Batch-Advatage Transformer with Hypergraph Optimized Grammar (BAT/HOG)

Anonymous Authors[1]

Abstract

We present a novel approach to the issue of molecular optimization. Our approach uses a hypergraph replacement grammar inferred from the ZINC database, with grammar construction optimized for molecular structure creation. We treat the optimization as a reinforcement learning problem, using a batch-advantage modification of the policy gradient algorithm - using individual rewards minus the batch average reward to weight the log probability loss.

The reinforcement learning agent is tasked with building molecules using this grammar, with the goal of maximizing benchmark scores available from the literature. To do so, the agent has policies both to choose the next node in the graph to expand and to select the next grammar rule to apply. The policies are implemented using the Transformer architecture with the partially expanded graph as the input.

We achieve state of the art performance on common benchmarks from the literature, such as penalized logP and QED, with only hundreds of steps (without pre-training) on a budget GPU instance. Competitive performance is obtained on more advanced GuacaMol v2 goal-oriented benchmarks. Coupled with a Transformer based discriminator, the model achieves competitive results on the GuacaMol distribution benchmarks; training is stable over a range of hyperparameter values.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

2. Generating guaranteed valid SMILES strings

2.1. Context-free grammar

Non-binary trees are a natural description of molecular structure, be it using the SMILES approach (cutting the cycles), or the approach of (**?**) (treating cycles as graph nodes to be resolved into individual atoms at a later stage).

A context-free grammar (CFG) is, in turn, a natural way of describing non-binary trees. It allows us to efficiently represent molecules as non-binary trees with some constraints on tree structure. That efficiency matters for model training: RNN-style models have problems passing information over very long distances. The Transformer architecture we use doesn't have that problem, but its RAM requirements grow approximately as a square of the number of generation steps (in our case, rules needed to generate a given molecule). Finally, for any architecture, length of the sequence used to represent a molecule directly affects training time.

Thus the average number of tokens needed to represent a molecule constrains the maximum size of molecules it's practical to produce.

Using a CFG allows us to combine a granular representation (yielding a combinatorial variety of structures), with explicit representation of particularly frequent or complex combinations (eg double aromatic cycles).

To generate guaranteed valid molecules, we introduce a new grammar that respects atomic valences and explicitly introduces cycles.

2.2. Respecting valences

The initial rules are

```
smiles -> nonH_bond
smiles -> initial_valence_1 bond
smiles -> initial_valence_2 double_bond
smiles -> initial_valence_3 triple_bond
```

Any token ending in `bond` represents an open bond to

which some atom must be attached; `nonH_bond` requires that that atom not be hydrogen (see Section **??** on treatment of hydrogen). The molecule can then be grown via rules such as

```
nonH_bond -> valence_2 bond
double_bond -> '=' valence_2
double_bond -> '=' valence_3 bond
double_bond -> '=' valence_4 double_bond
triple_bond -> '#' valence_3
triple_bond -> '#' valence_4 bond
```

So far it seems as if we could only produce linear molecules. However, the `valence` tokens can expand to individual atoms as well as to structures that contain branches, for example

```
valence_4 -> 'C'
valence_3 -> 'N'
valence_3 -> valence_4 branch
valence_2 -> 'O'
valence_2 -> valence_3 branch
valence_2 -> valence_4 '(' double_bond ')'
```

This allows us, so far, to generate arbitrary tree-shaped molecules, ie those without cycles.

2.2.1. Implicit hydrogens

A complication in the above approach is that the SMILES format allows for implicit hydrogens, ie any valence of any atom that's not specified is assumed to be occupied by hydrogen. Thus in effect, these hydrogen atoms are represented by empty strings, which is a problem for us as a CFG does not allow to expand a token into nothing (otherwise parsing would be much harder).

If the implicit hydrogen atoms were represented by for example lowercase `h`, we could have grammar rules such as

```
bond -> 'h'
bond -> nonH_bond
branch -> 'h'
branch -> '(' nonH_bond ')'
```

If all we cared about was generating new SMILES strings, we could have used the above 4 rules and then inserted a post-processing step that replaces each `h` with an empty string. However, we also want to be able to parse existing SMILES strings into our grammar, so we go with a two-step approach: we start with a grammar as described above, and then eliminate all occurrences of `bond` by replacing them with either nothing or `nonH_bond`, and similarly for `branch`.

Thus for example, the rule

```
valence_3 -> valence_4 branch
```

is replaced by the two rules

```
valence_3 -> valence_4
valence_3 -> valence_4 '(' nonH_bond ')'
```

2.3. Cycles

The different kinds of cycles are introduced by rules such as

```
nonH_bond -> aliphatic_ring
aliphatic_ring -> valence_3_num cycle_bond
aliphatic_ring -> valence_4_num
                          cycle_double_bond
valence_2 -> vertex_attached_ring
vertex_attached_ring -> valence_4_num
                        '(' cycle_bond ')'
```

The challenging bit here is the numbering - the two ends of a cycle must be marked with the same numeral, to know we should connect them; furthermore, SMILES reuses these numerals, so if the numeral `1` was used in a cycle that's already closed, we can use it again for the next cycle we start. Thus cycle numeral management is a task that's quite hard to solve using just a context-free grammar.

We solve this by attaching additional properties to some nonterminal tokens. When we expand a nonterminal which has `ring` as a substring, indicating a new cycle, we give it a unique cycle identifier, which is attached as a property to any tokens resulting in that expansion, which have substrings `num` or `cycle`. When these are in turn expanded, the cycle id is again propagated. The grammar rules for cycle propagation are formulated in a way that guarantees there are ever at most two tokens tagged by any one cycle ID, in fact exactly two until we expand them into numerals.

2.3.1. Numeral assignment

When a nonterminal is about to be expanded into a numeral, we know by construction that it has a cycle id. We have two cases to consider: firstly, there exists another token with this cycle id. In that case, we scan the token sequence so far to determine the lowest available numeral, and generate a mask that makes sure that specific numeral will be chosen; and store the pair (cycle id, numeral) in a cache. In the second case, when there are no tokens with this cycle id, we look up the numeral to use in the cache, by cycle id.

2.3.2. Cycle size

A cycle with only two atoms (same numeral attached to atoms that are already connected by a regular bond) is considered illegal in SMILES. To prevent these, we attach and propagate another property to nonterminals that are part of a cycle, namely ring size, incrementing it with each rule expansion. Cycle size masking then forbids the rules that would have led to premature cycle closure. As cycles of length more than 8 don't occur in the database (and our scoring function penalizes cycles with length more than 6), we also forbid cycle expansion beyond length 8.

2.3.3. CYCLE CHAINING

Aliphatic cycles can form arbitrarily long, potentially branching chains, due to a pair of rules that allow starting a new cycle while constructing another:

```
cycle_bond -> aliphatic_ring_segment
                                cycle_bond
aliphatic_ring_segment -> valence_3
        '(' cycle_bond ')' valence_3_num
```

In the first of these rules, the `cycle_bond` token on the right hand side will get the same cycle id as that on the left hand side; while the `aliphatic_ring_segment` token will trigger generation of a new cycle id, that will be propagated to the appropriate nonterminals on the right hand side of the second rule, in this case `cycle_bond` and `valence_3_num`.

This is just an example, a similar pattern is used to attach an aliphatic cycle to aromatic cycles described below.

2.3.4. AROMATIC CYCLES

Aromatic cycles and chains thereof must fulfil additional relationships between the number of atoms and the number of double bonds in each cycle. Solving this in a general way is beyond the scope of current work, we limit ourselves to rules that allow us to generate any single aromatic cycle of length 5 or 6, and common patterns of linked pairs of aromatic cycles, possibly with chains of aliphatic cycles attached. For example, here are some of the rules used in generating a coupled pair of aromatic rings of size 6 and 5:

```
double_aromatic_ring -> 'c' num1
  aromatic_atom aromatic_atom aromatic_atom
  'c' num 'n' num1 aromatic_atom
  aromatic_atom aromatic_atom_num
aromatic_atom -> 'n'
aromatic_atom -> 'c'
aromatic_atom -> 'c' '(' nonH_bond ')'
aromatic_os -> 'o'
aromatic_os -> 's'
aromatic_atom_num -> 'c' num
```

2.4. Making sure the expansions terminate

The final challenge we address is making sure the rule expansion terminates before the maximum allowed number of expansion rules. To do this, we define the concept of *terminal distance* of a token, defined as the length of the shortest sequence of rules needed to transform that token into a sequence consisting only of terminals.

We calculate that distance for all tokens reachable from the root token by means of the following algorithm:

1. Define a set T of all tokens observed so far, and seed it with the root token `smiles`.
2. Iterate over the elements t of T.
   (a) For each t we firstly apply all applicable rules and add any new tokens generated thereby to T, seeding their terminal distance with 0 for terminals and ∞ for nonterminals.
   (b) We then calculate the terminal distance of a token t as one plus the minimum over all applicable rules of the sum of terminal distances of the tokens on the right hand side of the respective rule:

$$TD(t) = 1 + \min_{r \in G: r.lhs = t} \left(\sum_{t' \in r.rhs} TD(t') \right) \quad (1)$$

We repeat step 2. until convergence, that is, until no new tokens are observed and the terminal distance for known tokens no longer changes after an iteration.

We then define the change in terminal distance made by a production rule as

$$\Delta TD(r) = \sum_{t' \in r.rhs} TD(t') - TD(r.lhs) \quad (2)$$

By definition of the terminal distance, for every nonterminal t there exists a production rule r such that $r.lhs = t$ and $\Delta TD(r) = -1$.

Finally, we define the terminal distance of a token sequence as the sum of terminal distances of the individual tokens.

We use the terminal distance concept to make sure the rule expansion terminates before the maximum number of steps, in the following manner: at each step, we consider the number s of steps left and the terminal distance td of the sequence generated so far.

At each step, we make sure $td \leq s$, using induction. First, we choose the maximum rule sequence length to be larger than the terminal distance of the root symbol. Second, at each rule selection step we consider all rules r whose left hand side is the next nonterminal to expand, and only allow those where $\Delta TD(r) + td \leq s - 1$. That is a nonempty set because $\Delta TD(r') = -1$ for at least one applicable rule r'. Thus, by the time we run out of steps, that is, $s = 0$, we know $td = 0$, that is our token sequence consists only of nonterminals.

Note that in our case, we have to apply this algorithm not to the tokens of our original grammar, but to the extended tokens (including cycle size information); and likewise use the production rules consisting of the original rules plus the propagation of cycle size information[1]. This is because cycle masking forbids some rules (eg those that would create a cycle of two atoms) that would be allowed by our original grammar. Effectively, the rules for propagating additional

[1]This is an additional limit maximal cycle size, namely to make sure the number of extended tokens is finite

information, along with the original CFG rules, induce a CFG on the space of extended tokens, and it is the terminal distance within that CFG that we must consider.

Also for computational efficiency reasons (to not multiply the number of tokens unnecessarily) we disregard the cycle id propety of the tokens when calculating terminal distance, as it only affects choice of numeral but not terminal distance.

2.5. Limits of context-free grammars

Of the above modifications to a pure context-free grammar, cycle size masking could have been implemented by including the size value into the token string, and making a copy of each rule whose left hand side is that token, for every size value that occurs. This would increase the number of rules by a couple dozen, but stay within the limits of a CFG.

On the other hand, numeral choice and terminal masking do not appear possible to achieve using a CFG alone. However, these merely restrict the list of possible production rules at any step - but the resulting SMILES string is still valid according to the original CFG without these extra attributes. This is imporant because it allows us to use our original CFG to parse known molecules, eg from the ZINC database, to train our model on.

2.6. Extraction of Hypergraph Cliques

2.7. Rule-pair Encoding

2.8. Grammar conciseness and expressiveness

Our approach leads to a more concise way of representing molecules using production rules, with an average 2.85 production rules per atom and 62.8 rules per molecule in the ZINC dataset, compared to average 5.46 rules per atom and 120.8 rules per molecule in the grammar used by (**?**). This in turn allowed us to generate larger molecules (about 100 atoms for a typical molecule that uses the maximum number of steps, and over 400 atoms in special cases) on commodity hardware.

Our grammar can represent arbitrarily long, branching chains of aliphatic cycles, as well as single aromatic cycles with five or six atoms, pairs of aromatic cycles, and arbitrarily long branching chains of aliphatic cycles attached to aromatic ones.

Because our grammar is more restrictive than a fully generic SMILES grammar (to make it easier for us to generate guaranteed valid molecules), it can't represent all the molecules in the ZINC database, but rather a little over a third of the molecules, 92K out of 250K. Our grammar is also expressive enough to represent the recent state-of-the art constructed molecules, such as those in (**?**) and (**?**).

This approach can be extended to cover the whole ZINC database by introducing additional production rules - certainly by brute force, by adding to the grammar every unknown pattern that the parser encounters; and most likely also in a more elegant way. Extending our grammar to represent most or all molecules from that dataset is subject of ongoing work.

There is something of a tradeoff between expressiveness and guaranteed correctness - as one adds more features to the grammar, it becomes harder to maintain the guaranteed-correctness property. Fortunately, that property is not actually necessary for our approach (best-of-batch policy gradient) to work - it's sufficient for a large fraction, say 75%, of each batch, to be valid for the optimization to succeed.

Future attempts to expand the grammar's expressiveness might make use of that by relaxing the guaranteed-correctness property if needed.

3. Model choice

3.1. Reinforcement learning

The literature ((**????**)) on molecule generation has focussed on training an autoencoder for the chosen molecule representation, using a database of known molecules such as ZINC, then doing Bayesian optimization over the latent space.

Instead, we consider molecule creation as a reinforcement learning problem. At each step the action taken by the network is choosing the next production rule; the state is the full sequence of the production rules so far; and the reward is nonzero for the step following molecule completion (ie when the sequence defined by applying the chosen production rules contains no more nonterminals), and zero for all other steps. Details of the reward specification are in Sections **??** and **??**.

The first step of the autoencoder/Bayesian optimization approach, training the autoencoder on a database of known molecules, typically served two objectives: firstly, ensuring that the model learns to produce chemically valid molecules (for the majority of models where that is not guaranteed by construction); secondly, ensuring that the molecules the model produced are 'similar' to the molecules the model has been trained on.

While this approach has useful traits, it also has disadvantages: firstly, once the autoencoder is trained, its notion of 'similarity to known good molecules' is cast in stone, and no subsequent optimization over the latent space would allow it to produce, for example, models with similar structure but substantially larger than the ones it was trained

on. In contrast, casting molecule generation as a reinforcement learning problem allows us to treat 'similarity to known good molecules' as the gradual concept that it is, for example (as we do) by including a variably-weighted loss term penalizing the coefficient distance from a version of the model trained off-policy on known-good molecules. This allosw us to search over a greater space of possible molecules, while maintaining control over the degree of similarity to the known-good ones.

Secondly, learning a latent space representation of the space of all the molecules in the database is a reasonably hard task, which is not necessary if all we want is to generate molecules that maximize some metric and are similar to existing ones. We conjecture that simply teaching a model to have a high probability of producing those molecules is a computationally cheaper task, which can be used to achieve the same aim via anchoring (see below).

Finally, given the extremely non-Euclidean structure of the space of all valid molecules, it seems promising to conduct the search for optimal molecules directly on a representation that seems close to it, in our case the sequence of grammar production rules tailored to molecular structure, rather than on the Euclidean latent space.

3.2. Architecture

As the molecular properties we seek to optimize are essentially nonlocal, and because a grammar-rule representation of a branching tree means that even atoms that are nearby on the tree can end up generated by rules that are far apart in the rules sequence, we use the Transformer architecture (**?**), chosen because any two items in the sequence are directly linked by its attention mechanism. Our implementation of the Transformer decoder outputs one vector of logits per call (for each molecule in the batch), to be used in choosing the next production rule for each sequence in the batch; and omits the calculation steps that use the sequence generated by the encoder as it doesn't exist in our caase.

We use 6 layers, 6 heads, $d_k = d_v = 16$, $d_{model} = 128$, $d_{inner\_hid} = 256$.

Prior to calculating the log-softmax to be used for sampling the next grammar rule, we calculate a mask as described in Section **??** and subtract 1e6 from the logits at the indices forbidden by the mask.

3.3. Training

We use same maximal sequence length as (**?**), 277. That allows us a maximum batch size of 40 on an AWS p2.xlarge instance.

In the first stage, we train the model off-policy on 92K molecules from the ZINC database (**?**) that can be represented by our grammar. In order to reward the model for producing molecules similar to the ones it observes, we use simple policy gradient loss (**??**). Here s goes over the model steps, and $\pi_s(r_s)$ is the model-produced probability (after applying all masking) of choosing rule number r_s at step s.

$$loss = -\sum_{s=0}^{S} \log\left(\pi_s\left(r_s\right)\right) \tag{3}$$

We optimize in batches of 40 molecules, using the average loss for the batch, and Adam optimizer with a learning rate of 1e-4. After each batch, we do an on-policy simulation of a batch of 40 molecules with the same loss function, to judge convergence - but without updating the model coefficients.

To judge convergence, we compare the distribution of log probability, logP, SA score, and the number of aromatic cycles in the molecules produced by the model to those in the database, and find that they have converged after 15 epochs of training (30K batches, about one week hours on a p2.xlarge). We take the model thus trained as our base model.

We then proceed to on-policy training using best-of-batch policy gradient with base model anchoring. That is, our loss is the log-likelihood *for the best-scoring molecule in the batch* plus the L2-distance between the coefficients of the current model and the base model, multiplied by a weight w_a(**??**). The purpose of the second term ('anchoring') is to prevent catastrophic forgetting, and to control the degree of similarity of optimized model to the base model (and thus to the molecules in the database).

$$loss = -\sum_{s=0}^{S} \log\left(\pi_s\left(r_s\right)\right) + w_a \left|\vec{p} - \vec{p}_{base}\right|_2^2 \tag{4}$$

The reward function is used to choose the best molecule of the batch, but its value for the best molecule does not affect the loss function, which remains a simple log likelihood for the best molecule.

We found that using the best molecule in the batch, rather than the batch average, during the second phase was crucial for successful optimization - without it, as soon as the search found a somewhat successful molecule, the optimization tended to converge to always producing that molecule; while taking best-of-batch enabled continued exploration.

3.4. Optimization and the reward function

We optimize the metric used in prior literature ((**?**), (**?**), (**?**)), namely logP minus the synthethic accessibility score, penalized for cycles of more than 6 atoms. Each of those

three terms is evaluated for each of the molecules in the ZINC dataset using code shared by (**?**), and their mean and standard deviation are calculated. These are used to normalize each of the 3 score constituents, before adding them up to calculate a molecule's score.

The overall reward function is thus represented by (**??**).

$$R = logP_{norm} + SA_{norm} + C_{norm} \tag{5}$$

For reasons explained in Section **??**, we use an extended version of (**??**), namely (**??**).

$$\begin{aligned} R \quad = \quad & logP_{norm} + SA_{norm} + C_{norm} \\ & + w_{SA} \min\left(SA_{norm}, 0\right) \\ & - w_{ac} \min\left(num\_aromatic\_cycles - 5, 0\right) \end{aligned} \tag{6}$$

Here the first three terms on the right are the standard scoring function used in the literature, and the final two terms allow us to penalize low SA scores and too large a number of aromatic cycles in a molecule.

4. Results

Depending on how harshly we penalize the distance to the base model (choice of w_a), our model produces a range of molecules with scores substantially exceeding state-of-the-art, with varying sizes, as shown in Table **??** and discussed in detail below.

Table 1. Molecule statistics (higher score is better)

Source	Score	Norm. SA	Ar. rings
Average over ZINC	0.0	0.0	1.85
Stdev. over ZINC	2.07	1.0	0.97
Kusner et al., 2017	2.93	1.35	1
Jin et al., 2018	5.30	0.87	5
Kajino 2018	5.56	1.22	4
Strong Anchor 1st	**5.68**	1.31	5
Strong Anchor 2nd	5.60	0.52	4
Strong Anchor 3rd	5.33	0.21	5
Weak Anchor B, 1st	9.48	0.27	7
Weak Anchor B, 2nd	8.73	0.08	5
Weak Anchor A	12.25	0.08	12
Unconstrained	**46.45**	2.46	13

When trying to optimize (**??**) without constraning similarity to known molecules ($w_a = 0$) and using the original scoring function from the literature (that is, $w_{SA} = 0$ and $w_{ac} = 0$), we've found that the optimization tried to get as many aromatic cycles as possible, leading to a large logP but very low SA score, and then attach a large number of halogen atoms to these to bring the SA score back up (see Figure **??**, right). While this allowed for an order of magnitude increase in the molecule score ('Unconstrained' in Table **??**), such molecules don't appear to be realistic candidates for having useful properties.

We next added weak anchoring ($w_a = 1e8$) and a penalty on negative normalized SA score ($w_{SA} = 20$). The result is shown in Table **??** as 'Weak Anchor A' and in Fig **??**, left. We see that the introduced constraints temper but don't entirely remove the tendency of the optimization to blindly maximize the score by adding a lot of aromatic cycles.

To counteract that, we add a penalty on the number of aromatic cycles exceeding 5 ($w_{ac} = 5$). The three top-scoring molecules are shown in Table **??** as 'Weak Anchor A', and in Figure **??**. We see that even after constraining the number of aromatic cycles, we can achieve scores well in excess of 8, and a larger variety of atoms and structures than in the previous cases. However, we still see many more halogens than in a typical molecule from the ZINC database.

For a final set of simulations, we increase the anchoring weight to $w_a = 2e8$, keep the SA penalty as before, and remove the aromatic cycle penalty. The three top-scoring molecules are shown in Table **??** as 'Strong Anchor, SA penalty', and in Figure **??**. We see that these molecules appear much more similar to the ZINC database - in addition to the aromatic cycles of 6 carbon atoms we see an aliphatic cycle, aromatic 5-cycles containing sulphur and nitrogen, and just one chlorine atom. All of these molecules' scores exceed those of the top molecule of (**?**).

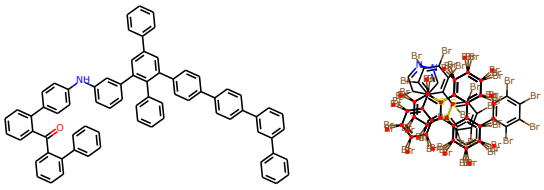

Figure 1. Top molecule with weak anchoring and SA penalty (left) and no anchoring (right)

5. Conclusion

A major contribution of (**?**) was regarding the molecule as a junction tree, with any cycles represented as nodes on par with non-cycle atoms. This made it possible to create a model guaranteed to generate valid molecules. However, their implementation was comparatively complex and also required all possible cycles to be enumerated in a dictionary upfront.

Here we have shown that the same idea can be implemented

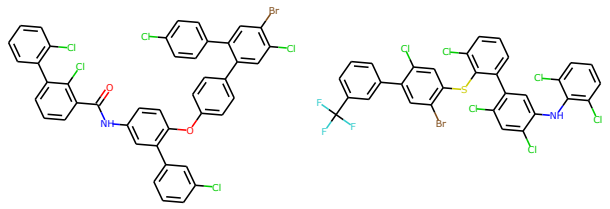

Figure 2. Top 2 molecules with weak anchoring, and SA and aromatic cycle penalty

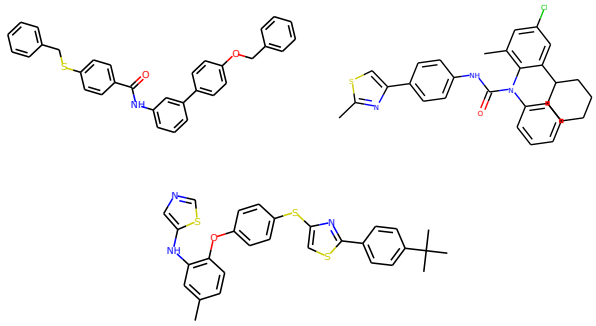

Figure 3. Top 3 molecules with strong anchoring and SA penalty

in a simpler fashion, using a custom context-free grammar, combined with some additional masking for choosing the next production rule.

A properly constructed context-free grammar gives us the best of both worlds: the ability, with a small number of production rules, to represent all possible valid combinations of atoms to build a given structure, as in (**?**); and at the same time, the ability to explicitly specify complex structures (eg linked aromatic rings) upfront using a single rule per structure, similarly to (**?**), while letting further rule expansions supply the detail. All that is done as part of a single grammar, with the decomposition of a molecule into a production rule sequence being done using a standard CFG parser, and the generation of new molecules being done by any model able to recursively produce a sequence of tokens (we chose the Transformer because of its nonlocal information propagation, but an RNN stack could have been used as well).

To solve some of the shortcomings of a purely CFG-based approach, we propagated additional information with the tokens during rule expansion, and used it to limit the set of allowed production rules at each step.

This approach, combining a custom CFG with additional rule masking that uses non-local information, is applicable beyond molecules, to any domain where we need to optimize complex graph structures under a mixture of local and nonlocal constraints: the local constraints can be taken care of in the CFG design, and the nonlocal ones enforced via additional masking.

Finally, we show that molecule optimization can be naturally cast as a reinforcement learning problem, and the state-of-the-art results we get even with the very basic policy gradient method suggest further scope for improvement using more advanced approaches, for example methods based on tree search.