number following **P** is interpreted as the priority. The option **P19** e.g. makes the process renice itself to 19. This ensures that in case of an automatic restart (option **O**) the correct priority is set.

r: **r** stands for recover and must be used if the program shall recover from a computer breakdown. While running, from time to time, the program makes copies of the last graph it has worked on in a file named `lastgraph_20.03.04.b.c3` e.g. . If the computer breaks down, the generation can restart right at the point where the last saving was done. The program has to be called just as before, only with the additional option "**r**" for recover. So in the case above one would have to type **minibaum 20 4 b c3 r**. Time consuming checks on the startgraph are left out using this option, because the startgraph should have been checked in the first run. Doing changes in the startgraph file and then using this option would lead to mistakes anyway.

If the data files are newer than the lastgraph file, then the breakdown was right in the middle of the saving and that was really bad luck. Restarting now would produce duplicates in the files which would have to be removed afterwards or there might even be only parts of codes, which would have to be removed before the restart. If the lastgraph file itself is not in order: You could find out the graph with the smallest code from the list, put it into the lastgraph format and restart, but better you save an earlier lastgraph together with the corresponding datafiles somewhere !

R: Some statistics, e.g. the running time and the number of graphs with the maximal number of vertices that were generated are stored in a file called **REMARKS...** .

s: **s** stands for single out. In this case the graphs that have not the maximal number of vertices are not written anywhere, but just suppressed. In some cases this option gives the possibility to optimize the construction and make it (a bit) more efficient. All the times given in the paper are without this option.

S: Only generate *Snarks*. This is only implemented as a filter and since snarks are hard to find, it is not efficient.

sh: *sh* stands for short coding. Using this option the files are named *Sodes...* instead of *Codes....* Now the coding is so that every Code starts with the number (size: unsigned char) of entries that should be taken from the previous code. This number is followed by the entries in which the codes differ. So the first code is always a "0" followed by the "almost" usual code. In fact the entries for the first three edges are given as "2", "3", "4", since they have to be given only once. So the code start "0 2 3 4". In case of recovering, the program also starts with a "0", so the length of the files produced during runs with or without recovering can differ sligthly.

Example: Normal codes 3 5 6 5 7 8 7 8 8 and 3 5 6 7 8 7 8 7 8

give short codes 0 3 5 6 5 7 8 7 8 8 and 3 7 8 7 8 7 8

The advantage of this coding grows with the number of vertices and the size of the files. In the case of 18 vertices girth 3 it takes less than 30 % of the room the normal coding takes.

t: If the process shall run only for some fixed time intervall, e.g. 18:00 to 6:00, then this option should be used. Here the times must be given as 4 digit strings separated by blanks, so e.g. t 1815 0630 makes the program run only between 18:15 in the evening and 6:30 in the morning. This option also uses the "**at**" command.

Using this option, the program is only suspended from execution – not killed.

Because of problems when trying to get logged out after having started the program outside the desired time region, one should write the command , e.g. **minibaum 24 3 t 2000 0600 &**, into a file and then execute the file or logout directly after starting the program in the background. Otherwise one could get problems with an automatic restart or a termination initiated by the system.

T: This option can be used like **t** – only that in this case the program is really killed (kill -9). This options always allows the program to run on weekends (saturday and sunday) – so e.g. **minibaum 40 8 T 0200 0700** would be killed friday at 7 o'clock, started saturday at 2 o'clock and killed again sunday at midnight. The program **minibaum 40 8 T 1900 0700** would be started friday at 19:00 and killed again monday at 7:00. Maybe using this option one should set **Sicherungsintervall** to 1800.

This option may not be used together with **O** or **t** .

Zx: Only generate graphs with cyclic connectivity at least x. This is implemented only as a filter, so it is not too efficient.

The setting of the Constants

In the program there are constants that should be set by the user. These are:

- **knoten**: The maximal number of vertices the program can deal with. The maximal possible value is 96. In most cases the program needs a lot of machine memory, so that for larger jobs (e.g. generating all cubic graphs with girth at least 8 and 38 vertices) one should set knoten to the smallest possible number (in this case 38) to reduce the memory consumption.

- **baumgrenze**: This constant somehow bounds the memory consumption of the program. For DEC machines with a memory of 32 Megabytes it turned out that 13 is a value that leads to the use of about 7% of the main memory when knoten is set to 40, which should not disturb other users too much. This value can also be used on smaller (but not too small) machines. A value of 14 already needs about 38% and it depends on how many other large jobs are running whether this value can be used.

  The memory consumption of the program also depends on the girth of the graphs that the program works on at the moment. So if one wants to generate all cubic graphs on 24 vertices one can set baumgrenze to 17 without problems, because only for a short time the program works on graphs with a large girth and really needs a lot of memory. If one wants to generate all cubic graphs with 40 vertices and girth at least 8, the program works only on graphs with a large girth and it is impossible to set baumgrenze that high.

  In most cases a larger value of baumgrenze increases the memory consumption and decreases the time consumption.

  The following numbers give the values **T** after that increasing **baumgrenze** has no effect if one calls **minibaum v g** :

g=3:	T=v-6	if v<16
	T= v-7	else
g=4:	T=v-7	
g=5:	T=v-8	
g=6:	T=v-9	
g=7:	T=v-12	if v<28
	T=v-13	else
g=8:	T=v-17	
g=9:	T=v-22	
g=10:	T=v-29	

So for **minibaum 20 5** there is no difference between setting baumgrenze to 12 or to 16.

- **listenlaenge:** This constant tells the program how many graphs of every size must be gathered before writing them to a file. As soon as there are **listenlaenge** graphs of one size, all the graphs gathered are written into files and the last graph is saved in the lastgraph file.

  So if one wants to generate a lot of graphs – e.g. all cubic graphs on 22 vertices, one should choose a large value (e.g. 10 000) – if one wants to generate sparse graphs – e.g. all graphs with girth at least 8 on 38 vertices – one should set **listenlaenge** to 1.

- **sicherungsintervall:** This constant gives the real time interval in seconds after that the next lastgraph saving should be done anyway, even if no further graphs were generated in the meantime. The time elapsed between the savings may nevertheless be longer, because the saving is not possible at any stage of the program.