`Surge` - A Fast Open-Source Chemical Graph Generator

Brendan D. McKay[1*], Mehmet Aziz Yirik[2] and Christoph Steinbeck[2*]

[1] School of Computing, Australian National University, ACT 2601, Australia
[2]Institute of Inorganic and Analytical Chemistry, Friedrich-Schiller-University, Lessingstr. 8, 07743 Jena, Germany

Corresponding author email: brendan.mckay@anu.edu.au, christoph.steinbeck@uni-jena.de

Abstract

Chemical structure generators are used in cheminformatics to produce or enumerate virtual molecules based on a set of boundary conditions. The result can then be tested for properties of interest, such as adherence to measured data or for their suitability as drugs. The starting point can be a potentially fuzzy set of fragments or a molecular formula. In the latter case, the generator produces the set of constitutional isomers of the given input formula. Here we present the novel constitutional isomer generator `surge` based on the canonical generation path method. `Surge` uses the `nauty` package to compute automorphism groups of graphs. We outline the working principles of `surge` and present benchmarking results which show that `surge` is currently the fastest structure generator. `Surge` is available under a liberal open-source license.

Introduction

Chemical structure generators enumerate or generate molecular graphs of organic or bioorganic molecules. They are an integral part of systems for computer-assisted structure elucidation (CASE) [1] and can be used to create molecular libraries for virtual screening [2], [3] or enumerate chemical spaces in general [4]. The history of chemical graph generators goes back at least to the 1960s DENDRAL project which was aimed at the CASE of organic molecules based on mass spectrometric data [5]. DENDRAL was developed for NASA's Mariner program to search for life on Mars [5] [6]. Its structure generator used substructures as building blocks and was able to deal with overlapping substructures. In the early history of the structure generators, ASSEMBLE was another building block based structure generator [7]. In the field, there is a family of generators based on mathematical theorems such as algorithmic group theory [8] and combinatorics [9]. Besides DENDRAL, MASS [10] was also another good example for the applications of mathematical theorems in structure generation. It was a tool for

the mathematical analysis of molecular structures. SMOG [11] was the successor of the MASS algorithm.
Many works followed but few examples of practical usability are available even today [12]. Among the currently available structure generators, such as DENDRAL, ASSEMBLE, SMOG, COCON [13] and LSD [14], MOLGEN [15] constituted the state-of-the-art for decades in terms of speed, completeness and reliability. The first version of MOLGEN was based on the strategy of DENDRAL software and developed to overcome the limitations of DENDRAL [16]. The software is based on the orderly graph generation method [17]. Although MOLGEN is the *de facto* gold standard in the field, it has the downside of being closed-source software. The algorithm cannot be further developed or modified by scientists based on their interests. The most efficient and fast open-source chemical graph generator was MAYGEN [18] based on the orderly generation method. However, MAYGEN is approximately 3 times slower than MOLGEN.
The state of the art of large scale structure generation was recently set by the lab of Jean-Louis Reymond [19] in developing an in-house solution for a `nauty`-based structure generator, which enabled them to produce the numeration of 166 billion organic small molecules in the chemical universe database GDB-17. To the best of our knowledge, this in-house generator was not released as open-source or otherwise.
Thus, there is still the need for an efficient open-source chemical graph generator. In [18] we expressed the hope to “trigger a surge in the development of improved and faster” structure generators. Here we present the novel structure generator `surge`, based on the principle of the canonical generation path method. `Surge` is open-source and outperforms MOLGEN 5.0 by orders of magnitude in speed. Furthermore, `surge` is easily extensible with more features and adaptable to further application.

Methods

Data

We assembled a list of molecular formulae for benchmarking `surge` against MOLGEN 5.0 in Table 1-2. These formulae were taken from the natural products database COCONUT [20]. The size of these molecular formulae varies and is enough to challenge even the best constitutional isomer generators available (see results section).

Algorithm and mathematical background

`Surge` is based on the `nauty` [21] package for computing automorphism groups of graphs as well as canonical labels. Like `nauty`, `surge` is written in a portable subset of C and runs on a considerable number of different systems.
`Surge` is an integration of three existing tools from the `nauty` suite [22]: a) `geng` generates simple graphs based on certain boundary conditions, b) `vcolg` colors vertices in the output of `geng` and c) `multig` inserts multi-edges in the output of the first two tools (Figure 1).

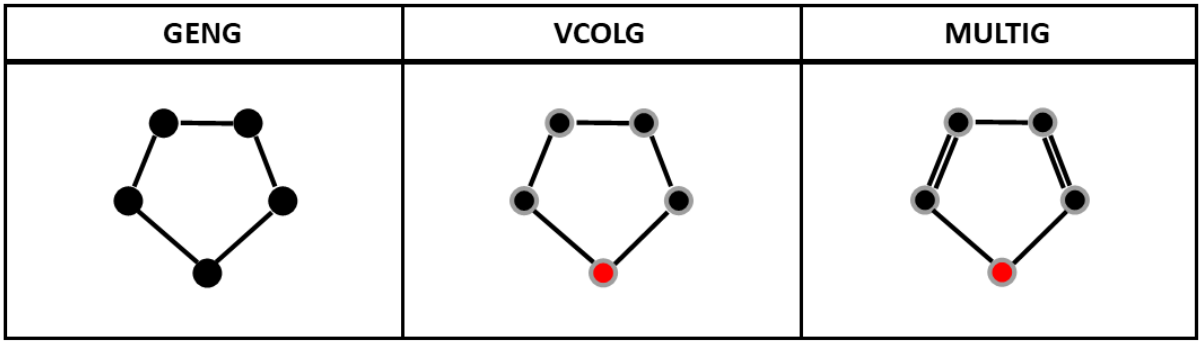

Figure 1: An example case for the combination of `geng`, `vcolg` *and* `multig` *functions for the furan molecule, C_4H_4O. First the simple graph is constructed. The nodes are coloured as, black for carbons and red for the oxygen. In* `multig`*, the edge multiplicities are optionally increased to create multiple bonds.*

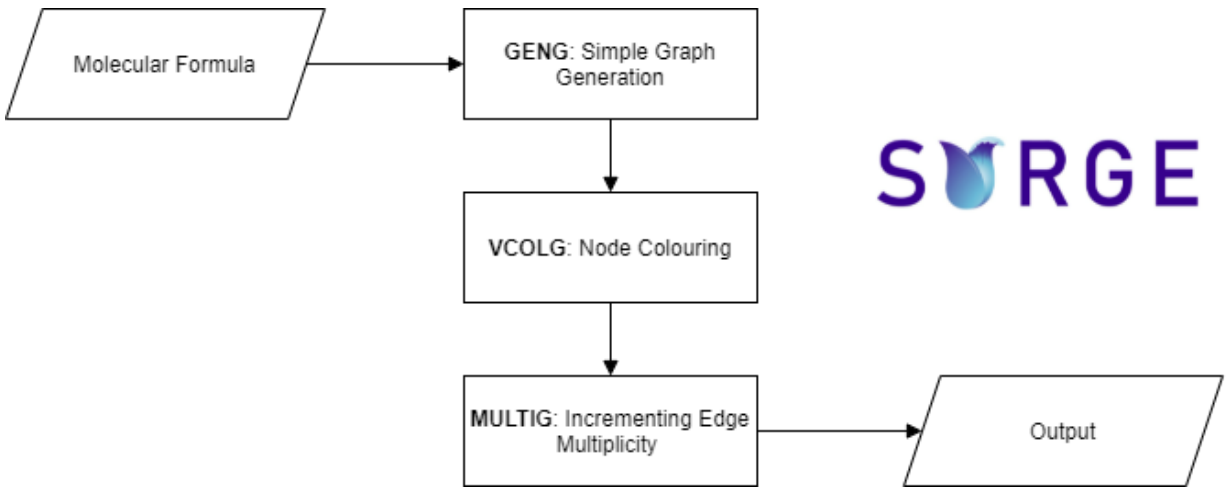

Figure 2: `Surge` *flowchart.*

An *isomorphism* between two graphs is a bijection between their vertex sets that maps edges onto edges. If the graphs have adornments, such as atom types for the vertices or bond multiplicities for the edges, then those adornments must be preserved by the mapping. If the two graphs are the same; i.e., the isomorphism is from a graph to itself, it is called an *automorphism*. The automorphisms form a group under the operation of function composition, called the automorphism group.

The meanings of isomorphism and automorphism are different for each of the three stages in our algorithm. Referring to Figure 1, at the first stage (which we call a simple graph) there are no vertex or edge adornments and all rotations and reflections, 10 in total, are automorphisms.

When vertex adornments are added in the second stage, the atom type becomes significant so only the identity mapping and the reflection through the oxygen atom are automorphisms. In the final stage, edge adornments are added but in this example the automorphism group is not further reduced since the reflection through the oxygen atom preserves both atom type and bond multiplicity. Note how the automorphism groups in the second and third stages are subgroups of the automorphism groups in the previous stages.

First stage

Input to `surge` consists of a molecular formula such as $C_7H_{12}O_2S$. Based on the element counts, in this case C=7, O=2, S=1, H=12, the atom valencies are used to calculate the plausible range of the number of edges of a connected simple graph representing the topology of a molecule with this formula, with hydrogen atoms omitted. Then `geng` is called to generate all the connected simple graphs with those parameters, subject also to a maximum degree condition depending on the molecular formula [23]. `Geng` generates one graph from each isomorphism class and these are passed to the second stage as they are produced, without any need to store them [23]. In this example, there are 10 non-hydrogen atoms and the number of edges is in the range 9-11.

Second stage

Given a simple graph G from the first stage, the second stage assigns elements to vertices in all distinct ways. The element counts must be correct, and we must have valence ≥ degree at each vertex. More onerously, we only want one member of each equivalence class of element assignment under the automorphism group of G. We next explain how this is accomplished.
The vertices of G are arbitrarily numbered 1,2,...,n. An element assignment can be represented as a list showing the element assigned to each vertex in order of vertex number. For example, a valid list might be L = (C,C,C,S,O,C,C,C,O,C).
Automorphisms of G have an action on lists that permutes their entries. Namely, for list L and automorphism γ, the list γ(L) assigns the same element to vertex γ(v) as L assigns to v, for each vertex v. Thus,

L = (C,C,O,S,O,C,C,C,C,C) and γ = (1 2 3)(5 6) imply γ(L) = (O,C,C,S,C,O,C,C,C,C).

If L is a list of elements and γ is an automorphism, L and γ(L) give equivalent assignment of elements to the vertices of G. Our task in this stage is to choose exactly one assignment from each equivalence class. Given a fixed ordering of the elements, for example C < O < S, two lists can be compared lexicographically, for example

(C,C,C,S,O,C,C,C,O,C) < (C,C,O,C,S,C,C,O,C,C)

This enables us to define

canon(L) = max { γ(L) | γ in Aut(G) },

the maximum list in the equivalence class of L. Note that canon(L)=canon(L') if L and L' are equivalent, so there is a unique maximum list L* in the equivalence class and we can recognize it by the condition canon(L*)=L*. To put it another way, if $\gamma(L) > L$ for any automorphism γ then $L \neq L^*$; otherwise $L = L^*$.
Now we describe the conceptual method for the second stage. For given G:

```
for each valid list L do
    for each automorphism γ of G do
        if γ(L) > L then
            reject L
        end if
        if L was not rejected then
            accept L
        end if
    end for
end for
```

This algorithm is efficient if the automorphism group Aut(G) is small, but that is not always the case. Therefore, we adopt a more complex approach. An automorphism of G is called *minor* if there are two leaves (vertices of degree 1) x,y with a common neighbour and the automorphism merely swaps x and y; i.e. (x y). The minor subgroup $M \leq Aut(G)$ is the subgroup generated by all the minor automorphisms.

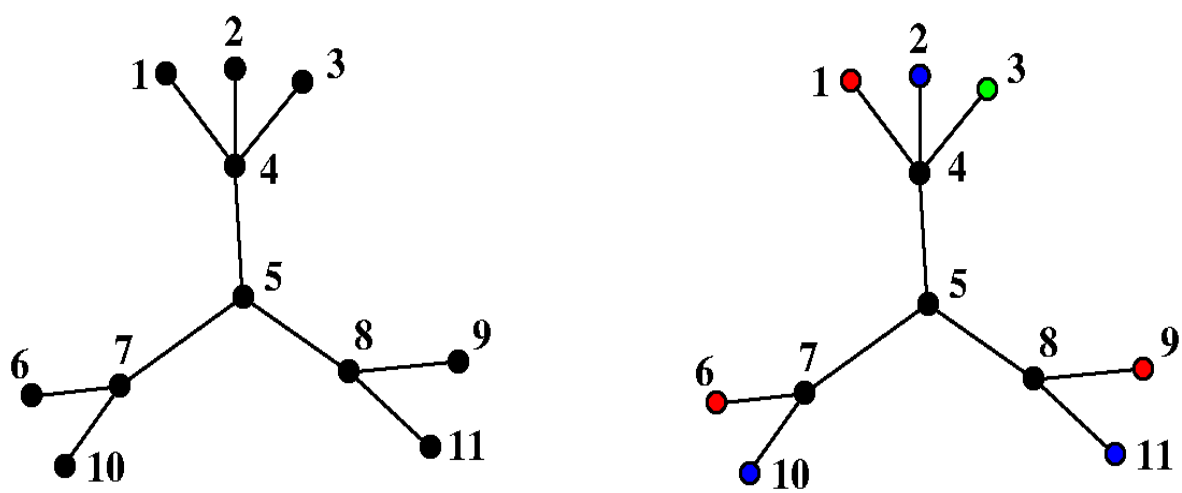

Figure 3. Two graphs with example flowers.

A *flower* is a maximal set of leaves with the same neighbour. In the left graph of Figure 3, the flowers are {1,2,3}, {6,10} and {9,11}. The minor subgroup M consists of all automorphisms that

preserve the flowers, such as (1 2 3)(9 11). The order of M is $3! \times 2! \times 2! = 24$. In addition to M, the automorphism group may contain automorphisms that do not preserve the flowers, such as (6 11)(7 8)(9 10). To capture such automorphisms, we colour the graph as in the right side of Figure 3. Vertices not in flowers are coloured black. Within each flower, vertices are coloured red, blue, green, … in order of vertex number, using a fixed list of colours that does not include black. Now let N be the group of automorphisms that respect the vertex colours. In the example, N has only the identity and (6 9)(7 8)(10 11).

An arbitrary automorphism of G can be obtained by first applying an element of N to capture how the flowers are mapped to each other, and then applying an element of M to capture the movement of leaves within each flower. In both steps the choice is unique, so we have a factorization

$$\mathrm{Aut}(G) = NM = \{ \gamma\delta \mid \gamma \text{ in } N, \delta \text{ in } M \}.$$

(In the language of group theory, M is a normal subgroup and N is a complete set of coset representatives.) In the example, consider (1 2)(6 11)(7 8)(9 10). It swaps the flowers {6,10} and {9,11} so we choose the element of N which does that, namely γ = (6 9)(7 8)(10 11). Then we have to arrange the leaves within the flowers with an element of M, namely δ =(1 2)(6 10)(9 11). This achieves $\gamma\delta$ = (1 2)(6 11)(7 8)(9 10).

The main advantage of factoring Aut(G) = NM is the following.

Theorem. For any list L, L = canon(L) if and only if L = max { δ(L) | δ in M } and L = max { γ(L) | γ in N }.

Proof. The "only if" direction is obvious since M and N are subsets of Aut(G). Suppose in the other direction that L = max { δ(L) | δ in M } and L = max { γ(L) | γ in N }. From the factorization of Aut(G) we know that L* = $\delta(\gamma(L))$ for some γ in N and δ in M. Note that in both L and L* the elements are in nonincreasing order within each flower, as they are maximized with respect to M. Also recall that the automorphisms in N preserve the order of vertex numbers within the flowers, by virtue of the fact that we coloured the vertices in order of vertex number when we computed N. This means that we can take δ to be identity, and so L* = γ(L). This proves that L* = L, since L = max { γ(L) | γ in N }.

In order to implement the condition L = max { γ(L) | γ in M }, we don't need to compute M explicitly. Instead, since M is generated by transpositions, it suffices that within each flower the elements are in decreasing order relative to vertex number. Using the ordering of elements that we have chosen, in the example we just need to enforce the inequalities element(1) ≥ element(2) ≥ element(3), element(6) ≥ element(10) and element(9) ≥ element(11). The program recursively assigns elements to vertices in order of vertex number and enforces these inequalities as they become active rather than at the end.

To implement the condition L = max { γ(L) | γ in N }, we compute N using `nauty` and test that γ(L) ≤ L for each γ in N. This is efficient in practice because N is very small most of the time.

We can also partly enforce N by means of inequalities: since vertex 6 is the least vertex in a non-trivial orbit {6, 9} of N, we can assume element(6) ≥ element(9). This is not necessary but it gives a small time improvement.

Third stage

After the assignment of elements to vertices is complete, the program moves to the next stage of selecting a bond multiplicity for each edge. This is the same type of problem as in the second stage. Instead of a list of elements for each vertex, we have a list of multiplicities for each edge. Instead of Aut(G), we use the subgroup of Aut(G) that preserves the element assignment. Otherwise M and N are defined as before. In the implementation, we don't use `nauty` to compute N but instead filter the N subgroup from the second stage, rejecting those automorphisms which don't preserve elements and converting the others to their action on the edges.

As an example, geng makes 534,493 unlabelled simple graphs in 1.3 seconds for Lysopine $C_9H_{18}N_2O_4$. For these graphs, the second stage subgroup N is trivial 58% of the time and never larger than 72. Assignment of elements to vertices produces 3,012,069,151 vertex-labelled graphs in 90 seconds.The N subgroup for the third stage is trivial 98% of the time and never larger than 24. Finally, the assignment of bond multiplicities produces 5,979,199,394 completed molecules in an additional 100 seconds.

As demonstrated by our examples, surge can generate molecular structures very quickly, allowing for the inspection of extremely large sets of isomers. The generation speed is several times faster than even the fastest output format (SMILES). On the other hand, any particular application will likely have stronger restrictions on the structure than just a molecular formula. For example, some substructures may make the molecule unstable or give it chemical properties undesirable in the application. Or, experimental investigation of an unknown compound may have determined some features of the structure, so that only molecules with those features are of interest.

For these reasons, surge provides a number of filters to limit the output. The 3-stage generation method allows some of them to be implemented almost for free, and all of them are much more efficient than filtering the output through an external program. For example, restrictions on the number of short rings and the planarity of the molecule can be enforced at Stage 1. Surge also provides some "badlists" of forbidden substructures (many of them inspired by the corresponding feature of MOLGEN).

The open-source nature of surge allows for a more advanced feature. By writing small code snippets, the user can insert custom filters into any of the three stages, and also perform such tasks as adding extra elements and command-line options. Several worked examples are provided with the program.

Results

`Surge` is available under a liberal open-source License (Apache 2.0) on GitHub at https://structuregenerator.github.io as well as from https://users.cecs.anu.edu.au/~bdm/surge/.

The system can be built with the standard Unix Configure/Make scheme and the resulting stand-alone executable is then run from the command line. By default, `surge` generates all constitutional isomers of a given molecular formula. `Surge` can write output in either SDfile [24]

or SMILES [25] format. SMILES output is produced very efficiently by constructing a template for each simple graph at the first stage, so that only atom types and bond multiplicity must be filled in before output.

We benchmarked `surge` with the set of molecular formulae given in Table 1. Since our motivation for developing structure generators is for the generation of large molecules, Table 1 consists of natural products, randomly selected from the natural products database COCONUT [20]. For the list of molecular formulae, `surge` outperformed MOLGEN by orders of magnitude (Figure 4) and MOLGEN terminated at a built-in limit of 2^{31}-1 structures. Reported computation times were linearly extrapolated based on the MOLGEN timing for 2^{31}-1 structures and the actual number of isomers reported by `surge`. Note that `surge` generates between 7 million and 22 million molecules per second for all of these examples.

`Surge` has a tiny memory footprint irrespective of the molecule size or the number of isomers. All of the examples in this paper run in at most 5 MB of RAM on Linux.

Table 1: Execution time (seconds) for selected MF of natural products on a compute-optimized c2-standard-4 Google cloud VM. Times for MOLGEN 5.0 were determined with the `-noaromaticity` flag to achieve comparability. Both MOLGEN and `surge` were instructed to generate but not to output structures.

Name of notable isomer	Molecular Formula	Species	#Isomers	SURGE time (s)	MOLGEN time (s)
Bassianolone	$C_{10}H_{16}O_5$	*Beauveria bassiana*	1092378303	69	5146
Pantothenate	$C_9H_{17}NO_5$	*Arabidopsis thaliana*	1652346465	165	11122
Lysopine	$C_9H_{18}N_2O_4$	*Parthenocissus tricuspidata*	5979199394	289	27250
Cribronic Acid	$C_6H_{11}NO_7S$	*Cribrochalina olemda*	2375932807	323	13445
Antibiotic CV-1	$C_7H_{14}N_2O_6$	*Streptomyces CO-1*	4193416397	448	24030
Thr-Thr	$C_8H_{16}N_2O_5$	*Trypanosoma brucei*	5955022220	575	37103
O-Succinyl-L-Homoserine	$C_8H_{13}NO_6$	*Escherichia coli K12*	5639328954	629	35128
Etrogol	$C_{13}H_{18}O_2$	*Stachylidium*	6316260274	746	44395
Indoleacetamide	$C_{10}H_{10}N_2O$	*Pseudomonas savastanoi*	13290477420	1187	59910

Colletotricole A	$C_9H_{13}NO_3S$	*Colletotrichum gloeosporioides A12*	20902484656	1765	88151
Nigerapyrone E	$C_{11}H_{12}O_4$	*Aspergillusniger MA-132*	31627481929	2179	181725
Siastatin B	$C_8H_{14}N_2O_5$	*Streptomyces verticillus var. quintum*	27692853176	2628	183167
P-Hydroxyhippuric Acid	$C_9H_9NO_4$	*Homo sapiens*	21964168804	2731	121362
Deacetyldemethyl anisomycin	$C_{11}H_{15}NO_3$	*Streptomyces sp. strain SA3097*	95541477841	4229	580772
Isoleucylisoleucyl Anhydride	$C_{12}H_{22}N_2O_2$	*Cordyceps bassiana*	59576199503	4782	516950
Hydantocidin	$C_7H_{10}N_2O_6$	*Streptomyces hygroscopicus*	40946033849	5238	262323
Aerugine	$C_{10}H_{11}NO_2S$	*Pseudomonas aeruginosa*	93330898027	8124	533440
Flavensomycinoic Acid	$C_9H_9NO_5$	*N/A*	113165341837	8870	793389
Dopamine 4-O-Sulfate	$C_8H_{11}NO_5S$	*Homo sapiens*	89694168554	9880	606333
Pestalactam C	$C_{10}H_{10}ClNO_3$	*pestalotiopsis sp.*	232824605597	14830	1700022
Glugaba	$C_9H_{16}N_2O_5$	*Escherichia coli*	176162377006	16265	1315301
Shihunine	$C_{12}H_{13}NO_2$	*Dendrobium loddigesii*	427207647324	19769	2504164
Gostatin	$C_8H_{10}N_2O_5$	*sumanensis*	187389585693	21781	1422863
Elaiomycin	$C_{13}H_{26}N_2O_3$	*N/A*	303023674167	29288	2729280
Oryzoxymycin	$C_{10}H_{13}NO_5$	*Streptomyces venezuelae var. oryzoxymyceticus*	552024644350	54372	6325646
Gammaglucys	$C_8H_{14}N_2O_5S$	*Mus musculus*	699785343381	69844	4989287
Phyllurine	$C_{10}H_{10}N_2O_3$	*Phyllanthus urinaria*	1511861775412	83186	8292585

Vanilloylglycine	$C_{10}H_{11}NO_5$	*Homo sapiens*	1182104108010	133136	21426660
Deoxyuridine	$C_9H_{12}N_2O_5$	*Phakellia mauritiana*	1795817811706	180727	13983652
Sulphostin	$C_5H_{13}N_4O_5PS$	*N/A*	2029911211739	226830	11893149

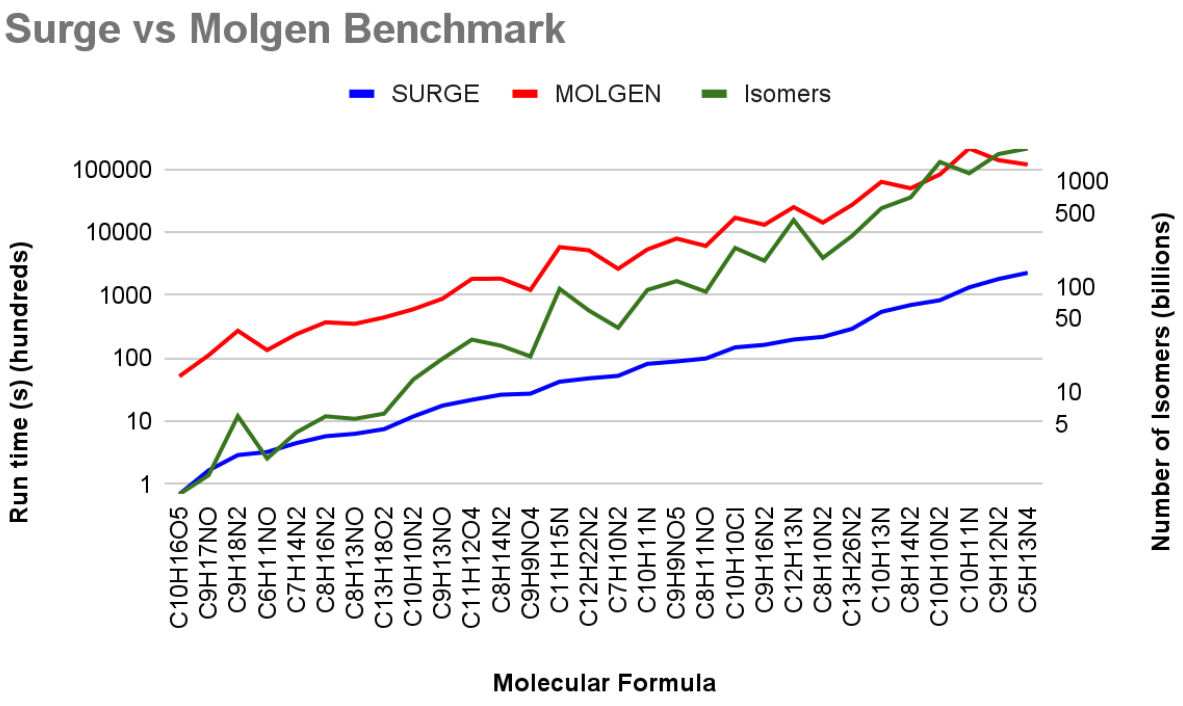

Figure 4: Comparison of the run times of `surge` *v1.0 vs MOLGEN 5.0 for long-running molecular formulae from selected natural products, plotted on a logarithmic time scale. In the majority of cases, MOLGEN terminated at a built-in limit of 2^{31}-1 structures. Reported computation times were linearly extrapolated based on the MOLGEN timing for 2^{31}-1 structures and the actual number of isomers reported by* `surge`*.*

For randomly selected 10 molecular formulae, 4 options of `surge` were tested and results are given in Table 2. These options are

-p0:1 At most one cycle of length 5
-P The molecule is planar
-B5 No atom has two double bonds and otherwise only hydrogen neighbours
-B9 No atom lies on more than one cycle of length 3 or 4

Table 2: Execution time (seconds) for selected MF of natural products on a compute-optimized c2-standard-4 Google cloud VM. `Surge` was run with its options and instructed to generate but not to output structures.

Molecular Formula	- p0:1		- P		- B5		- B9	
	#Iso	Time	#Iso	Time	#Iso	Time	#Iso	Time
$C_{11}H_{19}N_3O$	58175540999	3746	72486967073	5046	69648876936	4978	51275365737	3048
$C_{11}H_{18}N_2O_2$	53925725334	3648	67177819545	4914	64367528959	4838	47278714772	2946
$C_{11}H_{15}NO_3$	64661412269	4759	94361334994	7682	89131725467	7512	54627135057	3595
$C_9H_{18}N_2O_4$	5810409623	519	5979199394	541	5918503858	538	5583717596	484
$C_{11}H_{12}O_4$	17216498094	1894	30438650047	4485	28660902856	3777	14044693099	1256
$C_{10}H_{16}O_5$	989273530	107	1092378303	122	1060206152	122	895109814	88
$C_{13}H_{20}O_2$	1211481307	147	1514909702	203	1443691541	197	1038843543	101
$C_8H_{11}NO_6$	12795251232	1511	15771433061	1953	15035794185	1942	11169581507	1217
$C_9H_9NO_5$	62471125788	8244	109135601623	16008	102826808386	15645	51607646947	6062
$C_{12}H_{13}NO_2$	177274446997	13639	382246449331	34476	381333513411	34285	147423365942	9700

Limitations

Release 1.0 of `surge` does not perform a Hückel aromaticity test and therefore generates duplicate structures for Kekulé versions of aromatic rings that are graph-theoretically different. Benchmarking against MOLGEN 5.0 was therefore performed with the -noaromaticity switch of MOLGEN.

Conclusion

We have presented `surge`, a structure generator for constitutional isomers based on the canonical generation path method. To the best of our knowledge, `surge` is the fastest chemical structure generator available. A number of badlist options are available to avoid the generation of potentially unlikely structures. Current limitations include the lack of an aromaticity detection. `Surge` is hosted as an open-source package on GitHub, inviting the scientific community to use and extend it. `Surge` offers a plug-in mechanism for community-driven extensions. Plugins can hook into the various stages of the `surge` generation process, thereby offering efficient means to prune the generation tree.

Availability and Requirements

- Project name: `surge`
- Project home page: https://structuregenerator.github.io
- Operating system(s): Platform independent
- Programming language: C
- License: Apache 2.0

Competing Interests

All authors declare no competing interests.

Funding

MAY and CS acknowledge funding by the Carl-Zeiss-Foundation.

Acknowledgements

Author contributions

BDM wrote the code and developed the underlying `nauty` package. BDM, CS and MAY conceived the project. BDM and CS guided the development. MAY contributed to the conceptual development and performed the evaluation and testing.
All authors wrote, read and approved the manuscript.

Author information

Affiliations

School of Computing, Australian National University, ACT 2601, Australia
Brendan D. McKay

Institute for Inorganic and Analytical Chemistry, Friedrich-Schiller-University, Jena, Germany
Mehmet Aziz Yirik & Christoph Steinbeck

References

1. Elyashberg M, Argyropoulos D. Computer Assisted Structure Elucidation (CASE): Current and future perspectives. Magn Reson Chem [Internet]. Wiley; 2020; Available from: https://onlinelibrary.wiley.com/doi/10.1002/mrc.5115

2. Miyao T, Kaneko H, Funatsu K. Ring system-based chemical graph generation for de novo molecular design. J Comput Aided Mol Des. 2016;30:425–46.

3. Saldívar-González FI, Huerta-García CS, Medina-Franco JL. Chemoinformatics-based enumeration of chemical libraries: a tutorial. J Cheminform. 2020;12:64.

4. Blum LC, Reymond J-L. 970 Million Druglike Small Molecules for Virtual Screening in the Chemical Universe Database GDB-13. J Am Chem Soc. American Chemical Society; 2009;131:8732–3.

5. Lindsay RK, Buchanan BG, Feigenbaum EA, Lederberg J. DENDRAL: A case study of the first expert system for scientific hypothesis formation. Artif Intell. 1993;61:209–61.

6. Gulyaeva KA, Artemieva IL. The ontological approach in organic chemistry intelligent system development. Advances in Intelligent Systems and Computing. Singapore: Springer Singapore; 2020. p. 69–78.

7. Badertscher M, Korytko A, Schulz KP, Madison M, Munk ME, Portmann P, et al. Assemble 2.0: a structure generator. Chemometrics Intellig Lab Syst. 2000;51:73–9.

8. Holt DF, Eick B, O'Brien EA. Handbook of Computational Group Theory. CRC Press; 2005.

9. Kreher DL, Stinson DR. Combinatorial algorithms: generation, enumeration, and search. CRC press; 2020.

10. Serov VV, Elyashberg ME, Gribov LA. Mathematical synthesis and analysis of molecular structures. J Mol Struct. 1976;31:381–97.

11. Molchanova MS, Shcherbukhin VV, Zefirov NS. Computer Generation of Molecular Structures by the SMOG Program. J Chem Inf Comput Sci. 1996;36:888–99.

12. Yirik MA, Steinbeck C. Chemical graph generators. PLoS Comput Biol. 2021;17:e1008504.

13. Junker J. Theoretical NMR correlations based Structure Discussion. J Cheminform. 2011;3:27.

14. Nuzillard J-M, Georges M. Logic for structure determination. Tetrahedron. 1991;47:3655–64.

15. Gugisch R, Kerber A, Kohnert A, Laue R, Meringer M, Rücker C, et al. MOLGEN 5.0, a Molecular Structure Generator in Advances in Mathematical Chemistry. Advances in Mathematical Chemistry; Basak, SC, Restrepo, G , Villaveces, JL, Eds.

16. Grund R, Kerber A, Laue R. Construction of discrete structures, especially isomers. Discrete Appl Math. 1996;67:115–26.

17. Grüner T, Laue R, Meringer M. Algorithms for group actions: homomorphism principle and orderly generation applied to graphs. DIMACS Ser Discrete Math Theoret Comput Sci. 1997;28:113–22.

18. Yirik MA, Sorokina M, Steinbeck C. MAYGEN: an open-source chemical structure generator for constitutional isomers based on the orderly generation principle [Internet]. Journal of Cheminformatics. 2021. Available from: http://dx.doi.org/10.1186/s13321-021-00529-9

19. Ruddigkeit L, van Deursen R, Blum LC, Reymond J-L. Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17. J Chem Inf Model. 2012;52:2864–75.

20. Sorokina M, Merseburger P, Rajan K, Yirik MA, Steinbeck C. COCONUT online: Collection of Open Natural Products database. J Cheminform. 2021;13:2.

21. McKay BD, Piperno A. Practical graph isomorphism, II. J Symbolic Comput. 2014;60:94–112.

22. McKay B, Piperno A. nauty and Traces User's Guide [Internet]. 2019 Sep. Available from: https://pallini.di.uniroma1.it/Guide.html

23. McKay BD. Isomorph-Free Exhaustive Generation. J Algorithms. 1998;26:306–24.

24. CTFILE FORMATS BIOVIA DATABASES 2016 [Internet]. 2016. Available from: https://help.accelrysonline.com/ulm/onelab/1.0/content/ulm_pdfs/direct/reference/ctfileformats2016.pdf

25. Weininger D. SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules. J Chem Inf Comput Sci. ACS AMERICAN CHEMICAL SOCIETY; 1988;28:31–6.