INTRO TO ISOMORPHISMS AND AUTOMORPHISMS IN FLAGCALC

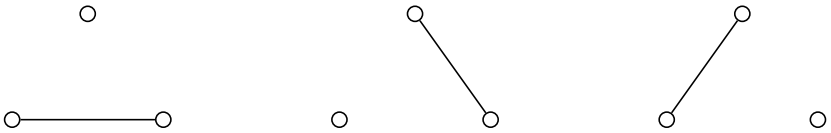
PETER GLENN

FlagCalc has a "help" feature: invoke the command `./flagcalc -h`. This is the most up-to-date record of features[1]. Here, we explore the `-f` feature and the `-i` feature[2].

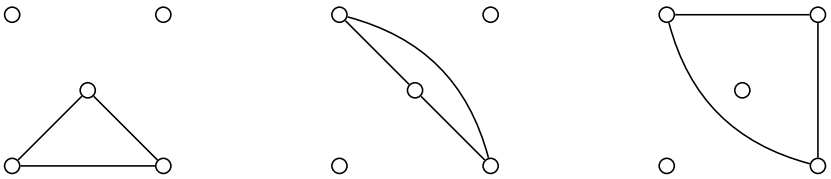
1. Isomorphic Graphs

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are "isomorphic" if there exists a bijection between the vertices $\sigma : V_1 \to V_2$ such that $(a, b) \in E_1$ if and only if $(\sigma(a), \sigma(b)) \in E_2$. In particular, isomorphic graphs have the same number of vertices.

These three graphs are isomorphic:



as are these three:



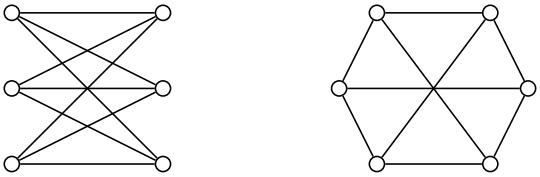
It is an exercise in itself to work out examples of isomorphic graphs: the game is in finding a visually appealing way to display, say, the complete bipartite graph $K_{2,2}$ on four vertices, and then finding another way of looking at that graph: these two graphs are isomorphic

Date: December 1, 2025.

[1]Missing from `-h` are the many "first order quantifiers" available, such as `FORALL`, `EXISTS`, `SUM`, `COUNT`, `MIN`, etc. These will be documented soon, and a crash course after reading the present paper would consist simply of looking at the shell scripts in the folder `scripts`.

[2]These two features are "early" in the evolution of the flagcalc ecosystem, and while quite powerful, there are some changes on the future roadmap: for example, while `-a ... sorted` computes one measure (query) per isomorphism class, it would be nice in the case of a measure like `Chigreedyt` and `Chigreedyp` to check how these vary within an isomorphism class (that is, most queries/measures are equivalent within any given set of isomorphic graphs, but the greedy chromatic number algorithm can produce varied output depending on how the vertices are ordered). Quick to note: notwithstanding this roadmap, there is already the ability to compute within the isomorphism class by using `-a ... all` and manually comparing within an isomorphism class, as demonstrated later in this paper.

Or the complete bipartite graph $K_{3,3}$ on six vertices:



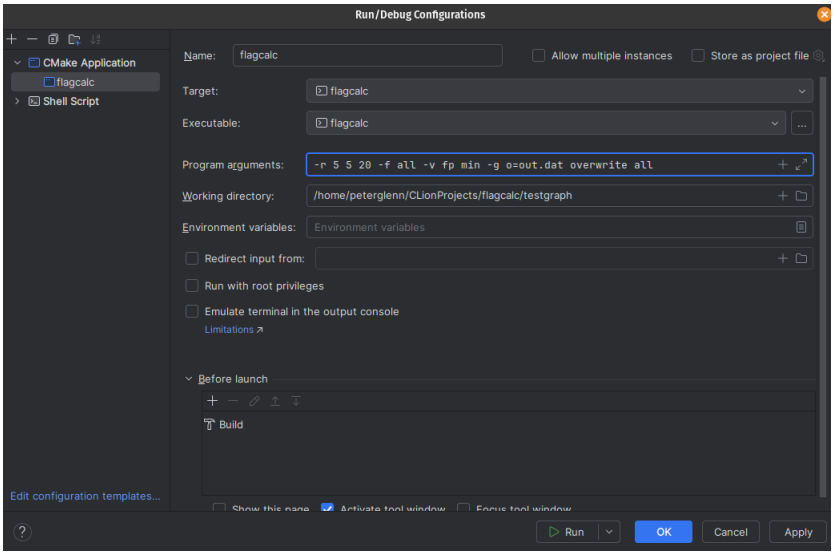
2. Isomorphisms with FlagCalc

One basic exercise is to pick a small vertex count, say, five, and request, say, twenty random graphs with an average edge count of $0.5\binom{5}{2} = 5$; then invoke flagcalc with the command-line options `-f all`; we add the optional "verbosity" level of `fp min`:

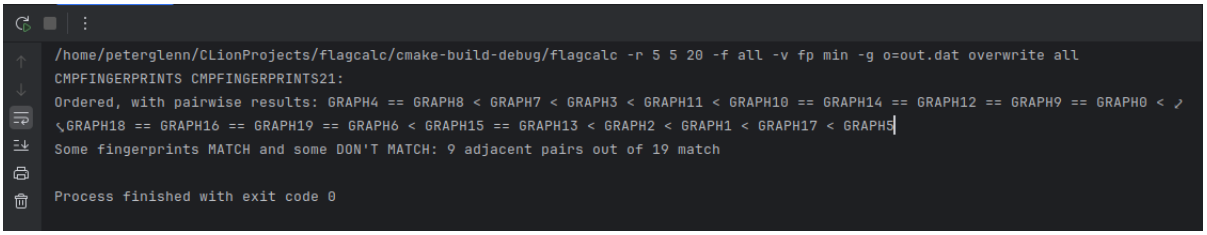
```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 5 5 20 -f all -v fp min
CMPFINGERPRINTS CMPFINGERPRINTS21:
Ordered, with pairwise results: GRAPH14 < GRAPH10 < GRAPH4 < GRAPH7 == GRAPH16 < GRAPH11 < GRAPH3 == GRAPH8 <
 GRAPH6 == GRAPH12 < GRAPH2 < GRAPH19 < GRAPH13 < GRAPH0 < GRAPH17 == GRAPH1 == GRAPH9 < GRAPH18 == GRAPH15 < GRAPH5
Some fingerprints MATCH and some DON'T MATCH: 6 adjacent pairs out of 19 match

Process finished with exit code 0
```

Please note that -f asks flagcalc to linearly order graphs according to their "fingerprint" (not a term in the literature; use `-v fp` or `-v fp FpMin` (in either case, omit `min`) to see examples of a graph's "fingerprint"), and so the isomorphic graphs work out to appear adjacent ("==") to each other in the ordering shown in the output above.

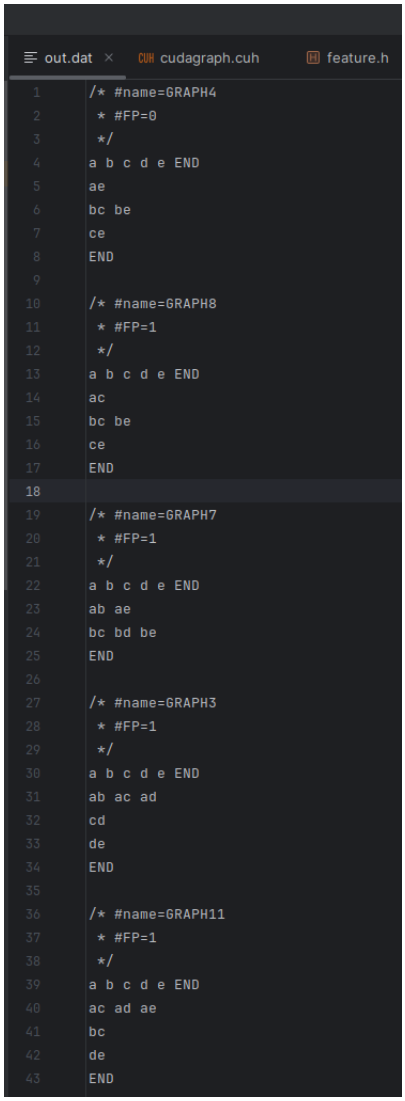
Now, this doesn't print out the graphs unless we change the verbosity level to `-v fp fpnone graphs`. And this is a fine way to get the material for study. An alternate invocation uses the `-g o=out.dat overwrite all` option to save the graphs to a file.



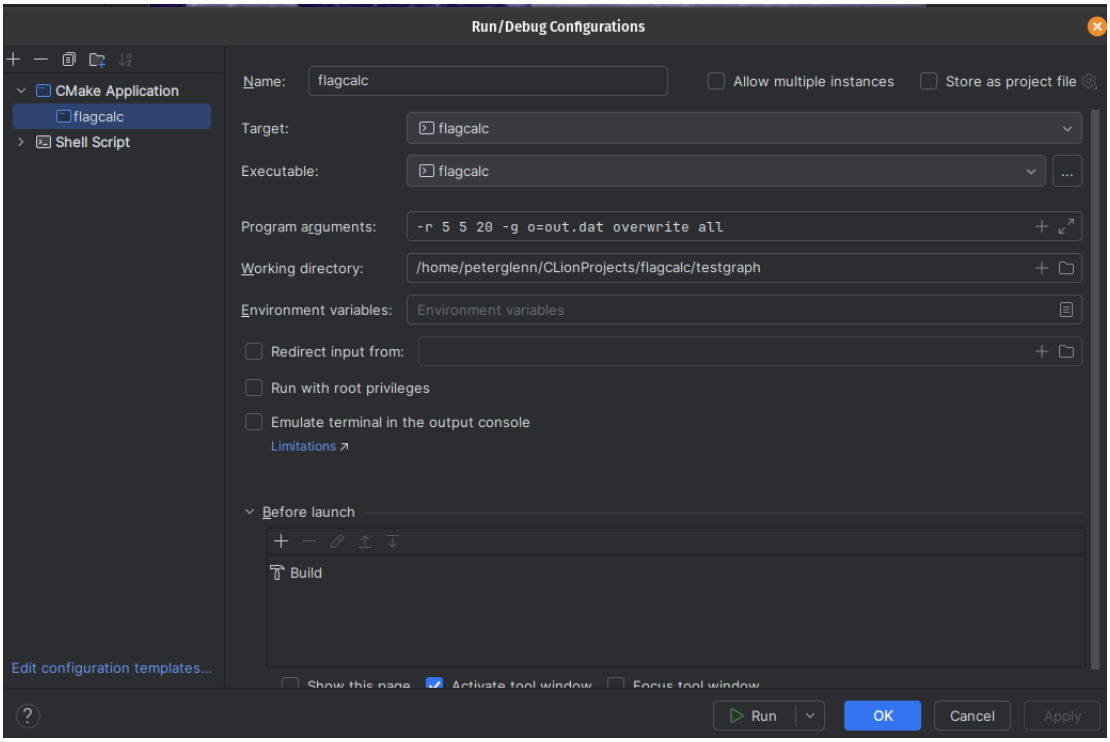
Producing



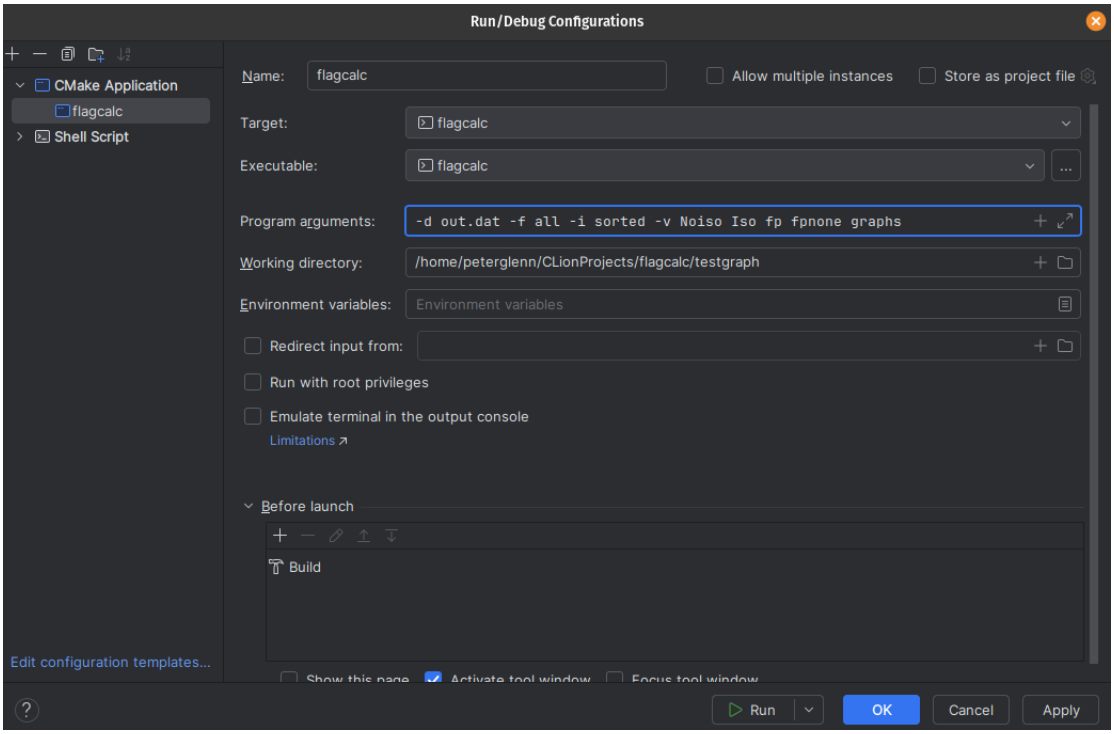
which matches to (only the first few graphs in the out.dat file are displayed here):



Please note that once you have used `-g` to save to a file such as `out.dat`, you can replace the `-r` random source of graphs, with `-d out.dat` to read from the file named "out.dat". This allows continued analyses of a fixed set of random graphs, and indeed, one can simply start the project with the following invocation:



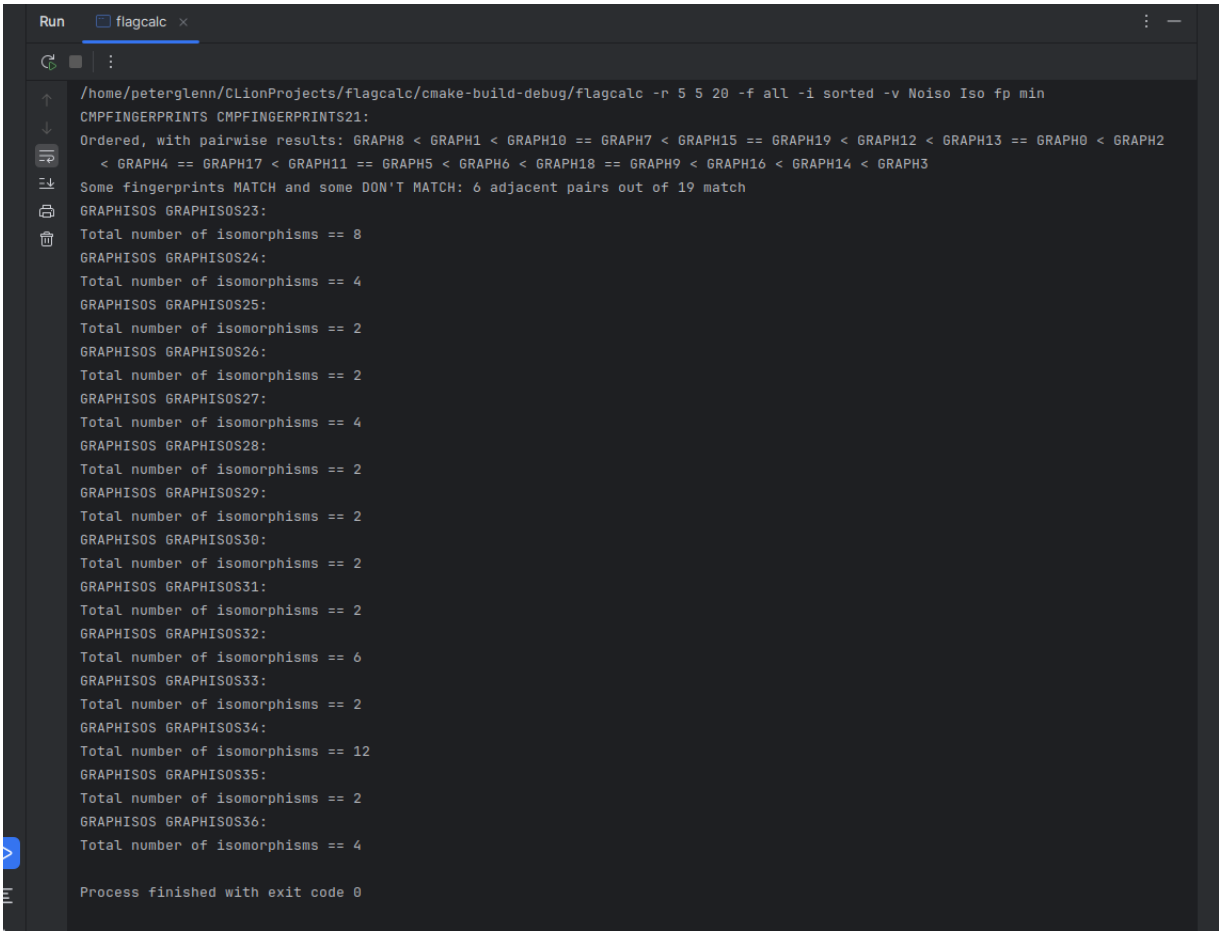
That is, if invoked as above, then for the remainder of the work just use the file "out.dat" as the source of graphs, rather than the randomizer `-r` feature:



As an exercise now, choose an arrangement of vertices, draw each of the twenty graphs according to the `-g` output or the verbose output, and explore how they match up with those marked "==" (that is, isomorphic).

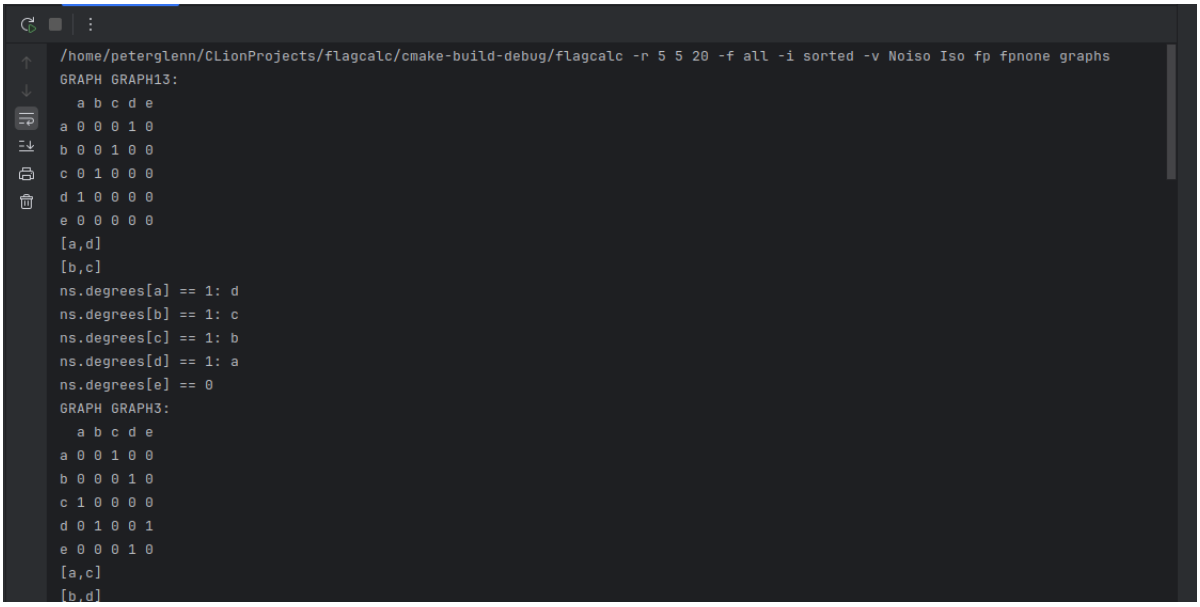
3. Automorphisms with FlagCalc

Next, we consider isomorphisms of a graph with itself, which are called *automorphisms*. Run flagcalc as above, but with the additional option `-i sorted` (placed after `-f all`) and with verbosity options `-v Noiso Iso fp min`. This produces

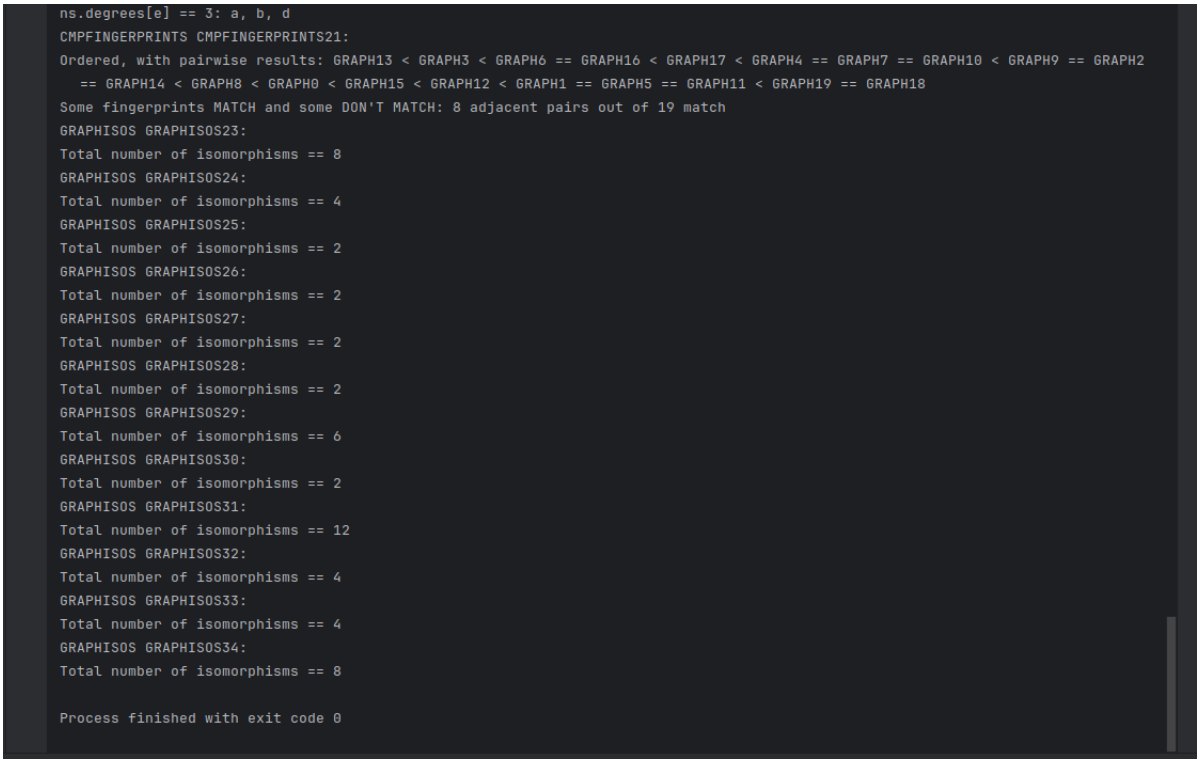


Please note that for each equivalence class of isomorphic graphs this outputs the number of automorphisms of the graph (convince yourself that if two graphs are isomorphic, then their automorphism counts respectively are equal).

Use either the "out.dat" file or add the additional verbosity level `graphs` to output the twenty test-graphs, so one can experiment with paper and pen on these graphs, counting automorphisms and checking that they match the output.

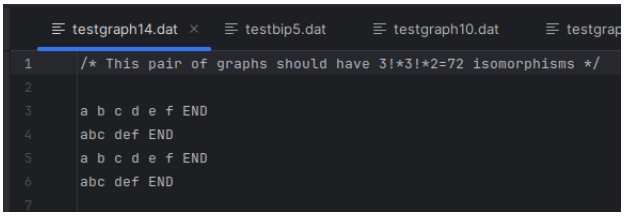


...which continues...

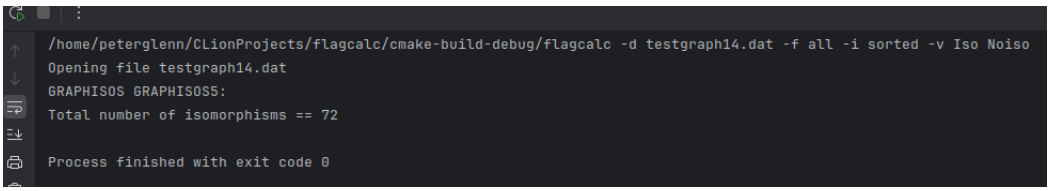


For example, each of the three three-vertex graphs at the opening of this paper have two automorphisms apiece. Remove the `Noiso` from `-v`, and you will see these two automorphisms spelt out:

Each of the three five-vertex graphs at the opening of this paper have twelve automorphisms apiece (verify this as an exercise). Many of the files named `testgraphxx.dat` include as a commented line the calculation of how many automorphisms they should have; these are fun to verify individually, e.g.

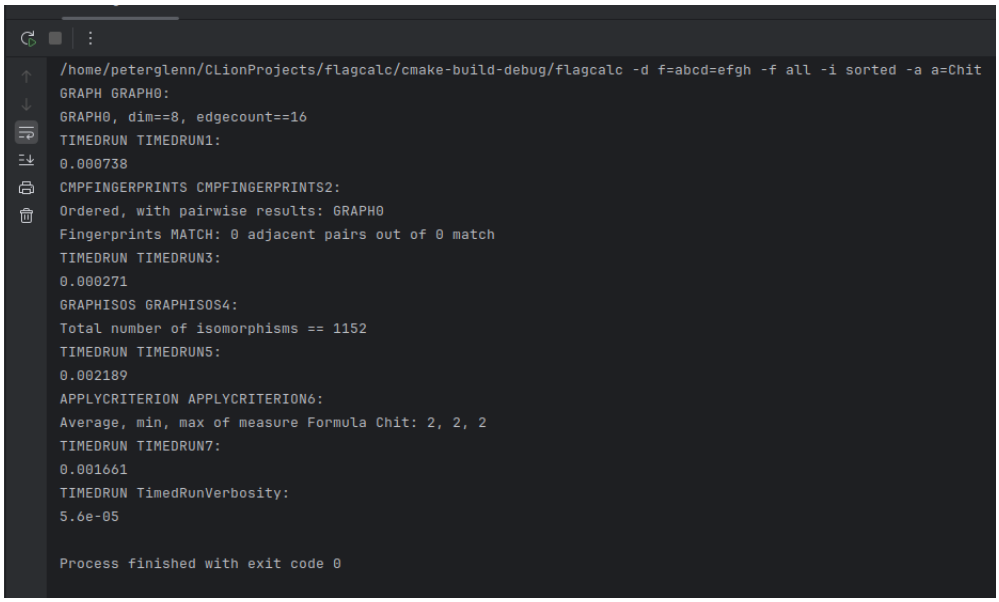


So for example in this case, compute the automorphism count using `-d testgraph14.dat -f all -i sorted -v Iso Noiso`



4. Example Queries: Chromatic Number and Edge-Chromatic Number

As an immediate exercise, use the `-d f="abcd=efgh" -f all -i sorted -a a=Chit` to explore chromatic number in relation to a graph being bipartite.



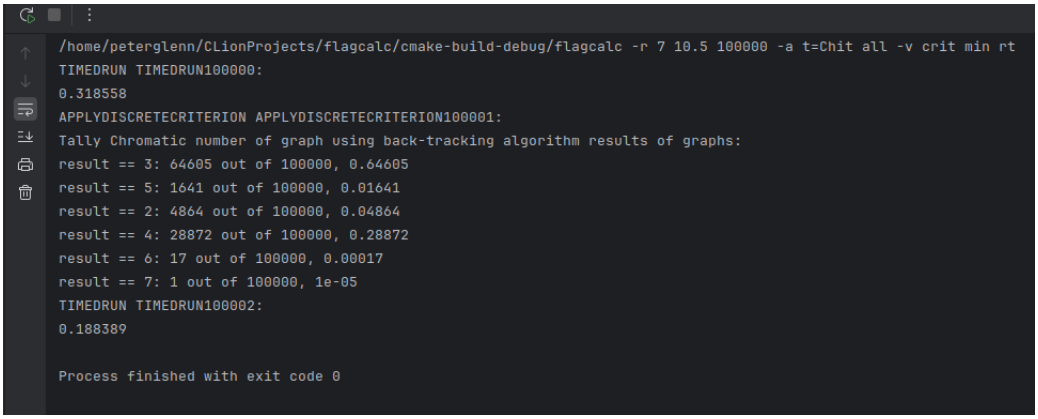
Here, the `-d f=<...>` option uses the notation `abcd=efgh` to request the complete bipartite graph $K_{4,4}$. One can also do $K_{3,4}$ with `abc=defg`, and again, try to figure out the relationship between chromatic number Chit and the particular complete bipartite graph under consideration.

As a next project, explore the same bipartite graphs when it comes to the tally Chiprimet (edge-chromatic number).

5. Queries Generally

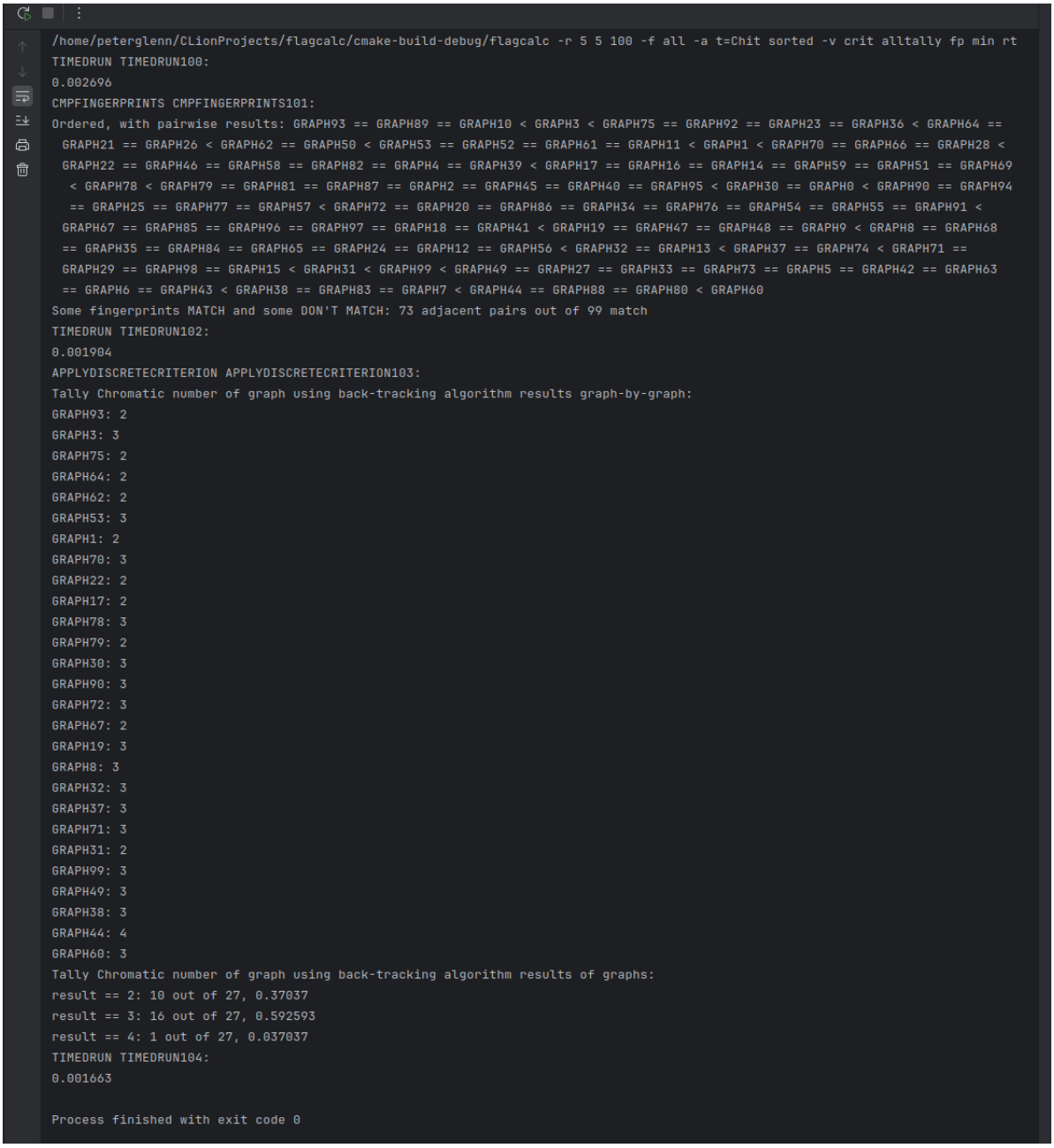
Two schools of thought exist in the way flagcalc is being used (as for example in the scripts under the `scripts` folder): first, since ordering by isomorphism class is slow, many invocations simply bypass `-f` and `-i` and use (often huge numbers of random graphs) the `-a` "query" feature, which is widely capable up to the limits of a weakly-typed full-featured first order querying language. Note that in this case one passes the argument `all` to `-a`, meaning please run the query on *all* graphs on the workspace, as opposed to just the most recent graph. An example is

```
./flagcalc -r 7 10.5 100000 -a t="Chit" all -v crit min rt
```



The other school of thought does invoke `-f all`, and then goes right ahead to the querying with `-a`: here, the keyword `sorted` replaces `all` usually: this means, please run the query once per isomorphism class. That is, by and large our queries are invariant under automorphisms of the graph. One example is:

```
./flagcalc -r 5 5 100 -f all -a t="Chit" sorted -v crit alltally fp min rt
```



Additional queries can be obtained just by looking through the other options delineated by `-h`, such as `m=girthm` (graph's "girth"), `m=radiusm` (graph's "radius"), and so on. These are defined in for example Diestel, R. "Graph Theory" Fifth Edition (2017).

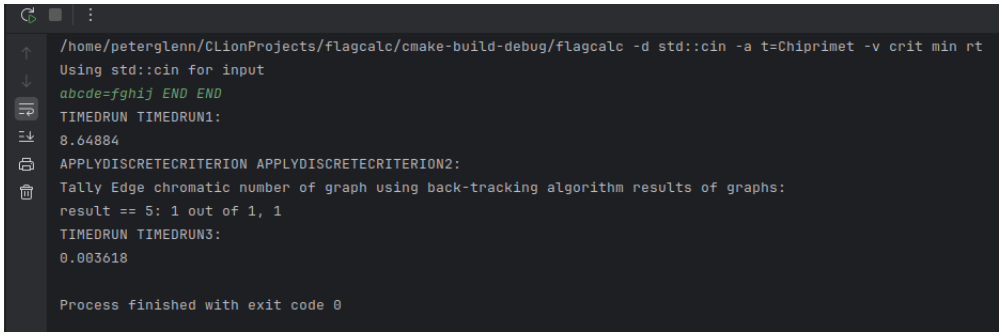
6. Inputting Graphs Generally

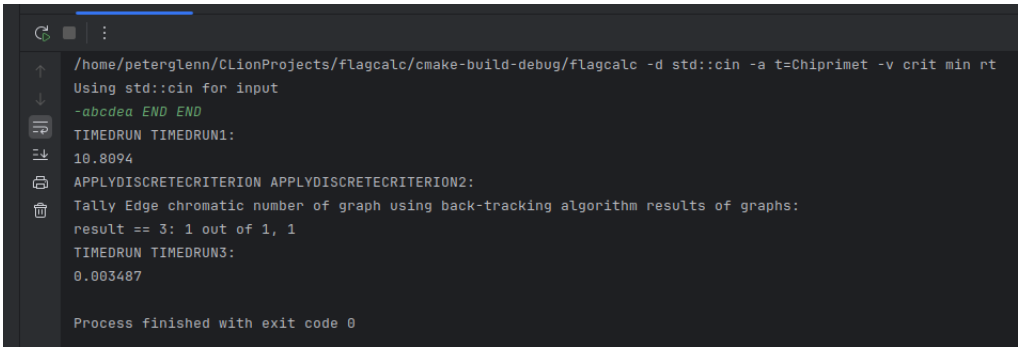
Populate the workspace either with graphs from

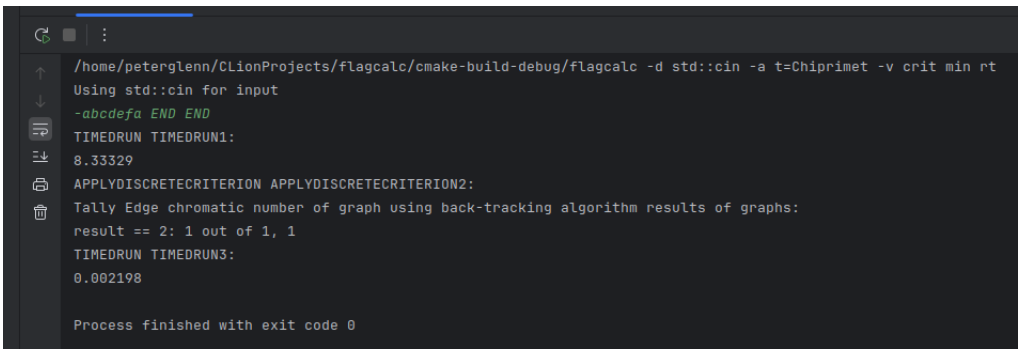
- `-r` Choose from several randomizers (see `-h`), or default to the randomizer that chooses incidence between any two vertices as of likelihood P, where

  `-r <vertexcount> <edgecount> <numberofgraphs>`

  and `edgecount` $= P \cdot \binom{\texttt{vertexcount}}{2}$, where the parentheses are the usual binomial coefficient n-choose-k.
- `-d` Read from the filename in `-d filename.dat`, or from standard input `-d std::cin`, or "inline" graphs with `-d f="-abcda"`. Examples using `std::cin` are:







The "language" for inputting graphs begins with unlimited names for vertices, starting with a letter `a`...`z` or `A`...`Z` (giving 52 options) followed by any number of underscores, colons

and digits `0...9`. There are additional features, such as named vertices using curly brackets; see the README.md file.

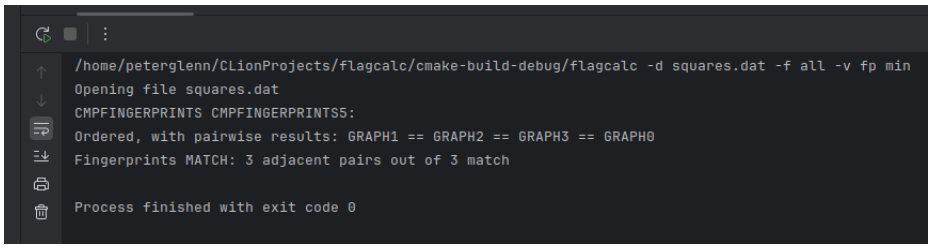
After choosing vertex names, the old-school approach requires a space-delimited list of vertices, followed by `END`. The new school omits the list of vertices and simply deduces from the set of edges the names of the vertices:

```
ab bc cd da END END
```

would be a square (same as the $K_{2,2}$ that opened this paper). Equivalently, the following are also all $K_{2,2}$:

```
ac=bd END END          abcd !ac !bd END END          -abcda END END
```

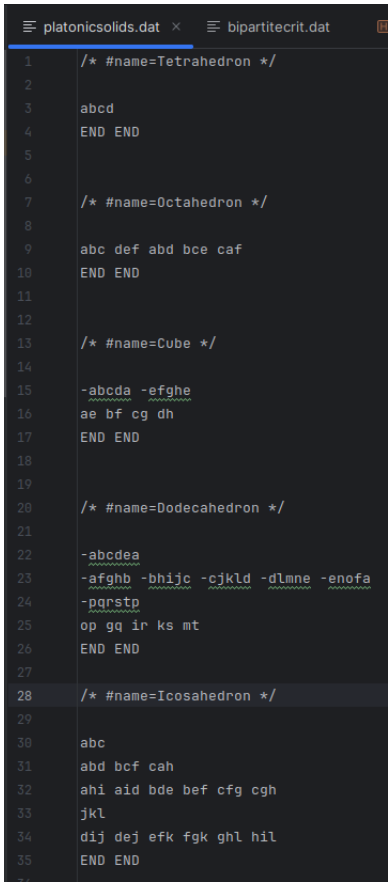
Indeed, we put all four into a file named `squares.dat`, then run flagcalc with `-d squares.dat -f all -v fp min`.



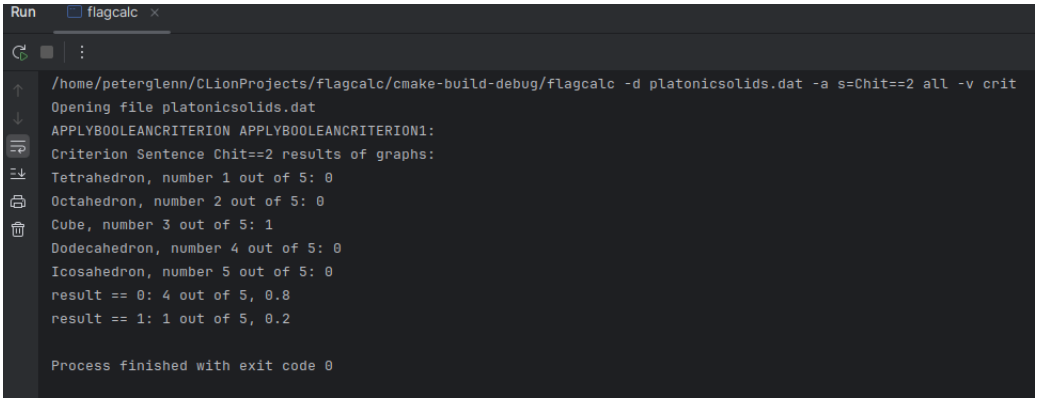
Additional features include tripartite graphs (`abc=def=ghi`) and radial graphs (`a+bcde` or `abc+defg`: draw these out, after asking flagcalc to produce their adjacency matrices)).

7. Automorphisms of the Platonic Solids

The five Platonic solids are encoded for flagcalc in the file `platonicsolids.dat` (as an exercise, try encoding them yourself, or find varying ways to encode them).



Use `-d platonicsolids.dat -f all -i sorted -v Noiso Iso graphs` to compute the number of automorphisms of each Platonic solid. Use criterion (from `-h`) like `conn1c` to affirm that each Platonic solid is indeed connected (1-connected). Use the criterion `-a s="Chit==2"` with `-v crit` to reveal which Platonic solid(s) are two-colorable:



Use `-a e=Cycless all` with verbosity `-v crit allsets` to list the number of cycles on each Platonic graph (be prepared for a large number). Looking ahead to the paper discussing first order queries, run

```
-d platonicsolids.dat -a z="st(SET (c IN Cycless, st(c)==dimm, c))" all -v
crit
```

to compute the number of undirected Hamiltonian cycles on each graph (a Hamiltonian cycle by definition visits each vertex exactly once). `dimm` stands for "graph dimension" (a throwbback, since dimension is a "tally" (aka integer) not a "measure" (aka floating point)). `st` stands for "size tally": it is the size of the "set" found within the parentheses[3].

8. Outputting Graphs Generally

The verbosity options are not presently listed in the `-h` output or documented elsewhere; for now, here is the relevant code:

[3]Replacing `SET` with `SETD` speeds up some thirty percent; `SETD` means, a set that allows multiple appearances of any element (that is, no checks are performed as to whether a new element already exists in the set). It is much faster, and here it is just as correct, since the quantifier ranges once over the elements in the set named `Cycless`. Another way to get the same numbers faster is

```
-d platonicsolids.dat -a z="COUNT (c IN Cycless, st(c) == dimm)" all -v crit
```

Additionally, one can ask for the "ensemble" (French for "set") of undirected Hamiltonian cycles itself:

```
-d platonicsolids.dat -a e="SETD (c IN Cycless, st(c)==dimm, c)" all -v crit allsets
```

Also one can use the set-valued measure `Cyclesvs` that takes one vertex as an argument, and finds all undirected cycles containing that vertex:

```
-d platonicsolids.dat -a z="COUNT (c IN Cyclesvs(0), st(c) == dimm)" all -v crit
```

It is obviously much faster (fifty percent on this data). Finally, nesting quantifiers, the same output obtains as follows:

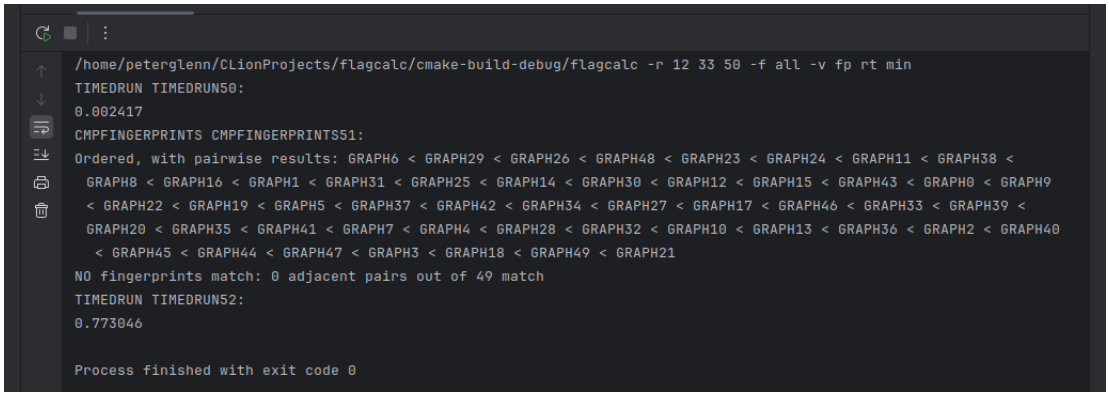
```
-d platonicsolids.dat -a z="COUNT (c IN Cyclesvs(0), FORALL (v IN V, v ELT c))" all -v crit
```

```
#define VERBOSE_CMPFINGERPRINTLEVEL "cmp"
#define VERBOSE_ENUMISOMORPHISMSLEVEL "enum"

#define VERBOSE_DONTLISTISOS "Noiso"
#define VERBOSE_LISTGRAPHS "graphs"
#define VERBOSE_LISTFINGERPRINTS "fp"
#define VERBOSE_ISOS "Iso"
#define VERBOSE_RUNTIMES "rt"
#define VERBOSE_VERBOSITYRUNTIME "vrunt"
#define VERBOSE_VERBOSITYFILEAPPEND "vappend"
#define VERBOSE_MINIMAL "min"
#define VERBOSE_MANTELSTHEOREM "Mantel"
#define VERBOSE_FINGERPRINT "Fp"
#define VERBOSE_SAMPLERANDOMMATCHING "srm"
#define VERBOSE_FPMINIMAL "FpMin"
#define VERBOSE_FPNONE "fpnone"
#define VERBOSE_APPLYCRITERION "crit"
#define VERBOSE_SUBOBJECT "subobj"
#define VERBOSE_PAIRWISEDISJOINT "pd"
#define VERBOSE_APPLYSET "set"
#define VERBOSE_SETVERBOSE "allsets"
#define VERBOSE_CRITVERBOSE "allcrit"
#define VERBOSE_MEASVERBOSE "allmeas"
#define VERBOSE_TALLYVERBOSE "alltally"

#define VERBOSE_APPLYSTRING "strmeas"
#define VERBOSE_APPLYGRAPH "measg"
#define VERBOSE_RANDOMSUMMARY "randomizer"

#define VERBOSE_ALL "Noiso graphs fp Iso rt vrunt vappend min Mantel Fp srm FpMin subobj pd set allsets randomizer"
#define VERBOSE_DEFAULT "Noiso graphs fp Iso rt vrunt vappend min Mantel Fp srm FpMin crit rm fpnone vrunt subobj pd set allsets randomizer"

#define VERBOSE_FORDB "db"
```

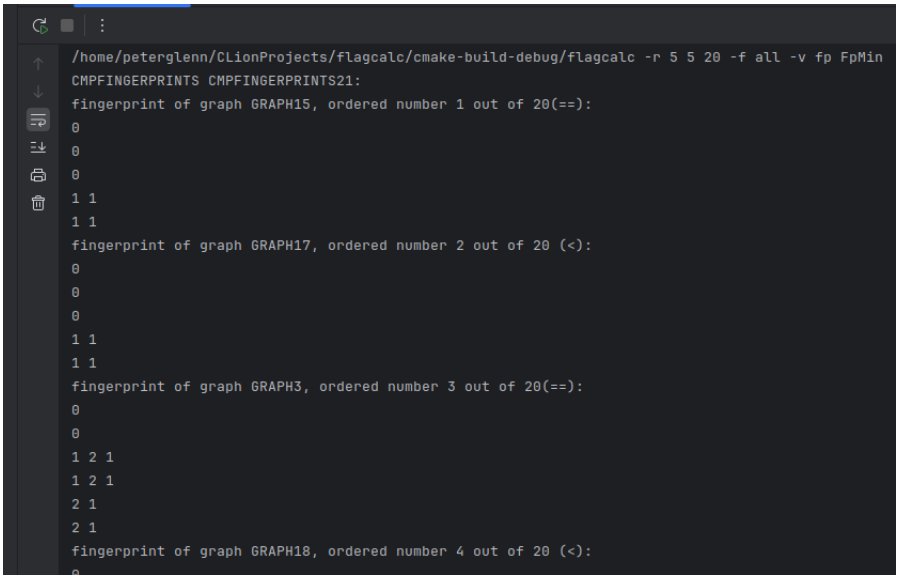
Next, the `-g` feature takes `sorted` to mean one graph per isomorphism class; note the `#FP` entry in each graph's output is 1 or 0 according to whether the graph is non-isomorphic to its immediate successor (or 1 if it is the last graph). Another option, once one is using `-a`, is to pass the argument `passed` to `-g`, meaning "only output those graphs that pass (all) the boolean criteria in `-a`.

9. Further Thoughts

- Exercise: Produce input as a file, as `f="<data>"`, or using `std::cin` for each of the six graphs that open this paper (e.g., the first one, on three vertices, is `f="a bc"`, then `f="ab c"`, and `f="a b c ac"`).
- Exercise: Produce an input file containing ten isomorphic graphs on your choice of vertex count (probably something below eight or nine).
- Exercise: Establish a table of random graphs with $P = 0.50$, that shows the runtime for `-f all` as the number of vertices increases from 2 to a reasonable upper bound (like 12, or depending on how many random graphs you decide to ask for).



- Exercise: find examples of one isomorphism class with varying results of `Chigreedyt`. Practice constructing graphs with increasing differences between `Chigreedyt` and the `Chit`.
- Exercise: Use the graph `*abcde` (which means "the complete graph K_5"; equivalently, `abcde` without the asterisk) to figure out the chromatic number of complete graphs, and the number of automorphisms of K_n generally.
- Exercise: Practice computing "fingerprints" by hand, using verbosity levels `fp` or `FpMin` on one or a few input graphs (hint: "fingerprints" sort vertices by degree, then proceed to adjacent vertices, again sorting by degree).



- Another (research level) question: when working with large numbers of graphs on a given vertex count, what is the average number of equivalence classes of isomorphic graphs?