Fast Generation of Cubic Graphs

RICHARD BEAN

concerning a paper by

Gunnar Brinkmann

and the algorithm therein

30 October 2002

History of Cubic Graph Generation

- 1889 – de Vries, up to 10 vertices enumerated
- 1967 – Balaban, 12 vertices enumerated
- 1974 – Petrenjuk + Petrenjuk, 12 vertices listed
- 1976 – Bussemaker et al, 14 vertices listed
- 1976 – Faradzev, 18 vertices ("orderly algorithm")
- 1986 – McKay + Royle, 20 vertices

"Orderly" algorithm

McKay (1998, "Isomorph-free exhaustive generation"): I. Algorithms

"... there is a canonical labelled object in each isomorphism class and that is the one generated."

Canon = "rule"

"The labelling is chosen to impose restrictions on subobjects, for example that there must be one maximal subobject which is also canonical."

* Pioneered independently by Faradžev and Read.

Improvements of Brinkmann

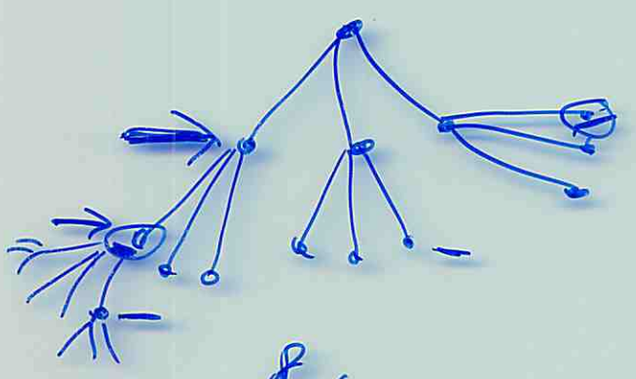
* the way a graph is to be checked in "maximal form"
* the use of a result about the relation between the girth of a regular graph and the maximal form
* avoidance of checks at points where they are usually inefficient.

Orderly algorithm?

* Generation of chess node count up to 10 ply – generate unique nodes first.

* Latin squares – maximize number of intercalates up to 12×12

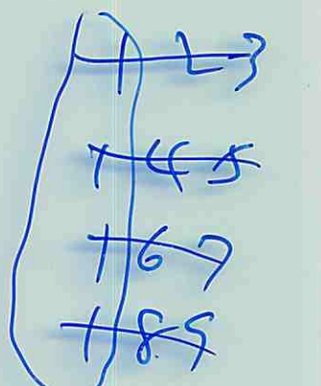
* Set-covering problem – recursively add entries but avoid subsequences we've seen before.

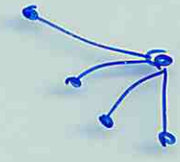


ply = half-move

$20^2 = 400$

8902 positions, 5362 unique positions





Graph representation

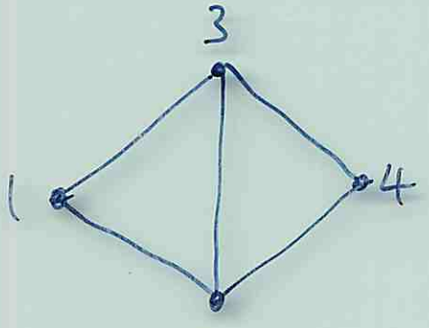
n = maximal label of a vertex
(vertex set = subset of $\mathbb{N}$)

representation is a list of length ℓ^2 for some $\ell \geq n$ (ℓ = basic number of the representation)

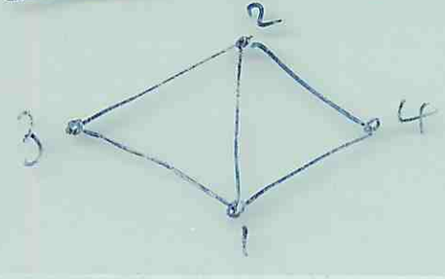
such that:
for $j \in \{1, \dots \ell\}$

$$(i-1)\ell + j = \begin{cases} 1 & \text{if } (i,j) \in G \\ 0 & \text{otherwise} \end{cases}$$

$G > G'$ if representation of G is lexicographically larger than G'.



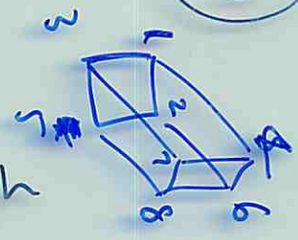
0110 1011 1101 0110



0111 1011 1100 1100

maximal representation

Maximal representation

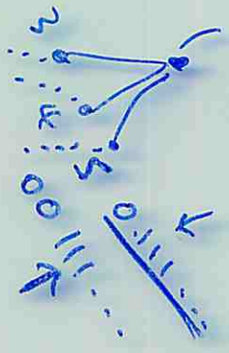


$max(G)$ denotes the unique graph isomorphic to G, such that for all G' isomorphic to G, $G' \leq max(G)$.

If $G = max(G)$ G is **maximal**.

Lemma 1. Let $G = (V, E)$, $G' \subseteq G$, $G' = (V', E')$ be graphs with the property that for every pair of edges (v,w) in $G \setminus G'$ and (v', w') in G' the edge (v', w') is lexicographically smaller than (v, w). Then G maximal $\rightarrow$ G' maximal.

Lemma 2. Let G be a maximal k-regular graph ($k > 1$). To construct a cycle of minimal length in G:

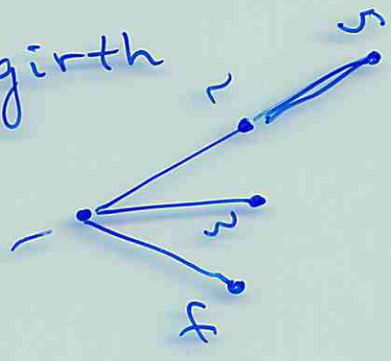


* Go from vertex $1 \rightarrow 2$.
* From vertex i, go to the smallest vertex adjacent to i except the one you just came from.
* Stop at vertex 1 - you came from vertex 3.

The Algorithm

Given
* number of vertices
* minimal girth

Begin with $B_{3,1}$:

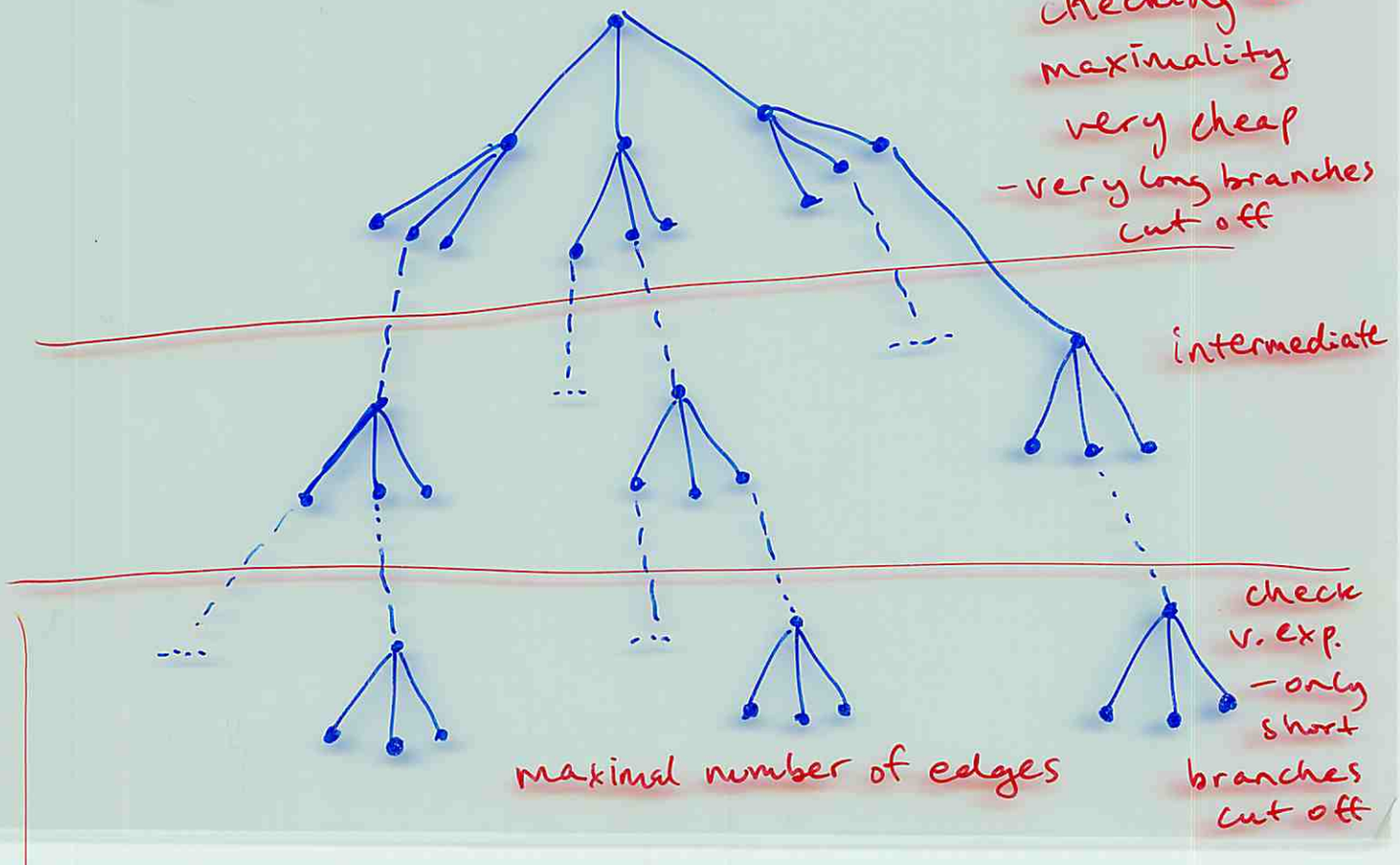


At each step, either:
* add an edge to smallest vertex with valence < 3

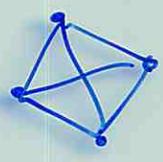
or * remove last edge filled in.

After edge addition, check whether graph still in maximal representation.

– (After construction of first cycle, don't add in smaller cycles.)



Checking for maximality



"Border value" the point at which intermediate checking is abandoned because it is too expensive – with 24 vertices, don't check if the starting vertex is greater than 17.

Testing the maximality of a graph G_i:

* First, assign the number 1 to every vertex of valence 3 in G_i.
* For the vertices adjacent to vertex 1, assign the vertex numbers 2, 3, 4 in each of the 6 possible ways.
  (starting sequence 0111... 10.... 100.... 1000...)
* Assign the vertices 5 or 5, 6 to the vertex which got number 2.
* Compare the starting sequence for G_i and the new numbering.

Checking for maximality (cont'd)

* If the starting sequence was larger, G was not maximal; there is no need to add more edges.

* If the starting sequences are the same; continue labelling with the vertex that was assigned the number 3.

The checking trees

If we have a graph G and a successor in the generation tree G' we can store the results of maximality testing for G where each vertex has valence 3.

The root of checking tree number k corresponds to assigning number 1 to vertex k. Then every root has 3! successors and every node that is not a root has 0, 1 or 2 successors.

1! 2!

The checking trees (cont'd)

The trees are enlarged or cut with every insertion or deletion of an edge (memory limits and the border value taken into account.)

Why is the checking tree techaique efficient?

* Bollobás / McKay + Wormald proved that asymptotically the ratio of cubic graphs with trivial automorphism group $\rightarrow 1$.

Active leaf: leaf of the checking tree which might lead to a smaller representation.

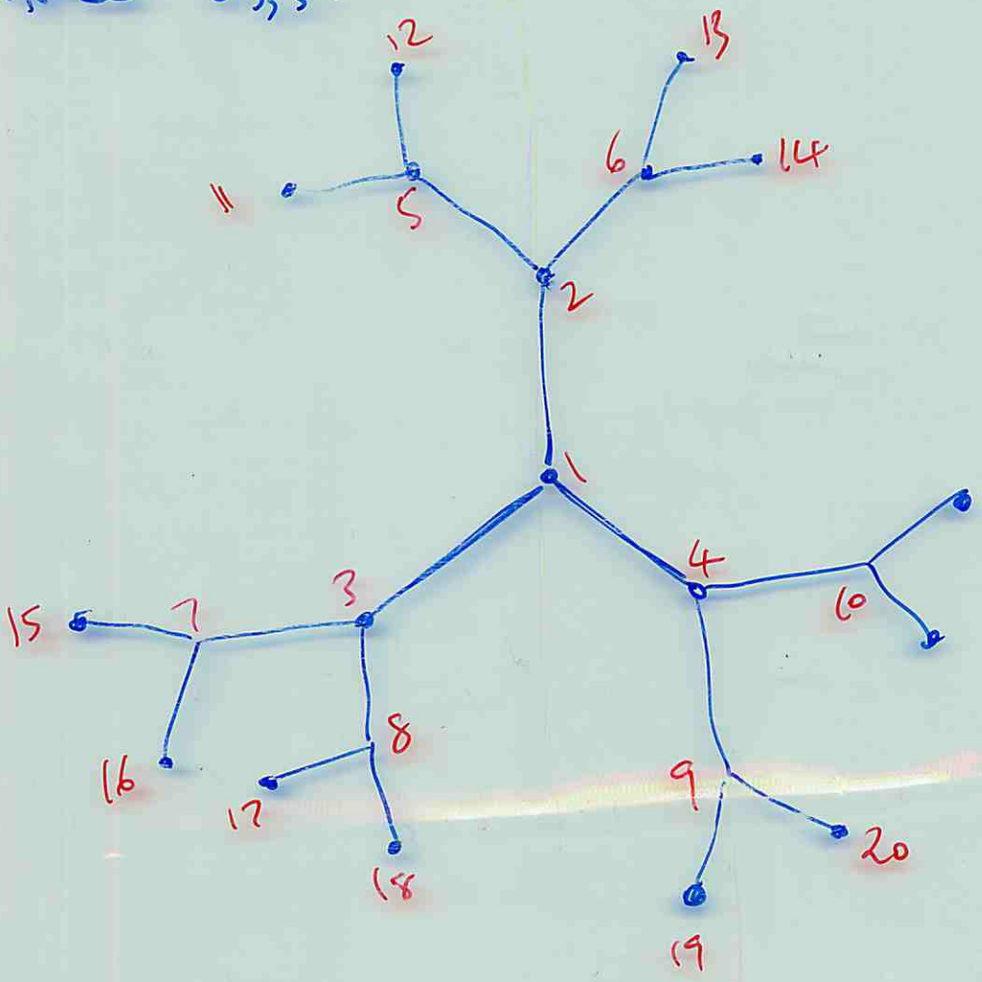
Dead leaf: leaf which corresponds to a numbering that already gives a larger beginning sequence than the representation we want to test.

...Efficient? (cont'd)

Whenever a leaf is marked dead, we backtrack to the node closest to the root where every successor is marked dead - and mark a dead branch.

This saves many potential assignments.

With 32 vertices, minimal girth 7: around every vertex we have the same tree $B_{3,3}$:



As soon as a vertex is marked as giving ~~minimal~~ a smaller representation, checking trees save a lot of time.

Running times

	1995, 1996?	2002	Minibaum
16	18 sec	< 1 sec	
→ 20	30 min	23 seconds	
→ 24	3 days 15h	? ~80 min	

Improvements?

* use more RAM
* use bitmaps
* ensure bitmaps match machine word size (32 or 64 bits)

32-bit word.

32/64