INTRO TO THE FIRST-ORDER QUERYING LANGUAGE IN FLAGCALC

PETER GLENN

1. Queries

Flagcalc answers "queries" as in the follow general patterns that operate on one hundred graphs of eight vertices and an edge incidence likelihood of $14/\binom{8}{2} = 0.5$:

```
-r 8 14 100 -a s=<bool-valued query> all -v crit allcrit
-r 8 14 100 -a z=<discrete-valued query> all -v crit alltally
-r 8 14 100 -a a=<continuous-valued query> all -v crit allmeas
-r 8 14 100 -a e=<set-valued query> all -v crit allsets
-r 8 14 100 -a p=<tuple-valued query> all -v crit allsets
```

Each of these are passed on the command line to `./flagcalc`, e.g.

```
./flagcalc -r 8 14 100 -a s="cr1" all -v crit allcrit
```

computes the property "triangle-free" (aka `cr1` [1]) for each of the graphs (`allcrit`) or just the likelihood of being triangle-free (replace `-v crit allcrit` with `-v crit min` to see just the respective "`true`" and "`false`" totals out of these hundred graphs). Also, always use verbosity level "`rt`" to output runtime in seconds.

2. The idea behind quantifiers with three inputs

Flagcalc has "first order" quantifiers, taking two or three inputs: first input is a list of variables and sets to draw their values from; next input is optional: a boolean-evaluated "criteria"; and third input is a "value" to use when the criteria evaluates as `True`, for example in the case of quantifier `FORALL`:

$$\texttt{FORALL (}v_1\ \texttt{IN}\ S_1, v_2\ \texttt{IN}\ S_2, \ldots,\ \texttt{<boolean criteria>},\ \texttt{<value>)}$$

where the middle ("criteria") is optional and is evaluated according to the weak-typing of flagcalc into a boolean. Some examples

```
FORALL (v IN V, v > 0, ac(v-1,v))
```

is a basic example that checks each vertex is adjacent to its immediate successor; allowing the use of measure `dimm` (the graph's dimension), this becomes

```
FORALL (v IN V, v+1 < dimm, ac(v,v+1))
```

Date: December 15, 2025.

[1] Other measures have more meaningful names, this was a first one implemented, before the overall picture emerged

Now, matching parentheses is difficult in nested queries; a syntax-highlighting IDE such as the default Ubuntu/your flavor of Linux text editors or JetBrains' can help. For an example of a nested query:

```
FORALL (u IN V, FORALL (v IN V, u == v OR ac(u,v)))
```

or equivalently

```
FORALL (u IN V, FORALL (v IN V, u != v, ac(u,v)))
```

or equivalently as well,

```
FORALL (u IN V, v IN V, u != v, ac(u,v))
```

These have just tested the property of a graph being "complete" (having all possible edges). Check this with

```
cliquem == dimm IFF FORALL (u IN V, FORALL (v IN V, u != v, ac(u,v)))
```

for fun using a larger vertex incidence, e.g. `-r 5 8 100`. Looking ahead to pseudo-second order queries, we have the same test:

```
FORALL (s IN Ps(V), st(s) == 2 IMPLIES s ELT E) IFF cliquem == dimm
```

i.e. "the two statements are equivalent (`IFF`):

- "for all sets s in the powerset of the set of vertices, if the size of s is two then s is an element of the set of edges, and
- "the graph's largest clique (complete subgraph) has dimension equal to the graph's dimension".

If that's not rewarding enough, note some logically equivalent queries, and along the way see `Knc` which takes a clique size and a minimal number of embeddings (returning `true` if the graph has at least that minimal number of complete subgraphs on the given vertex size):

```
FORALL (s IN Ps(V), st(s) == 2 IMPLIES s ELT E) IFF Knc(dimm,1)
```

or,

```
FORALL (s IN Ps(V), st(s) == 2, s ELT E) IFF cliquem == dimm
```

or,

```
FORALL (s IN Ps(V), st(s) == 2, s ELT E) IFF st(E) == nchoosek(dimm,2).
```

For a little peek, the two parameters passed to `Knc` are *n* and *mincnt*:

and there are analogous "tallies" (integer-valued measures) `Knt`, taking one argument.

3. Flagcalc is weakly-typed

FlagCalc is weakly-typed, meaning it tries to fit a typecast variable into whatever type is expected in the context. Some of the types in flagcalc are integer ("discrete"), real ("continuous"), and boolean ("bool"). Others include set, tuple, string, and graph. Sets and tuples can be nested (and here it is very interesting to study algorithmic complexity classes).

For example, treating the set `{0,2,3}` as a discrete (integer) outputs its cardinality, 3. Treating a continuous value like 3.5 as a discrete value outputs 3 as well. Treating a discrete or continuous value as a bool outputs 1 if non-zero, and 0 otherwise. Treating a tuple as a set simply "forgets" the ordering and removes duplicates. Treating a set as a tuple simply orders it randomly (or to be precise, in whatever order the set was put together).

Please get accustomed to seeing `0` for `false` and `1` for `true`; we are hard-core, after all. Also, please note the association:

"C++"-style type	flagcalc type	flagcalc measure name	flagcalc measure suffix
bool	mtbool	criterion	c
integer	mtdiscrete	tally	t
float	mtcontinuous	measure	m
<various>	mtset	set	s
<various>	mttuple	tuple	p
<unimplemented>	mtgraph	graph	g
std::string	mtstring	string	r

4. A list of first-order quantifiers

We can now create many features around this list of implemented "quantifiers": each considers the value only when the criteria passes.

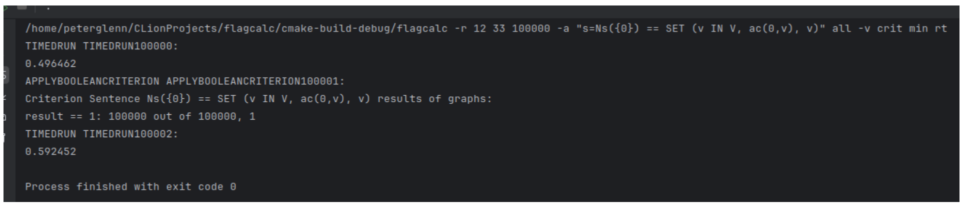
- `FORALL` and `EXISTS`: these are familiar as $\forall$ and $\exists$ in standard first-order logic (as in, e.g., H. Enderton's "A Mathematical Introduction to Logic" (2001, 1972) textbook).
- `SUM` and `TALLY`: the continuous (resp. discrete) values are added up.
- `PRODUCT`: the product of all values
- `COUNT`: same as `TALLY` with value constant 1: `COUNT` omits the "value" input
- `AVERAGE`: the mean, that is, the average of all values (shorthand for `SUM...` divided by `COUNT ...`)
- `MIN` and `MAX`: what is the minimum/maximum of the associated value (including infinity or negative infinity)
- `RANGE`: shorthand (and algorithmic improvement) for `MAX...` minus `MIN...`
- `MEDIAN` *Not implemented*
- `MODE` *Not implemented*
- `SET` and `SETD`: form a set consisting of each of the values, and in case of `SET` remove duplicates; do not waste time removing duplicates when `SETD` is used
- `TUPLE`: form a tuple (an ordered set) consisting of each of the values in the order they arise

- `BIGCUP`, `BIGCUPD`, and `BIGCAP`: borrowing from LaTeXnotation, when values are sets, find their union $\bigcup$ (resp. intersection $\bigcap$); and for `BIGCUPD` like in `SETD`, do not waste time removing duplicates

5. Fun with adjacency

Now it connects to the built-in `Ns`, which takes a set as an input and returns the vertices that neighbor this set (excluding those within the set):

`-r 12 33 100000 -a s="Ns({0}) == SET (v IN V, ac(0,v), v)" all -v crit min rt`



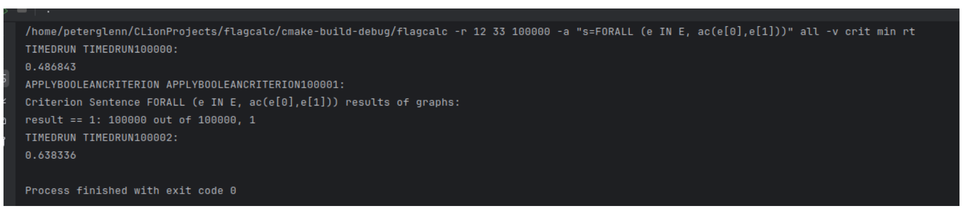
```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 12 33 100000 -a "s=Ns({0}) == SET (v IN V, ac(0,v), v)" all -v crit min rt
TIMEDRUN TIMEDRUN100000:
0.496462
APPLYBOOLEANCRITERION APPLYBOOLEANCRITERION100001:
Criterion Sentence Ns({0}) == SET (v IN V, ac(0,v), v) results of graphs:
result == 1: 100000 out of 100000, 1
TIMEDRUN TIMEDRUN100002:
0.592452

Process finished with exit code 0
```

See that all 100000 sample random graphs return `1` aka `True`.

Additionally a set can be dereferenced: the elements of `E` are sets of size two.

`-r 12 33 100000 -a s="FORALL (e IN E, ac(e[0],e[1]))" all -v crit min rt`



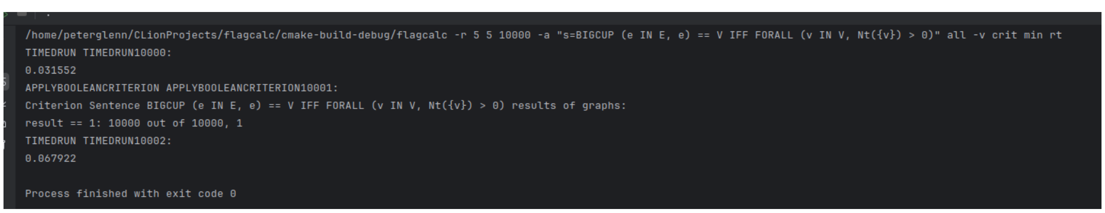
```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 12 33 100000 -a "s=FORALL (e IN E, ac(e[0],e[1]))" all -v crit min rt
TIMEDRUN TIMEDRUN100000:
0.486843
APPLYBOOLEANCRITERION APPLYBOOLEANCRITERION100001:
Criterion Sentence FORALL (e IN E, ac(e[0],e[1])) results of graphs:
result == 1: 100000 out of 100000, 1
TIMEDRUN TIMEDRUN100002:
0.638336

Process finished with exit code 0
```

Please note a feature/bug: to dereference several layers deep, one needs parentheses: for example, if dereferencing two layers deep, then instead of `s[1][2]` use `(s[1])[2]`.

And measures like `Nt` ("Neighbor Tally") are occasion to experiment with `BIGCUP`:

```
-r 5 5 10000 -a s="BIGCUP (e IN E, e) == V IFF FORALL (v IN V, Nt({v}) > 0)"
    all -v crit min rt
```



```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 5 5 10000 -a "s=BIGCUP (e IN E, e) == V IFF FORALL (v IN V, Nt({v}) > 0)" all -v crit min rt
TIMEDRUN TIMEDRUN10000:
0.031552
APPLYBOOLEANCRITERION APPLYBOOLEANCRITERION10001:
Criterion Sentence BIGCUP (e IN E, e) == V IFF FORALL (v IN V, Nt({v}) > 0) results of graphs:
result == 1: 10000 out of 10000, 1
TIMEDRUN TIMEDRUN10002:
0.067922

Process finished with exit code 0
```

or something unnatural, that reveals the underlying ordered nature of sets:

```
-r 12 33 100000
    -a s="FORALL (e1 IN E, e2 IN E, NOT (e1[0]==e2[1] AND e1[1]==e2[0]))"
          all -v crit min rt
```

```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 12 33 100000 -a "s=FORALL (e1 IN E, e2 IN E, NOT (e1[0]==e2[1] AND e1[1]==e2[0]))" all -v crit min rt
TIMEDRUN TIMEDRUN100000:
0.481745
APPLYBOOLEANCRITERION APPLYBOOLEANCRITERION100001:
Criterion Sentence FORALL (e1 IN E, e2 IN E, NOT (e1[0]==e2[1] AND e1[1]==e2[0])) results of graphs:
result == 1: 100000 out of 100000, 1
TIMEDRUN TIMEDRUN100002:
2.01385

Process finished with exit code 0
```

Indeed, the following example returns all true:

-r 12 33 100000 -a s="FORALL (e IN E, e[0]<e[1])" all -v crit min rt

All this makes perfect sense to a logician, but is horribly confusing to someone new to first order logic. Therefore, the rest of this paper will focus on concrete examples of getting things done. But first, some extensions for the expert.

6. Extensions

6.1. **Naming.** In C++ one can use "using" to give a shorthand for an expression. Here, without the A.I. or automatic code speedups, we manually tell flagcalc that we want to compute "expression" once, then refer to it by a given "name".

The word NAMING takes two comma-delimited inputs: a name and an expression.

NAMING (s_1 AS E_1, s_2 AS E_2, , ..., <expression>)

For example,

FORALL (v IN V, NAMING (ns AS SETD (w IN V, ac(v,w), w), st(ns) < dimm AND st(ns) % 2 == 0)

This computes the set "ns" of immediate neighbors to a given v: is the size of ns less than the graph's dimension ("dimm") and an even number (zero mod two)?

Immediately this led to a dilemma, the fact that quantifiers have both a criteria and a value section, so NAMING cannot bridge that comma delimiter. The solution is to allow the user to omit the word NAMING and build in namings into the quantifier itself using just AS:

FORALL (v IN V, ns AS SETD (w IN V, ac(v,w), w), st(ns) > 0, dimm/st(ns) < 0.5)

6.2. **Concurrency.** Very briefly, any flagcalc quantifier can be preceded by THREADED, and a few by now are implemented to be preceded by GPU:

THREADED FORALL (s IN Ps(V), st(s) > 0, EXISTS (p IN Cyclesvs(s[0]), p <= s)

Because the powerset "Ps(V)" of the set of vertices grows quickly in size, here the CPU's threads are split amongst the subsets of V, and the question is then posed as to whether there is a cycle originating in the first element in the subset s that is contained entirely within s.

6.3. **Partitions and Sortings.** The "relational" operators PARTITION and SORT have the same syntax as the quantifiers above, except only take the first two out of three inputs.

- PARTITION: In the variables section (first section) use two instead of one variable. In the middle "criteria" section give some boolean criteria involving those two variables (that is an equivalence relation: symmetric, transitive, and reflexive). The result is the set of sets of values from the set quantified over, i.e. partitioned by "criteria":

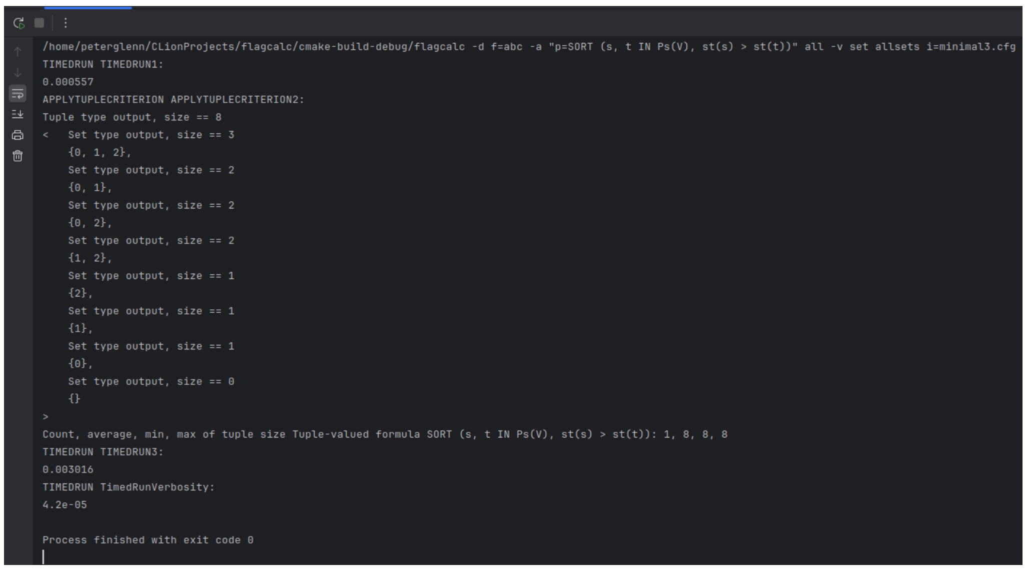
`-r 120 70 1 -a s="st(THREADED PARTITION (u,v IN V, connvc(u,v))) == connm" all -v set allsets i=minimal3.cfg`

checks that the size of the set partitioning by the notion of "connectedness" is the same as the measure "`connm`".

- `SORT`: In this case the criteria is an ordering. The result is like in `PARTITION`, but instead of a set a tuple.

  `-d f="abc" -a p="SORT (s, t IN Ps(V), st(s) > st(t))" all -v set allsets i=minimal3.cfg`

  sorts the powerset on three elements by set size.



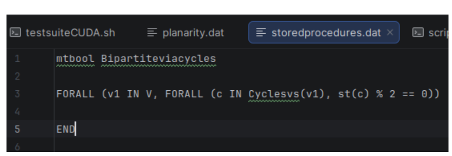
6.4. **Stored Procedures.** One of the most powerful features of Flagcalc as far as code maintenance and readability, is the ability to "code" in the query language: we would like to invoke, say,

`-a s="NOT cr1 IMPLIES NOT Bipartiteviacycles"`

or more naturally

`-a s="cr1 IF Bipartiteviacycles"`

that is, "triangle-free if bipartite". So how, if `Bipartiteviacycles` is not a built-in measure (aka not found in the output of `./flagcalc -h`)? See if the following excerpt from a file named `storedprocedures.dat` makes sense



It is using modular arithmetic to make sure all cycles have an even length.

We use it all together as in

```
-d f="abc=defg" -a isp="../scripts/storedprocedures.dat" s="cr1 IF
Bipartiteviacycles" all -v crit.
```

Please note that stored procedures can be recursive or call each other even if declared in the wrong order; that a stored procedures .dat file can "include" other .dat files, and that stored procedures (a term borrowed from SQL) can take any number of arguments, as in

```
mtbool Cycleoflength ( mtdiscrete n )

EXISTS (c IN Cycless, st(c) == n)

END
```

To add comments, bracket them with `/*` and `*/`; to "include" another stored procedure file, use `#include "<filename.dat>"`, and please be inspired by glancing at the various examples in flagcalc's default `scripts` folder (such as, `storedprocedures.dat` and `planarity.dat`).

6.5. "If-then-else" statements.

The tool naturally leads to questions of flow control. A direct example is the notation (borrowed from C) of

$$\phi \texttt{ ? a : b}$$

meaning

If ϕ evaluates `true` then the value is a, and otherwise it is b

Of course, this is more than what it is in C: in flagcalc it is a genuine feature not emulated in more wordy form. This may be a place for new features down the line, but one aim is to think in terms of a first-order rather than procedural language.

Secondly, there is "clever" flow control, when the flagcalc query coder recognizes the tool parses from right to left, one uses the more costly or more unlikely statement to *the left* of the less costly or likely one. For example

```
-r 8 14 100 -a s="FORALL(P IN Setpartition(V), st(P) > 3 OR EXISTS (S IN
Ps(V), st(S) == st(P), FORALL (p IN P, EXISTS (v IN S, v ELT p))))" all -v
crit min rt
```

runs in eleven seconds on a 24-core AMD Threadripper, while

```
-r 8 14 100 -a s="FORALL(P IN Setpartition(V), EXISTS (S IN Ps(V), st(S)
== st(P), FORALL (p IN P, EXISTS (v IN S, v ELT p))) OR st(P) > 3)" all -v
crit min rt
```

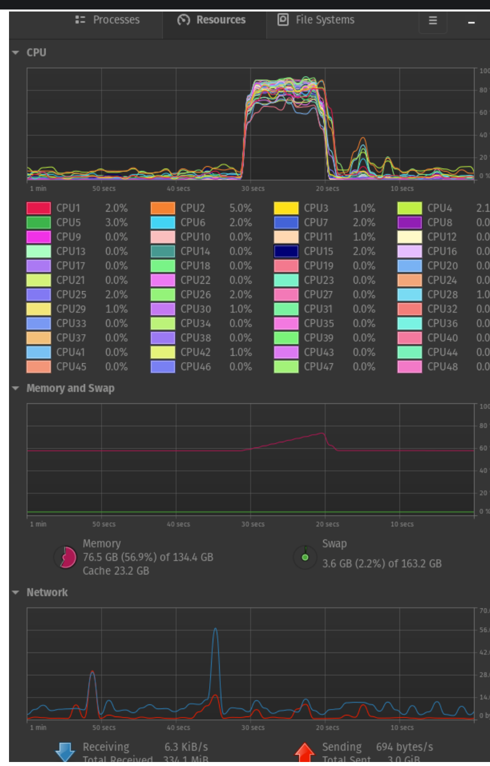
runs in a fraction of the time: one point three seconds on the same computer.

```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 8 14 100 -a "s=FORALL(P IN Setpartition(V), st(P) > 3 OR EXISTS (S IN Ps(V), st(S)
 == st(P), FORALL (p IN P, EXISTS (v IN S, v ELT p))))" all -v crit min rt
TIMEDRUN TIMEDRUN100:
0.002905
APPLYBOOLEANCRITERION APPLYBOOLEANCRITERION101:
Criterion Sentence FORALL(P IN Setpartition(V), st(P) > 3 OR EXISTS (S IN Ps(V), st(S) == st(P), FORALL (p IN P, EXISTS (v IN S, v ELT p)))) results of
 graphs:
result == 1: 100 out of 100, 1
TIMEDRUN TIMEDRUN102:
10.669

Process finished with exit code 0
```

```
/home/peterglenn/CLionProjects/flagcalc/cmake-build-debug/flagcalc -r 8 14 100 -a "s=FORALL(P IN Setpartition(V), EXISTS (S IN Ps(V), st(S) == st(P),
 FORALL (p IN P, EXISTS (v IN S, v ELT p))) OR st(P) > 3)" all -v crit min rt
TIMEDRUN TIMEDRUN100:
0.002765
APPLYBOOLEANCRITERION APPLYBOOLEANCRITERION101:
Criterion Sentence FORALL(P IN Setpartition(V), EXISTS (S IN Ps(V), st(S) == st(P), FORALL (p IN P, EXISTS (v IN S, v ELT p))) OR st(P) > 3) results of
 graphs:
result == 1: 100 out of 100, 1
TIMEDRUN TIMEDRUN102:
1.31666

Process finished with exit code 0
```



Please note one extant bug in flagcalc (see `scripts/bugs.dat`) has to do with certain quantifiers with the feature (not documented, except in the `testsuitethreaded.sh` script) of `THREADED` quantifiers; while this bug affects certain threaded queries, please note that automatically, that is already with the quantifiers in the present paper, the query is threaded across each of the distinct graphs from, say in this example, one hundred random graphs, as shown in the system monitor screenshot. The bug has to do with attempting to thread *within* one graph's vertex powerset, for example, that is, explicitly when using the keyword `THREADED`.

Here is another query, to illustrate something natural around the set-valued measure `Setpartition` again, with returns a set of all sets that are partitions of `V` into non-empty subsets. Note it is a desired feature also to have a `Sizedsetpartition` that takes *two* arguments, one being the partition size (for example, its cousin, `Sizedsubset` takes an input set and an input size for the subsets returned).

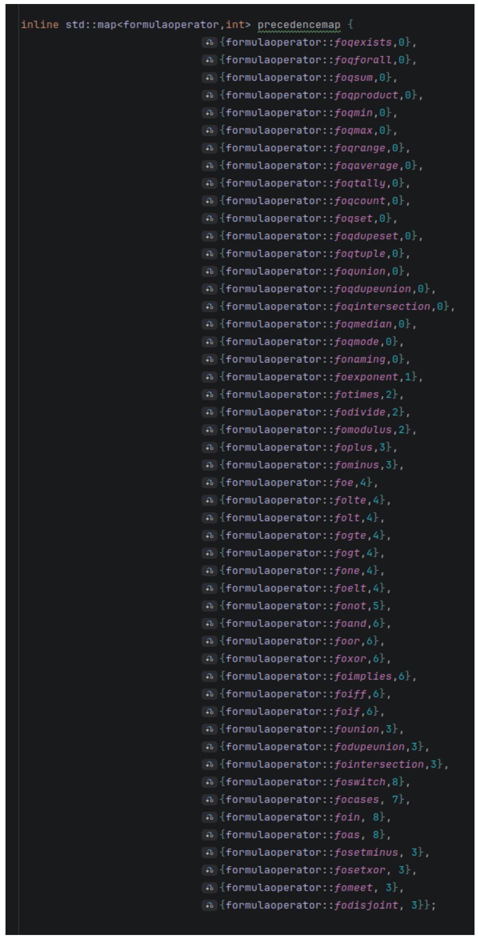
```
FORALL(s IN Setpartition(V), EXISTS (v1 IN V, v2 IN V, v3 IN V, v1 != v2
AND v1 != v3 AND v2 != v3, EXISTS (s1 IN s, s2 IN s, s3 IN s, v1 ELT s1
AND v2 ELT s2 AND v3 ELT s3)) OR st(s) < 3)
```

7. Examples

A logician will tell you there are basic logical connectives

(1) ψ `IF` ϕ
(2) ϕ `IMPLIES` ψ
(3) ϕ `IFF` ψ
(4) ψ `AND` ϕ
(5) ϕ `OR` ψ
(6) ϕ `XOR` ψ
(7) `NOT` ϕ

When it comes to queries around logical equivalence, the usual connector is `IFF`, which functions identically to `==` (that is, it does not first cast into boolean types), but with differing operator precedence, so `NOT (3 IFF 4)`, `NOT 3 == 4)` (and by operator precedence, `NOT 3 == 1`)). Please see the easily-adjusted precedence chart in the Flagcalc source code:



```
inline std::map<formulaoperator,int> precedencemap {
                                        {formulaoperator::foqexists,0},
                                        {formulaoperator::foqforall,0},
                                        {formulaoperator::foqsum,0},
                                        {formulaoperator::foqproduct,0},
                                        {formulaoperator::foqmin,0},
                                        {formulaoperator::foqmax,0},
                                        {formulaoperator::foqrange,0},
                                        {formulaoperator::foqaverage,0},
                                        {formulaoperator::foqtally,0},
                                        {formulaoperator::foqcount,0},
                                        {formulaoperator::foqset,0},
                                        {formulaoperator::foqdupeset,0},
                                        {formulaoperator::foqtuple,0},
                                        {formulaoperator::foqunion,0},
                                        {formulaoperator::foqdupeunion,0},
                                        {formulaoperator::foqintersection,0},
                                        {formulaoperator::foqmedian,0},
                                        {formulaoperator::foqmode,0},
                                        {formulaoperator::fonaming,0},
                                        {formulaoperator::foexponent,1},
                                        {formulaoperator::fotimes,2},
                                        {formulaoperator::fodivide,2},
                                        {formulaoperator::fomodulus,2},
                                        {formulaoperator::foplus,3},
                                        {formulaoperator::fominus,3},
                                        {formulaoperator::foe,4},
                                        {formulaoperator::folte,4},
                                        {formulaoperator::folt,4},
                                        {formulaoperator::fogte,4},
                                        {formulaoperator::fogt,4},
                                        {formulaoperator::fone,4},
                                        {formulaoperator::foelt,4},
                                        {formulaoperator::fonot,5},
                                        {formulaoperator::foand,6},
                                        {formulaoperator::foor,6},
                                        {formulaoperator::foxor,6},
                                        {formulaoperator::foimplies,6},
                                        {formulaoperator::foiff,6},
                                        {formulaoperator::foif,6},
                                        {formulaoperator::founion,3},
                                        {formulaoperator::fodupeunion,3},
                                        {formulaoperator::fointersection,3},
                                        {formulaoperator::foswitch,8},
                                        {formulaoperator::focases, 7},
                                        {formulaoperator::foin, 8},
                                        {formulaoperator::foas, 8},
                                        {formulaoperator::fosetminus, 3},
                                        {formulaoperator::fosetxor, 3},
                                        {formulaoperator::fomeet, 3},
                                        {formulaoperator::fodisjoint, 3}};
```

In addition Flagcalc offers usual mathematical symbols, and a `!=` b for `NOT (`a `==` b`)`.

Let us take a simple example, the definitions of cycles, walks, and paths. Flagcalc has a set-valued measure called `Pathss` taking two inputs, vertex u and vertex v. Insofar as a *walk* is sometimes a path but every path is a walk, we note measures like `nwalksbetweenp` and its fast CUDA analog that are designed around the simple fact from matrix algebra, that the n-th power of the adjacency matrix contains the number of walks of length n between vertices u and v in the coordinate $[u, v]$. This may seem forced or tricky to use; paths are more natural in that they are a sequence of vertices $v_0, v_1, ..., v_n$ such that each v_i is distinct, and such that there is an edge between any v_i and v_{i+1}. Walks, meanwhile, are like paths except that may visit any vertex repeatedly.

Noting this dearth of flagcalc native help with *walks*, let us code a stored procedure called `nWalkss` taking three mtdiscrete inputs (the two vertices and the walk length sought) and returning a set of tuples:

```
mtset nWalkss( mtdiscrete u, mtdiscrete v, mtdiscrete length )
```

```
SET (m IN Maps(length+1, dimm),
     FORALL (n IN st(m) - 1, ac(m[n],m[n+1])) AND m[0] == u AND m[st(m)-1] == v, m )

END
```

Now, a walk is a path precisely when each vertex it visits is distinct:

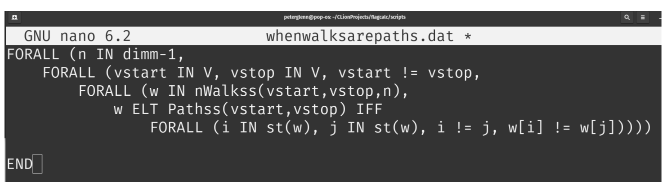
```
FORALL (n IN dimm-1,
    FORALL (vstart IN V, vstop IN V, vstart != vstop,
        FORALL (w IN nWalkss(vstart,vstop,n),
            w ELT Pathss(vstart,vstop) IFF
                FORALL (i IN st(w), j IN st(w), i != j, w[i] != w[j]))))
```

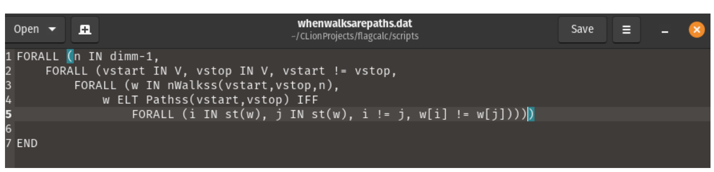
We invoke this with very low numbers in the interest of actually getting a response back:

```
-r 6 7.5 100 -a isp="../scripts/storedprocedures.dat" s="<the sentence
                    above>" all -v i=minimal3.cfg
```

Alternately (a variant on making this query into a stored procedure), there is a feature called `is` that takes a filename and executes the mtbool-valued query in `<filename>`: save the above query to a file named `whenwalksarepaths.dat`, end the file with a solo `END`, and execute:

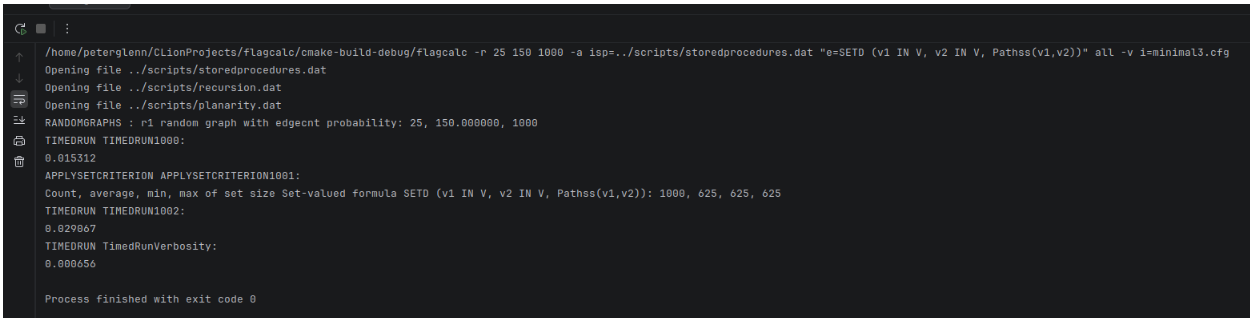
```
-r 6 7.5 100 -a isp="../scripts/storedprocedures.dat"
is="../scripts/whenwalksarepaths.dat" all -v i=minimal3.cfg
```





(please indulge the opportunity taken: to show a syntax-highlighter helping see where the parentheses match, not part of the otherwise well-used editor on the top image.)

Now, why the intractability? A quick check and the built-in `Pathss` runs in efficient time:



This comes down to a first lesson: computational complexity of "second order" notions like powerset and maps (the latter being the set of all maps between an n-sized set and an m-sized set). The latter, for example, numbers m^n in size: a number that grows very quickly. So we like to "invert" (to coin a term) our approach to things like the set of all n-walks, and code them natively in a procedural language.

Next, the tuple-valued measure `nwalksbetweenp` returns a two-dimensional matrix of the count of nwalks between the corresponding vertices. Observe

```
-d f="abc -defgd" -a isp="../scripts/storedprocedures.dat" p="TUPLE
(vstart IN V, TUPLE (vstop IN V, st(nWalkss(vstart,vstop,3))))"
p="nwalksbetweenp(3)" all -v crit allcrit min allsets rt
```

returns two lines of matching tuples of tuples (2d matrices). So let us verify our work on `nWalkss`:

```
s="FORALL (n IN 4, vstart IN V, vstop IN V, st(nWalkss(vstart,vstop,n)) ==
(nwalksbetweenp(n)[vstart])[vstop])"
```

or

```
s="FORALL (n IN 4, TUPLE (vstart IN V, TUPLE (vstop IN V,
st(nWalkss(vstart,vstop,n)))) == nwalksbetweenp(n))"
```

They run in comparable time.