Counting and visualization of non-isomorphic k-element Gödel's algebras

Gabriele Maurina

Università degli studi di Milano

gabriele.maurina@studenti.unimi.it

January 24, 2019

Abstract

We produce software to quickly count and visualize non-isomorphic k-element Gödel's algebras. We use *Python3* for ease of writing and reading, and for its many powerful libraries.
This project is based on the work of Diego Valota in his paper *Spectra of Gödel Algebras*[4].

Overview

Spectra problems

Given a class of structures $\mathcal{C}$, according to[4], we can compute:

- *spectrum* of $\mathcal{C}$: $Spec(\mathcal{C}) = \{k | k = |C|, C \in \mathcal{C}\}$, that is the set of cardinalities of structures occurring in $\mathcal{C}$;

Spectra problems

Given a class of structures $\mathcal{C}$, according to[4], we can compute:

- *spectrum* of $\mathcal{C}$: $Spec(\mathcal{C}) = \{k | k = |C|, C \in \mathcal{C}\}$, that is the set of cardinalities of structures occurring in $\mathcal{C}$;

and, when $\mathcal{C}$ is a variety of algebras, we can also define for every natural number $k \geq 1$:

- *fine spectrum* of $\mathcal{C}$: $Fine_{\mathcal{C}}(K)$, that is the function counting non-isomorphic k-element structures in $\mathcal{C}$,

Spectra problems

Given a class of structures $\mathcal{C}$, according to[4], we can compute:

- *spectrum* of $\mathcal{C}$: $Spec(\mathcal{C}) = \{k | k = |C|, C \in \mathcal{C}\}$, that is the set of cardinalities of structures occurring in $\mathcal{C}$;

and, when $\mathcal{C}$ is a variety of algebras, we can also define for every natural number $k \geq 1$:

- *fine spectrum* of $\mathcal{C}$: $Fine_{\mathcal{C}}(K)$, that is the function counting non-isomorphic k-element structures in $\mathcal{C}$,
- *free spectrum* of $\mathcal{C}$: $Free_{\mathcal{C}}(K) = |\mathbf{F}_{\mathcal{C}}(K)|$, that is the function computing the sizes of the free k-generated algebra $|\mathbf{F}_{\mathcal{C}}(K)|$ in $\mathcal{C}$.

Gödel's Algebras

Gödel's logic:

it is a first-order logic, in which the truth values $\mathcal{V}$ are a closed subset of the interval [0, 1]. Different such sets $\mathcal{V}$ in general determine different Gödel logics. [5]

Godel's algebras:

they are the algebraic semantics of Gödel's logic.

With respect to the spectra problems, we use $\mathbb{G}$ to indicate the variety of class $\mathbb{C}$ of Gödel's algebras.

Gödel's Algebras spectra problems

Here are the formulas to solve the spectra problems for $\mathbb{G}$.

$$Spec(\mathbb{G}) = \mathbb{N}^+ \tag{1}$$

That is because every finite chain can be equipped with the structure of a Gödel algebra.

$$Free_{\mathbb{G}}(k) = (d(k))^2 + d(k) \tag{2}$$

with:
$d(0) = 1$
$d(n) = \prod_{i=0}^{n-1} (d(i) + 1)^{\binom{n}{i}}$
Such equation was first introduced by Horn[3] in 1969 and later redefined in dual terms by Dantona and Marra[2].

Gödel's algebras' fine spectrum

Valota in his paper[4] proposes this recurrence formula to compute the fine spectrum of $\mathbb{G}$.

$$Fine_{\mathbb{G}}(k) = f(k) + pr(k) * g(k) \tag{3}$$

with:

$f(2) = 1$

$f(k) = Fine_{\mathbb{G}}(K-1)$

$pr(k) = \begin{cases} 0 \ \mathit{if} \ \ k \ \mathit{is\ prime}, \\ 1 \ \mathit{otherwise}. \end{cases}$

$g(k) = \sum_{(n_1,\cdots,n_t)\in fact(k)} f(n_1) * \cdots * f(n_t)$

$fact(k) = \{(n_1, \cdots, n_t) | k = n_1 * \cdots * n_t, n_1 \leq \cdots \leq n_t, t > 1\}$

The set $fact(k) \cup (k)$ is known as the set of *unordered factorizations* aka *multiplicative partitions*.

Software

Our software is focused on the fine spectrum problem for $\mathbb{G}$. It needs to do two things:

- Counting: that is computing $Fine_{\mathbb{G}}(k)$. It must be fast, in order to push k as high as possible.
- Visualization: it needs to visualize the algebras as forests. It should be easy to use, and the output should be easily exportable.

Counting: naive implementation

Our first naive implementation follows step by step equation 3, introduced earlier. However such algorithm is exponential because in order to compute $Fine_{\mathbb{G}}(k)$ it is necessary to compute several other $Fine_{\mathbb{G}}(i)$ with $i < k$, and each one of them requires more. This grows exponentially with each step.

Counting: naive implementation

With this approach we were able to compute $Fine_{\mathbb{G}}(k)$ with $1 \leq k \leq 150$ in $\approx$ 14 seconds.

Counting: naive implementation

With this approach we were able to compute $Fine_{\mathbb{G}}(k)$ with $1 \leq k \leq 150$ in ≈ 14 seconds.

k	$Fine_{\mathbb{G}}(k)$	k	$Fine_{\mathbb{G}}(k)$	k	$Fine_{\mathbb{G}}(k)$	k	$Fine_{\mathbb{G}}(k)$	k	$Fine_{\mathbb{G}}(k)$
1	1	11	8	21	44	31	136	41	344
2	1	12	12	22	52	32	162	42	406
3	1	13	12	23	52	33	170	43	406
4	2	14	15	24	69	34	193	44	466
5	2	15	17	25	73	35	199	45	493
6	3	16	23	26	85	36	248	46	545
7	3	17	23	27	91	37	248	47	545
8	5	18	31	28	109	38	279	48	646
9	6	19	31	29	109	39	291	49	655
10	8	20	41	30	136	40	344	50	740

Table: $Fine_{\mathbb{G}}(k)$ with $1 \leq k \leq 50$

Observation

Since we call $Fine_{\mathbb{G}}(k)$, with the same k, incredibly many times throughout the execution, it is probably worth storing the result after the first call.

Python is your friend

Python3 offers a decorator in its standard library which can do just that. A decorator is a function that wraps around another function and adds a certain functionality to it.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    return (n <=2 and 1) or fib(n-1) + fib(n-2)

print(fib(100)) #354224848179261915075
```

Python is your friend

Using such an optimization we were able to reduce the time to compute $\mathit{Fine}_{\mathbb{G}}(k)$ with $1 \leq k \leq 150$ from 14 seconds to ≈ 0.01 seconds.

Python is your friend

Using such an optimization we were able to reduce the time to compute $\mathit{Fine}_{\mathbb{G}}(k)$ with $1 \leq k \leq 150$ from 14 seconds to ≈ 0.01 seconds. But that is not enough. We tried with $1 \leq k \leq 5000$ and it took ≈ 8 seconds.

Problem: slow unordered factorization

Analyzing the algorithm, it is clear that the bottleneck now is the routine which uses trial division to compute *fact*(k), see equation 3.

```
def find_factors(N, numbers=[], min=2, max=-1):
        if N == 1:
                yield numbers
        else:
                if max == -1:
                        max=N
                for i in range(min, max):
                        if N % i == 0:
                                numbers.append(i)
                                for res in\
find_factors(N // i, numbers, i, max):
                                        yield res
                                del numbers[-1]
```

Solution: fast prime factorization + multiset partition

Proposition: the set of multiset partitions of the prime factorization of n, is equal to the set of multiplicative partitions of n

$$multiplicative\_partitions(n) = multiset\_partitions(prime\_factorization(n))$$

Solution: fast prime factorization + multiset partition

Proposition: the set of multiset partitions of the prime factorization of n, is equal to the set of multiplicative partitions of n

$$multiplicative\_partitions(n) = multiset\_partitions(prime\_factorization(n))$$

Proof:

By the fundamental theorem of arithmetic[1] there exist a unique sequence of primes p_i so that $n = p_1 * p_2 * \cdots * p_m$, with $n > 1$.
Any partition of such sequence is a multiplicative partition of n, that is because of the multiplication associative property.
Any multiplicative partition of n, when factorized, gives the prime factorization of n. Let us assume that it doesn't. It must then produce a sequence of primes different than the original $n = p_1 * p_2 * \cdots * p_m$ but that would be impossible according to [1].
Hence the two sets are equal.

It’s time to bring out the big guns

SymPy is a Python library for symbolic mathematics. It offers 2 wonderful functions that can help us speed up the unordered factorization:

It's time to bring out the big guns

SymPy is a Python library for symbolic mathematics. It offers 2 wonderful functions that can help us speed up the unordered factorization:

factorint(n)

Returns the prime factorization of n. It uses a combination of different algorithms to achieve the prime factorization in the fastest time possible. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The *Pollard rho* and *p-1* algorithms are used to find large factors ahead of time. It also periodically checks if the remaining part is a prime number or a perfect power.

It's time to bring out the big guns

SymPy is a Python library for symbolic mathematics. It offers 2 wonderful functions that can help us speed up the unordered factorization:

factorint(n)

Returns the prime factorization of n. It uses a combination of different algorithms to achieve the prime factorization in the fastest time possible. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The *Pollard rho* and *p-1* algorithms are used to find large factors ahead of time. It also periodically checks if the remaining part is a prime number or a perfect power.

multiset_partitions(list)

Returns all the unique partitions of a given multiset.

Example

```
print(factorint(12, multiple=True))
#[2, 2, 3]
```

Example

```
print(factorint(12, multiple=True))
#[2, 2, 3]

print(list(multiset_partitions(factorint(12,\
multiple=True))))
#[[[2, 2, 3]], [[2, 2], [3]], [[2, 3], [2]],\
[[2], [2], [3]]]
```

Example

```
print(factorint(12, multiple=True))
#[2, 2, 3]

print(list(multiset_partitions(factorint(12,\
multiple=True))))
#[[[2, 2, 3]], [[2, 2], [3]], [[2, 3], [2]],\
[[2], [2], [3]]]

print([[reduce(mul, primes) for primes in partition]\
for partition in multiset_partitions(factorint(12,\
multiple=True))])
#[[12], [4, 3], [6, 2], [2, 2, 3]]
```

Results

With such a speed up we are now able to lower the time to compute $Fine_{\mathbb{G}}(k)$ with $1 \leq K \leq 5000$ from 7 seconds to ≈ 1.4 seconds.

Results

With such a speed up we are now able to lower the time to compute $Fine_{\mathbb{G}}(k)$ with $1 \leq K \leq 5000$ from 7 seconds to ≈ 1.4 seconds. We were also able to compute $Fine_{\mathbb{G}}(k)$ with K up to 1 million in ≤ 10 minutes.

$$Fine_{\mathbb{G}}(1M) \approx 3 * 10^{41}$$

Visualization

We want to visualize a set S_k of forests, representing the k-element non isomorphic Godel's algebras, such that $Fine_{\mathbb{G}}(k) = |S_k|$.
The algorithm to generate them is strictly related with equation 3. Infact

$$S_1 = 1$$

$$S_k = (1 \oplus S_{k-1}) \cup (S_{n_1} \otimes \cdots \otimes S_{n_i}), \forall (n_1, \cdots, n_i) \in fact(k)$$

Naive visualization

A forest is a group of trees. A tree is an undirected connected graph that contains no cycles. The quickest and easiest way to represent them as nested lists. Specifically any node in a tree is a list, and any sublist is a child of that node.

Naive visualization

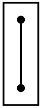
A forest is a group of trees. A tree is an undirected connected graph that contains no cycles. The quickest and easiest way to represent them as nested lists. Specifically any node in a tree is a list, and any sublist is a child of that node.

S_2 $\boxed{\bullet}$ [[]]

Naive visualization

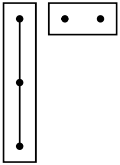
A forest is a group of trees. A tree is an undirected connected graph that contains no cycles. The quickest and easiest way to represent them as nested lists. Specifically any node in a tree is a list, and any sublist is a child of that node.

S_2 [[]]

S_3  [[[]]]

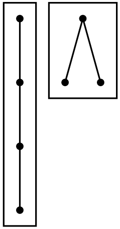
Naive visualization

S_4



[[[[]]]] [[] []]

S_5



[[[[[]]]]] [[[], []]]

Naive visualization

S_{12}

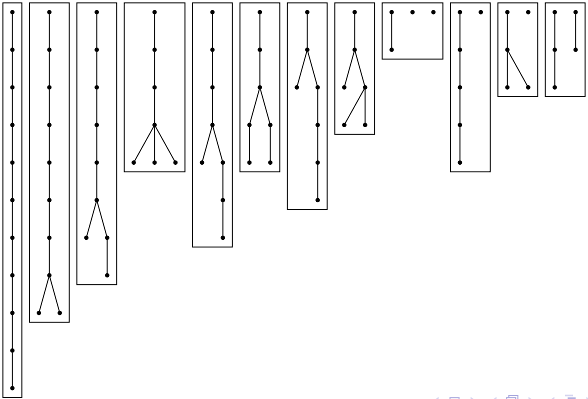
```
[ [ [ [ [ [ [ [ [ [ [ [ ] ] ] ] ] ] ] ] ] ] ] ]
[ [ [ [ [ [ [ [ [ [ ] ,  [ ] ] ] ] ] ] ] ] ] ]
[ [ [ [ [ [ [ [ ] ,  [ [ ] ] ] ] ] ] ] ] ]
[ [ [ [ [ [ ] ,  [ [ [ ] ] ] ] ] ] ] ]
[ [ [ [ [ [ ] ,  [ ] ,  [ ] ] ] ] ] ]
[ [ [ [ [ [ ] ] ,  [ [ ] ] ] ] ] ]
[ [ [ [ ] ,  [ [ [ [ ] ] ] ] ] ] ]
[ [ [ [ ] ,  [ [ ] ,  [ ] ] ] ] ]
[ [ [ [ ] ] ] ,  [ [ ] ] ]
[ [ [ [ [ [ ] ] ] ] ] ,  [ ] ]
[ [ [ [ ] ,  [ ] ] ] ,  [ ] ]
[ [ ] ,  [ ] ,  [ [ ] ] ]
```

Graphviz

Graphviz is an open source graph visualization software. It includes a python package that gives access its high level *APIs*. We used such a library to export the forests in a number of supported formats. Including eps, which is used for this presentation.

Graphviz

Graphviz is an open source graph visualization software. It includes a python package that gives access its high level *APIs*. We used such a library to export the forests in a number of supported formats. Including eps, which is used for this presentation. For example with k = 12 we have:



Final product

The final product is a Python3 script of around 120 lines of code (thanks, Python). It can be executed inside the console and offers different options for counting, including the SymPy optimization and for visualization, both naive, and trough Graphviz. It has a easy help command and a verbose option to monitor the progress of the exection. The commad to execute it is:

```
python3 godel-fine-forests.py [options] K
```

Options

The list of options is:

- -c count, count the non-isomorphic k-element Gödel's algebras up to K,
- -d <format> draw, draw the finite forest representing the non-isomorphic k-element Gödel's algebras, requires Graphviz installation,
- -h help, display help interface,
- -p print, print the naive visualization of the non-isomorphic k-element Gödel's algebras,
- -s sympy, activate the SymPy optimization, requires SymPy to be installation,
- -v verbose, display execution progress stats.

References

[1] H. Davenport.
The higher arithmetic: An introduction to the theory of numbers.
Cambridge University Press, New York, NY, USA, 8th edition, 2008.

[2] O.M. D'Antona and V. Marra.
Computing coproducts of finitely presented gödel algebras.
Annals of Pure and Applied Logic, 142(1-3):202–211, 2006.

[3] A. Horn.
Free l-algebras.
The Journal of Symbolic Logic, 34:475–480, 1969.

[4] D. Valota.
Spectra of gödel algebras.
Soon to be published, 2019.

[5] Wikipedia.
Gödel logic.