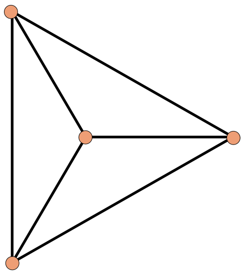
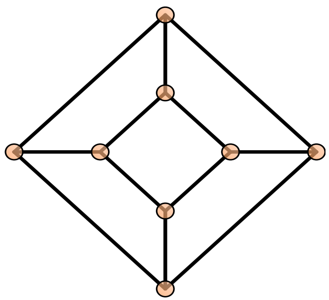
Fast Generation of Cubic Graphs

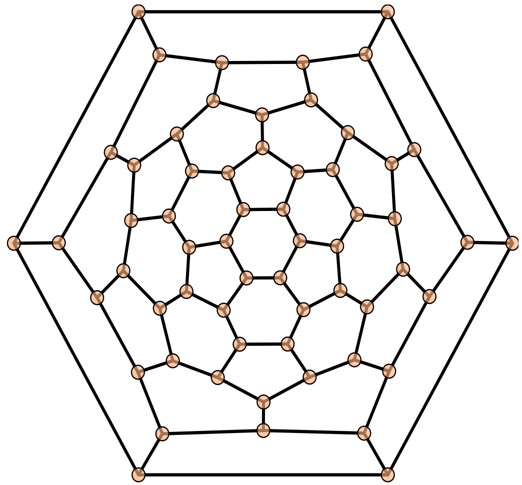
Gunnar Brinkmann and Jan Goedgebeur

Gunnar.Brinkmann@UGent.be
Jan.Goedgebeur@UGent.be

Cubic graphs are graphs where every vertex has degree 3 (3 edges meet in every vertex).







They are very interesting in chemistry as models of molecules (vertices are e.g. Carbon atoms as in fullerenes)

They are very interesting in mathematics (for a lot of conjectures smallest possible counterexamples are cubic graphs)

Enumeration of cubic graphs

1889 De Vries – up to 10 vertices

1966/67 Balaban – up to 12 vertices (computer)

1968 Bussemaker, Seidel – up to 10 vertices (by hand)

1971 Imrich – up to 10 vertices (by hand)

1974 A.L. Petrenjuk, A.W. Petrenjuk – up to 12 vertices

1976 Bussemaker, Cobeljic, Cvetkovic, Seidel – up to 14 vertices

1976 Faradzev – up to 18 vertices

1985 McKay, Royle – up to 20 vertices

1992 Brinkmann – up to 24 vertices
(In the meantime the same program – *minibaum* – has been used up to 30 vertices, that are 845.480.228.069 graphs.)

1999 Meringer – general regular graphs generator – faster for small vertex numbers, slower for large vertex numbers.

2000 McKay, Sanjmyatav – fast specialized algorithm, but was never released

The following algorithm is faster than any previously developed algorithm.

It uses a construction that is folklore (and was already used by De Vries and McKay, Sanjmyatav), some standard isomorphism rejection techniques (McKay's canonical construction path method), well known efficient datastructures. . .

. . . plus one **simple** new idea.

The generation algorithm consists of 2 steps:

Tetrahedron

$\Downarrow$

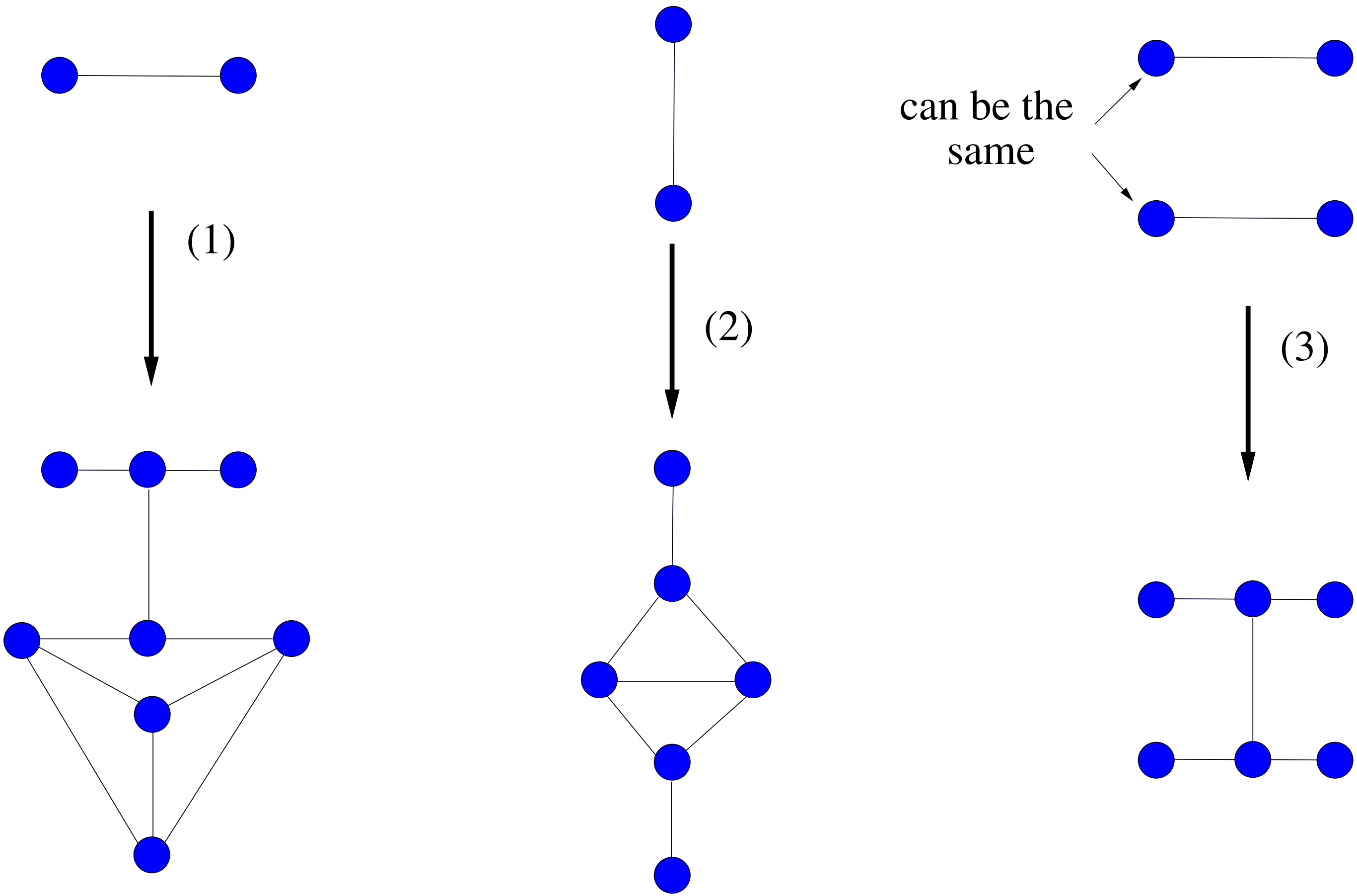
Prime graphs

$\Downarrow$

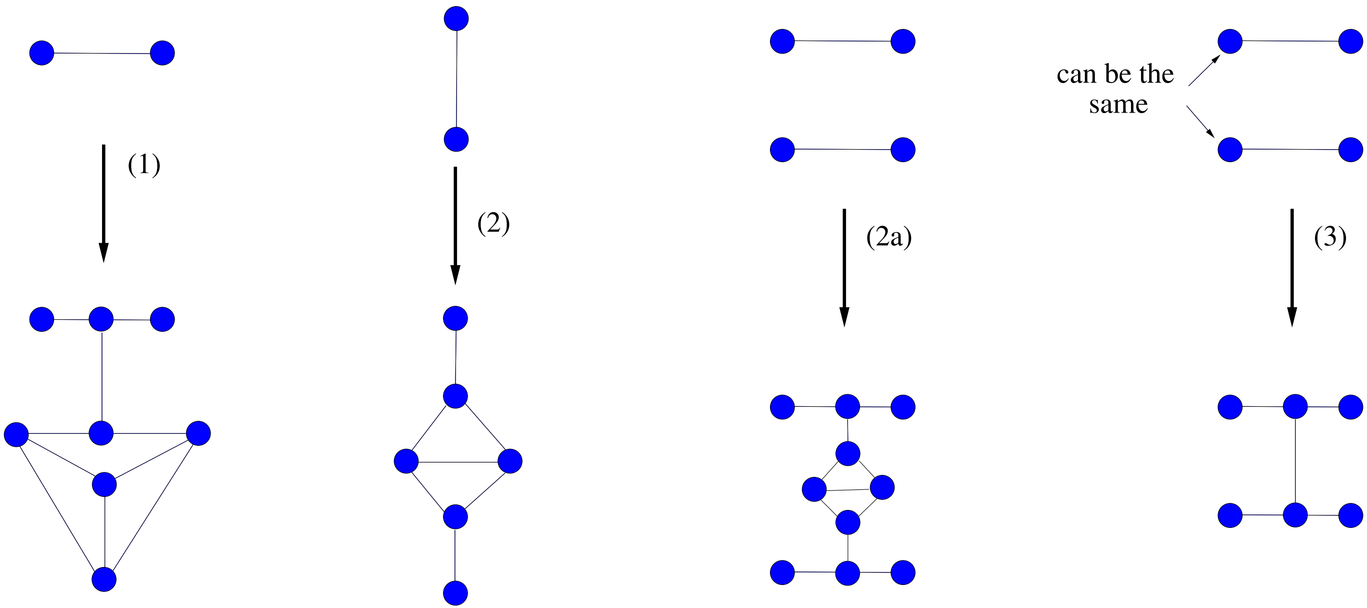
All (remaining) cubic graphs

Operations of De Vries (1889)

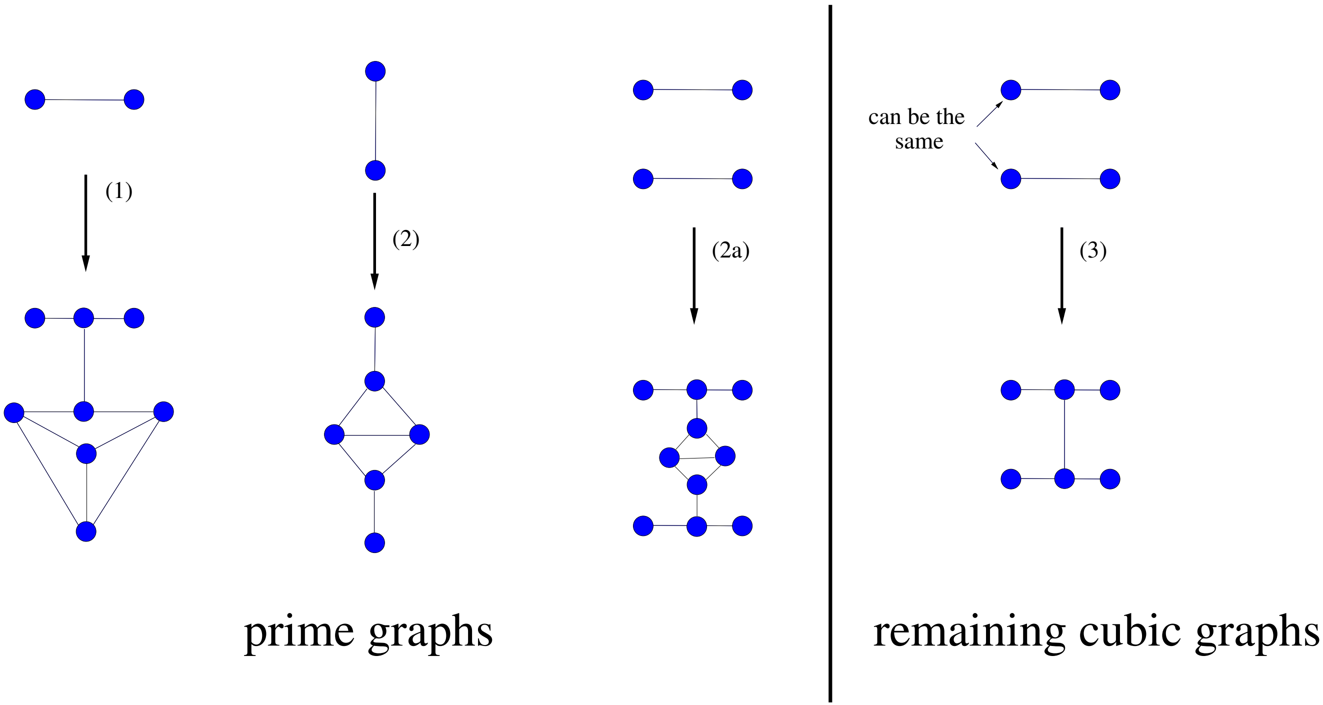
Start from the tetrahedron. . .



Small modification

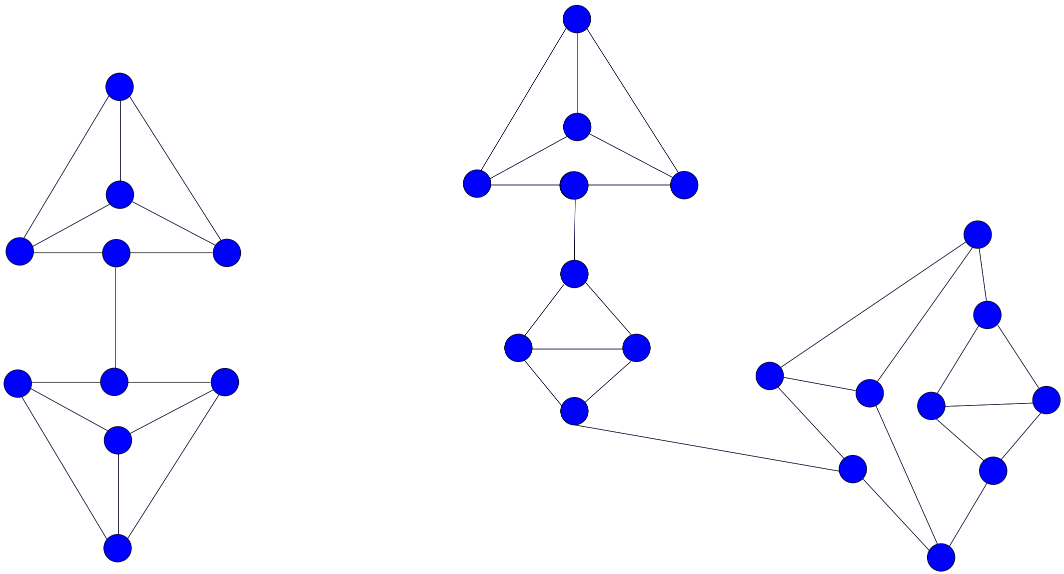


Operations to generate . . .



Now first only the operations (1),(2),(2a) can be applied – and operation (3) last.

So we first generate all graphs of the form



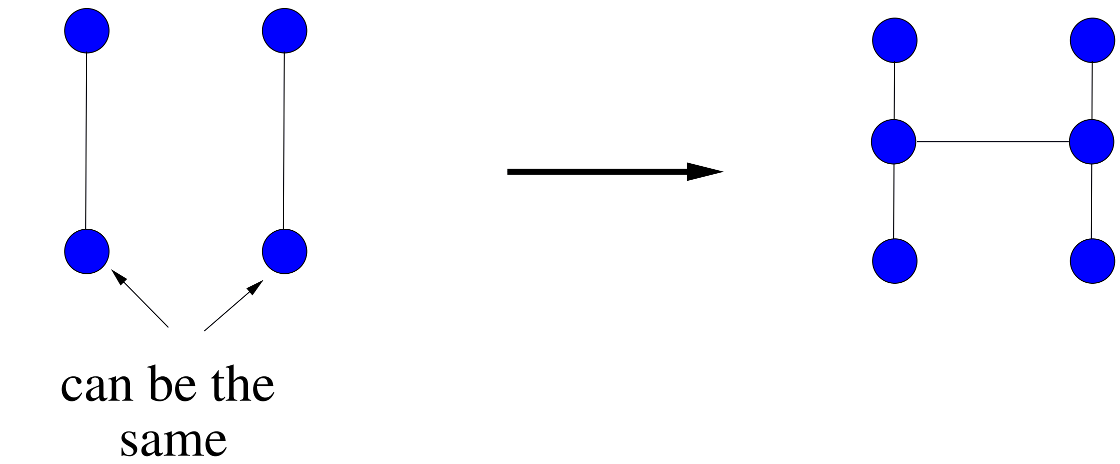
. . . these are the prime graphs.

There are relatively few prime graphs – for 26 vertices 0.0000025% of all graphs – and the rate is decreasing fast.

So **for this part** of the generation, efficiency is not the issue.

Generation of remaining cubic graphs

The following operation remains
(and completely determines the efficiency):



Isomorphism rejection (McKay's canonical construction path method)

- Assign a unique **inverse** operation for every graph (except the prime graphs) to obtain an ancestor.

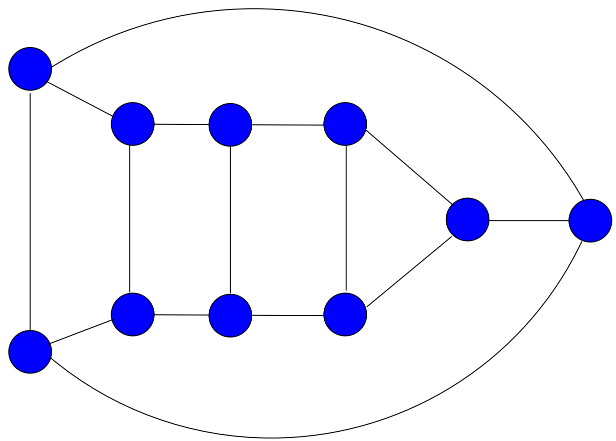
- Make sure that from the same graph you don't obtain the same ancestor twice *in the same way*.

- Assign a unique **inverse** operation for every graph (except the startgraphs) to obtain an ancestor:

Assign *up to isomorphism* an edge that must be removed (i.e. the *canonical edge*). First use some cheap criteria and only in case these don't help use *nauty* to compute a canonical form.

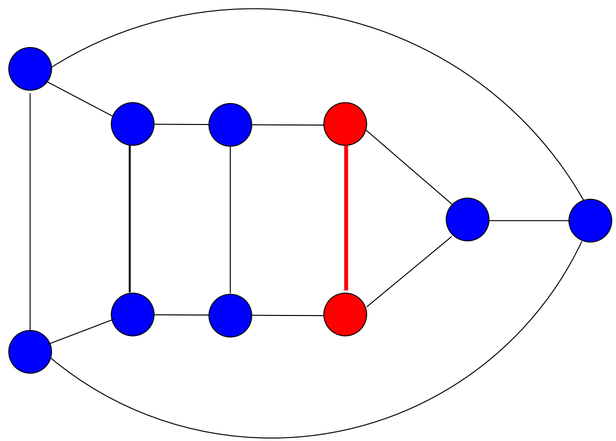
Cheap criterion to determine the *canonical* edge

E.g.: first choose the removable edges that have as few as possible vertices at distance at most 2.



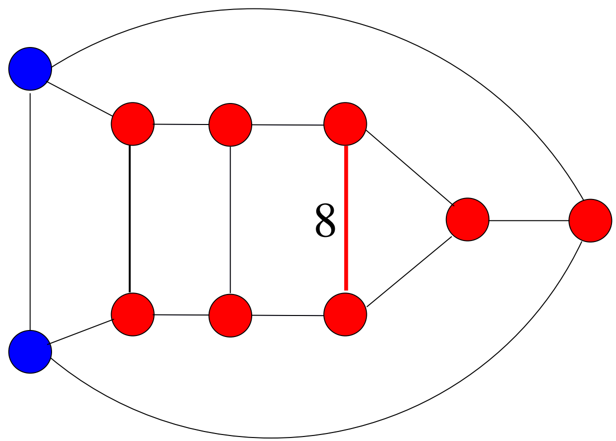
Cheap criterion to determine the *canonical* edge

E.g.: first choose the removable edges that have as few as possible vertices at distance at most 2.



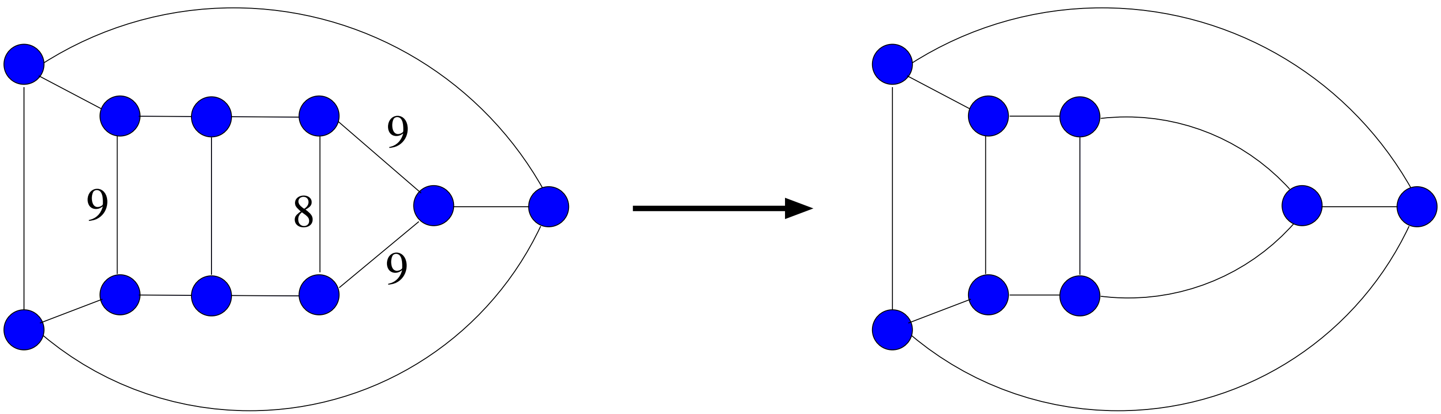
Cheap criterion to determine the *canonical* edge

E.g.: first choose the removable edges that have as few as possible vertices at distance at most 2.



Cheap criterion to determine the *canonical* edge

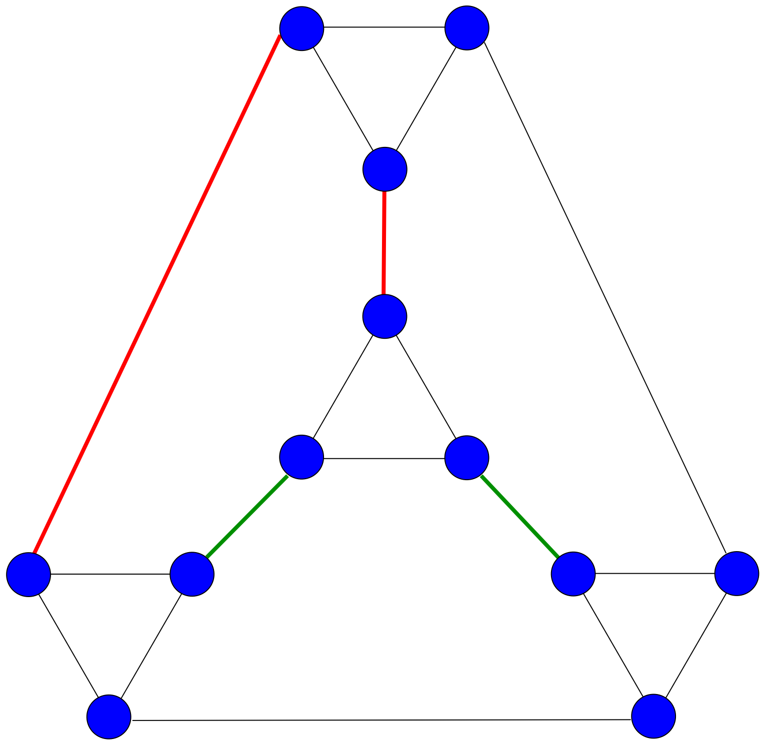
E.g.: first choose the removable edges that have as few as possible vertices at distance at most 2.



- Make sure that from the same graph you don't obtain the same ancestor twice *in the same way*:

Approximately: Compute the orbits of the automorphism group on the pairs of edges that can be chosen for extension.

The red pair and the green pair (and lots of others) give the same result.



So the rough pseudocode is

```
recursion(n) {
  // check canonicity of last operation
  // this may require to compute the group
  if (canonical) {
     if (n=wanted number of vertices)
         write up
     else {
         // compute possible extensions
         compute group // if not yet known
         compute equivalence classes (needs group)
         for each class choose extension i {
             // choose extension i
             extend(i)
             recursion(n+1)
         }
     }
   }
}
```

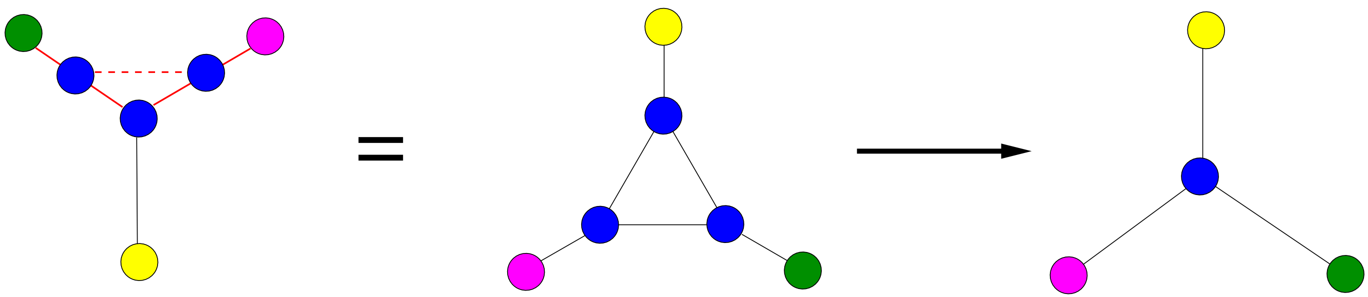
Just applying this (with some technical optimizations and a good implementation) already gives an algorithm that works quite well!

This is (more or less) the way the program of McKay and Sanjmyatav works.

Expensive parts are

- determining whether the last operation was *canonical*
- computing the symmetry group

Around 80% of the graphs have triangles –
if this part could be optimized. . .

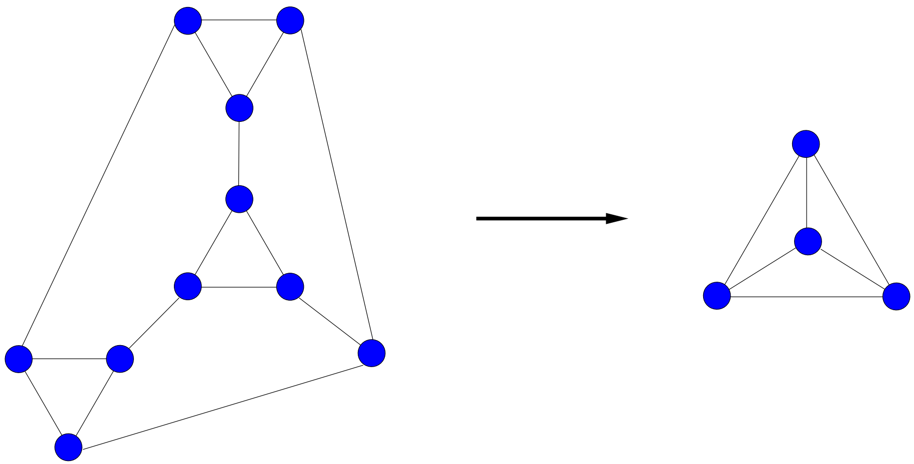


Simple new idea

Reduction: (determine ancestor)

Collapse **all** independent triangles (triangles that don't share an edge with other triangles) to a point – **alltogether in one operation**.

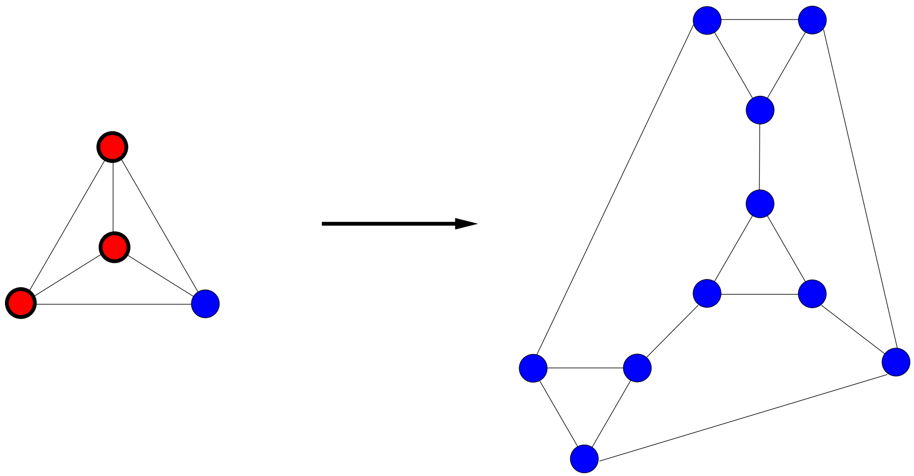
well . . . cheating a little bit. . .



The inverse operation:

Compute orbits of **sets** of vertices so that each triangle contains at least one. Blow these vertices up – no canonicity check.

And **no non-canonical graphs**!



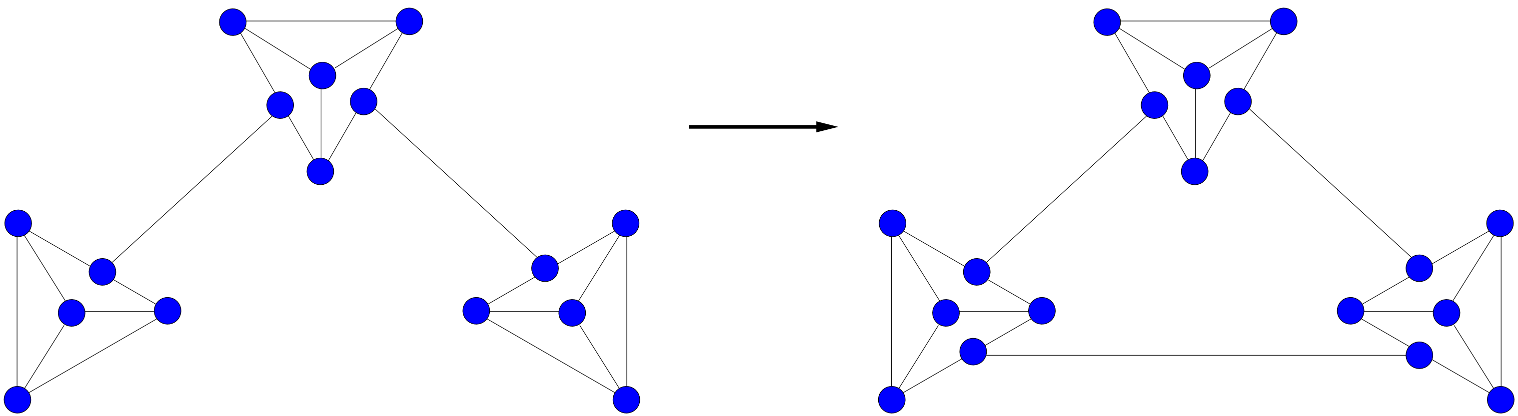
26 vertices

graphs with 0 triangles: 20.66%
graphs with 1 triangles: 32.45%
graphs with 2 triangles: 25.72%
graphs with 3 triangles: 13.59%
graphs with 4 triangles: 5.36%
graphs with 5 triangles: 1.66%
graphs with 6 triangles: 0.42%
graphs with 7 triangles: 0.08%
etc.

Expensive parts are

- determining whether the last operation was *canonical*
- computing the symmetry group

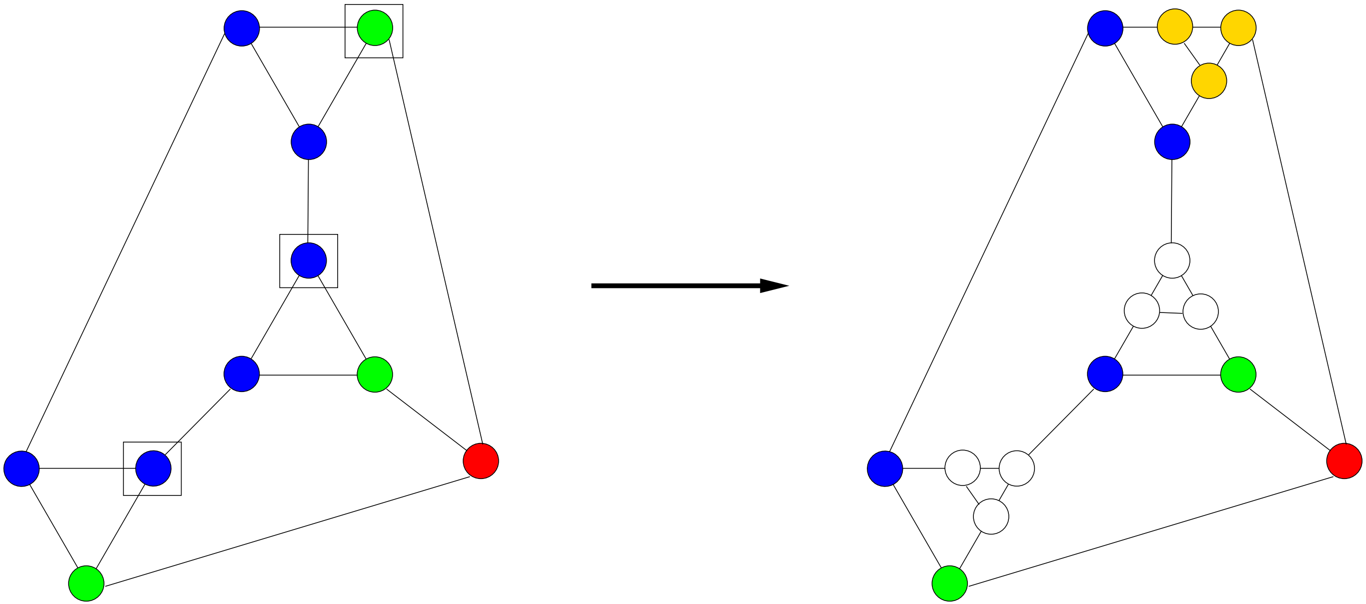
Adding an edge can change the group **dramatically** – the group can get larger and can get smaller.



So in case of adding an edge no information about the group can be reused.

Blowing up vertices to (all) triangles leads to subgroups (in a certain sense. . .).

A lot of information about the group can be reused to speed up the computation of the group – and in case of a trivial group we know that the group after the construction is trivial too!



Combined with look ahead for small cycles: graphs with girth 4 or 5 can be generated efficiently!

Ongoing work: similar principle of *simultaneaous blow up* for 4-gons.

Results: (Intel 64bit 2.33 GHz)

20 vertices: 80.000 graphs/sec 510.489 graphs

22 vertices: 88.000 graphs/sec 7.319.447 graphs

24 vertices: 93.000 graphs/sec 117.940.535 graphs

26 vertices: 95.000 graphs/sec 2.094.480.864 graphs

Speedup 3.76 to 3.3 compared to *minibaum*.

But well, . . . in principle minibaum was *fast enough*.

Everything you wanted to do with the graphs took longer than the generation. . .

So it is partly **record hunting**,

but the most important reason for the development of the new generator is the efficient generation of the subclass known as **Snarks**.

There are only very few Snarks among the cubic graphs

This construction method allows early detection of a lot of non-Snarks, so it is much faster than just applying a filter.

Thanks for your attention