POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Titre de mon document / Title

DAVID SAIKALI
Département de Génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Juillet 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Titre de mon document / Title

présenté par **David SAIKALI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Amal ZOUAQ, présidente
Gilles PESANT, membre et directeur de recherche
Louis-Martin ROUSSEAU, membre

DEDICATION

À tous mes amis du labos,
vous me manquerez...

ACKNOWLEDGEMENTS

Texte / Text.

RÉSUMÉ

La recherche de nouvelles molécules est un processus sans fin.

Le résumé est un bref exposé du sujet traité, des objectifs visés, des hypothèses émises, des méthodes expérimentales utilisées et de l'analyse des résultats obtenus. On y présente également les principales conclusions de la recherche ainsi que ses applications éventuelles. En général, un résumé ne dépasse pas trois pages.

Le résumé doit donner une idée exacte du contenu du mémoire ou de la thèse. Ce ne peut pas être une simple énumération des parties du manuscrit. Le but est de présenter de façon précise et concise la nature, l'envergure de la recherche, les sujets traités, les questions de recherche ou les hypothèses soulevées, les méthodes utilisées, les principaux résultats ainsi que les conclusions retenues. Un résumé ne doit jamais comporter de références ou de figures.

ABSTRACT

This thesis goes over our efforts to represent drug-like molecules using Constraint Programming.

Do this: Ajoute une phrase de motivation, soit avant soit après.

We also attempt to evaluate desirable properties to guide our results towards potentially useful molecules. To try and improve the realism of generated molecules, we combine Machine Learning, specifically Natural Language Processing, and Constraint Programming. We use perplexity and the success rate to evaluate our model's quality. This allows us to weigh the different constraints against the information learned by the model. If our work shows promise, we believe it could be useful in other domains to apply long-term structure in long sequence generation.

Written in English, the abstract is a brief summary similar to the previous section (Résumé). However, this section is not a word for word translation of the abstract in French.

The abstract is a brief statement of the subject matter, objectives, research questions or hypotheses, experimental methods and analysis of results. It also presents the main research conclusions and their possible applications. In general, an abstract should not exceed three pages.

The abstract should provide an exact idea of the thesis or dissertation's contents and it cannot be a simple enumeration of the manuscript's parts. The goal is to precisely and concisely present the nature and scope of the research. An abstract should never include references or figures. If the thesis or the dissertation is in English, the résumé (French-language abstract) should come first followed by the abstract.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

CHAPTER 1 INTRODUCTION

Drug discovery is a very time-consuming and costly endeavor due to its enormous design space — estimated to contain between 10^{23} and 10^{60} different molecules [1] — and to the lengthy and failure-fraught process of bringing a product to market. Automated molecule design is nowadays a vital part of drug discovery and material science, with computational approaches coming from deep generative models and combinatorial search methods [2]. It aims to extract from this huge design space the most likely candidates according to some desired properties. Even among these, only a few may lead to a usable product after extensive testing.

SMILES, a one-dimensional encoding of molecules, is one of the standards commonly used by this research community. It lends itself well to techniques used for Natural Language Processing (NLP), such as sequential generative neural models. LLMs in particular have come to the forefront of popular attention as impressive tools for generating text. However, this generation isn't limited to purely human languages as we can train the model on another text format, such as SMILES, and get a model capable of generating molecules.

SMILES also lends itself very well to CP. CP seems like a natural approach to molecule discovery since it allows hard constraints to be placed which could ensure only valid molecules are generated. Using a Context-Free Grammar (CFG) and a few additional constraints could allow us to describe valid SMILES strings in a CP model. We also believe it may be possible to model desirable molecular properties using CP, this would allow our model to restrict its search even further. This allows us to explore the huge design space of possible molecules while adding constraints in order to restrict that space to suitable candidates.

This CP approach, which excels at imposing hard rules and long-term structure while lacking the informed decision making that trained models gain from the dataset used, could allow us to answer one of the issues with sequence models in Machine Learning (ML). Often times, these models struggle to exhibit long-term structure, stemming in part from the token-by-token nature of the prediction process used to generate a sequence. In other words, these models do not explicitly learn the hard rules that determine validity nor desirability and merely mimic what was observed.

While this problem can be addressed at training, by changing what the model is trained on such that it can better learn the structure, we wish to introduce a Constraint Programming with Belief Propagation (CPBP) model at inference time (generation). This should enforce the presence of the desired structure, which is critical if it is mandatory [3,4]. In other words, this technique could allow us to target properties and structures that the model was never trained to generate while still maintaining advantages of the original model.

This guarantees that a generated molecule will respect the desired structure, which is critical if it is mandatory [3, 4].

This combined model is of more interest, however, when we wish to impose constraints that were not featured in the training dataset of the model. This allows the satisfaction of these new constraints while avoiding the retraining of the model, which can be costly with larger models.

This combination of both techniques could lead to valid, realistic and property-constrained molecules. However, there is a balance to maintain as we do not wish to stray too far from what was featured in the training dataset in order to respect the imposed constraints. This is particularly difficult for long-term structure, which requires balancing foresight over many yet-to-be generated tokens and the immediacy of next-token predictions from the sequence model.

1.1 Context-Free Grammar

A Context-Free Grammar is a set of rewrite rules used to generate strings. Formally, grammar $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ is defined, respectively, by a set of nonterminal symbols $\mathcal{N}$, a set of terminal symbols (its alphabet) Σ, a set of production rules $\mathcal{R}$, and a start symbol S. We denote $L(\mathcal{G})$ the language recognized by $\mathcal{G}$ *i.e.* the set of strings that grammar can generate. According to Chomsky's classification, there are many types of grammars, ranging from least to most restrictive: Recursively Enumerable (Type-0), Context-Sensitive (Type-1), Context-Free (Type-2) and Regular (Type-3). For a grammar to qualify as context-free, its production rules must respect two restrictions: the left-hand side of the production must be a single nonterminal, and the right-hand side must be a string of terminals and nonterminals.

The classic example of a CFG is one where we match opening and closing parentheses. This becomes necessary later to ensure the validity of the generated molecules.

As an example of a CFG, take the grammar defined as follows:

$$\mathcal{N} = \{S, A, B, C\}$$
$$\Sigma = \{\langle, \rangle\}$$
$$\mathcal{R} = \{ \text{①}\ S \to SS,\ \text{②}\ S \to AC,\ \text{③}\ S \to BC,\ \text{④}\ B \to AS,\ \text{⑤}\ A \to \langle,\ \text{⑥}\ C \to \rangle\}$$
$$S = S$$

This context-free grammar recognizes correctly bracketed words such as "$\langle\langle\rangle\rangle$", obtained by the successive application of rules: $S \xrightarrow{3} BC \xrightarrow{4} ASC \xrightarrow{6} AS\rangle \xrightarrow{2} AAC\rangle \xrightarrow{5} A\langle C\rangle \xrightarrow{6} A\langle\rangle\rangle \xrightarrow{5} \langle\langle\rangle\rangle$. Some of these rules could have been applied in a different order, but all such orderings correspond here to the same parse tree (the red one in Figure 1.1).

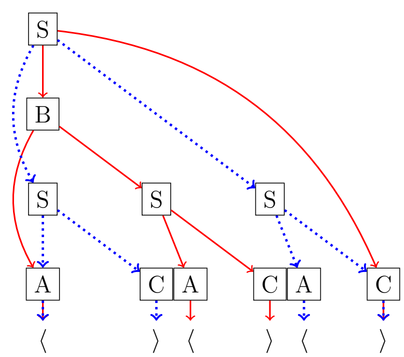

Figure 1.1 Grammar parse tree for the two words of length 4 recognized by the grammar shown in Section 1.1. The first word is in red, the second is in blue.

1.1.1 Chomsky Normal Form

Due to the particularities of our solver (which we will expand on later), we have to convert our grammar into its Chomsky Normal Form to be able to use it. A grammar is said to be in Chomsky Normal Form if it follows a few additional rules on top of those of a CFG. No production may contain the null symbol ϵ and the right-hand side of any production must be either: a single terminal or two nonterminals. The following CFG could be converted to be in Chomsky Normal Form by applying four steps.

$$\mathcal{N} = \{S, X, Y, Z\}$$
$$\Sigma = \{a, b\}$$
$$\mathcal{R} = \{S \to XYZ,\ X \to aXb,\ X \to \epsilon,\ Y \to aa,\ Y \to bb,\ Y \to X,\ Z \to abba,\ Z \to XabY\}$$
$$S = S$$

0. Initial grammar

$$\begin{aligned} S &\to XYZ \\ X &\to aXb \mid \epsilon \\ Y &\to X \mid aa \mid bb \\ Z &\to XabY \mid abba \end{aligned}$$

1. Remove null productions

$$\begin{aligned} S &\to XYZ \mid XZ \mid YZ \mid Z \\ X &\to aXb \mid ab \\ Y &\to X \mid aa \mid bb \\ Z &\to XabY \mid Xab \mid abY \mid ab \mid abba \end{aligned}$$

2. Replace unit productions

$$\begin{aligned} S &\to XYZ \mid XZ \mid YZ \mid XabY \mid Xab \mid abY \mid ab \mid abba \\ X &\to aXb \mid ab \\ Y &\to aXb \mid ab \mid aa \mid bb \\ Z &\to XabY \mid Xab \mid abY \mid ab \mid abba \end{aligned}$$

3. Shorten the right-side to two tokens

$$
\begin{aligned}
S &\to XC \mid XZ \mid YZ \mid XD \mid XE \mid EY \mid ab \mid EF \\
X &\to aG \mid ab \\
Y &\to aG \mid ab \mid aa \mid bb \\
Z &\to XD \mid XE \mid EY \mid ab \mid EF \\
C &\to YZ \\
D &\to EY \\
E &\to ab \\
F &\to ba \\
G &\to Xb
\end{aligned}
$$

4. Create unit productions for terminal tokens

$$
\begin{aligned}
S &\to XC \mid XZ \mid YZ \mid XD \mid XE \mid EY \mid AB \mid EF \\
X &\to AG \mid AB \\
Y &\to AG \mid AB \mid AA \mid BB \\
Z &\to XD \mid XE \mid EY \mid AB \mid EF \\
A &\to a \\
B &\to b \\
C &\to YZ \\
D &\to EY \\
E &\to AB \\
F &\to BA \\
G &\to XB
\end{aligned}
$$

1.2 Chemistry

This section will detail different important notions in organic chemistry needed to understand the rest of this work.

1.2.1 Chemical Notation

Atoms are the building blocks of molecules and the bonds they can make are what allows the formation of complex structures. In organic chemistry, the atoms of interest are: Boron (B), Carbon (C), Nitrogen (N), Oxygen (O), Fluorine (F), Phosphorus (P), Sulphur (S), Chlorine (Cl), Bromine (Br) and Iodine (I). The number of bonds an atom can make is limited by the electrons in its valence shell, also called valence electrons. This valence shell refers to the outermost layer of electrons.

A valence shell is made up of multiple subshells of different energy levels: 1s, 2s, 2p, 3s, 3p, 3d, etc. Each of these subshells can hold a different number of electrons and the valence shell of a given atom is said to be complete when the outermost subshells are full. This often comes back to reaching the configuration of a noble gas, which are the rightmost atoms in the periodic table.

Having a complete valence shell is the stable configuration that most atoms tend towards. To achieve this, atoms will make ionic bonds, a bond where an electron is taken from another atom, or covalent bonds, a bond where an electron is shared by two atoms. In the case of organic molecules, we will usually only consider covalent bonds.

If we take Hydrogen and Carbon as examples, two of the more common atoms in organic chemistry, they need one and four more electrons respectively to complete their valence shell. The earlier atoms used in organic chemistry have the following number of valence electrons: Boron has 3; Carbon has 4; Nitrogen and Sulfur have 5; Oxygen and Sulphur have 6; Fluorine, Chlorine, Bromine and Iodine have 7. They can do this by making the corresponding number of covalent bonds required to complete their valence shell (commonly represented as line segments between atoms; see e.g. Figure 1.2A).

As seen in Figure 1.2B, to reduce the visual clutter of molecular graphs, Carbon and Hydrogen atoms are omitted. Carbon atoms are simply vertices with no letter indicating anything and Hydrogen atoms are implicitly present to complete the valence shell of any atoms that appear to be missing a bond.

1.2.2 Hydrogen Bonds

Hydrogen bonds are inter-molecular bonds caused by polarized molecules. They require a donor and an acceptor. Covalent bonds do not always equally share the shared electron, specifically, the more electronegative an atom is, the more it pulls on the shared electron. The electronegative atoms that interest us in the context of organic molecules are: Fluorine (F), Sulphur (S), O (Oxygen) and N (Nitrogen).

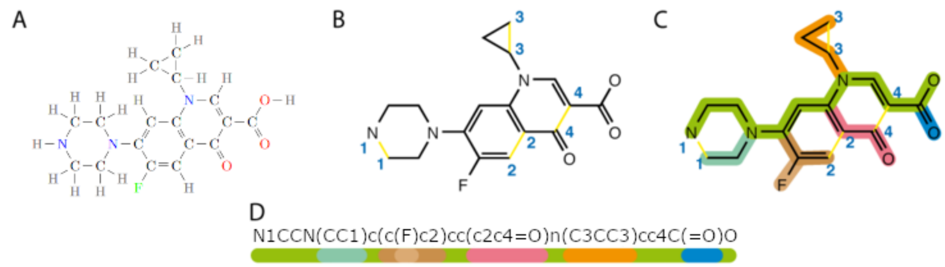

Figure 1.2 Deriving a SMILES representation for a molecule (reproduced in part from [5]). The structural formula of the molecule (A), its skeletal formula stripped of all hydrogen atoms and with broken cycles (B), the selected main path (shown in green) and branches (C), and the corresponding SMILES notation (D).

The **donor** is an electronegative atom linked to a Hydrogen atom. By pulling on the shared electron more than the Hydrogen atom does, the electronegative gains a partial negative charge. Inversely, the Hydrogen atom gains a partial positive charge.

The **acceptor** is an electronegative atom with a free electron pair on its valence shell. This electron pair, can then attract the partially positively charged Hydrogen from the donor.

This attraction, between two different polarized molecules, is what we call a Hydrogen bond. The most famous example of this is in water and is the reason for many of water's interesting properties (cohesion, high boiling point, high heat capacity, surface tension, expands when frozen, etc). In this case, the Oxygen atom is both the donor and the acceptor. The Oxygen atom, acting as the acceptor, is negatively charged and can attract the positively charged Hydrogen atoms from other water molecules. The same atom will donate its positively charged Hydrogen atoms to other Oxygen atoms.

1.2.3 Molecule Encodings

Molecules can be encoded in many different ways. Two common methods are representing molecules as graphs or as one-dimensional strings. We will be using a one-dimensional encoding in our work to simplify the representation and potentially allow a combined model using NLP models. We will present different one-dimensional encodings used in the molecule discovery field.

InChI

International Chemical Identifier (InChI) is a notation standard introduced by the International Union of Pure and Applied Chemistry (IUPAC) [6]. It provides a unique one-

Encoding	Representation
InChI	InChI=1S/C17H18FN3O3/c18-13-7-11-14(8-15(13)20-5-3-19-4-6-20)21(10-1-2-10)9-12(16(11)22)17(23)24/h7-10,19H,1-6H2,(H,23,24)
SMILES	N1CCN(CC1)c(c(F)c2)cc(c2c4=O)n(C3CC3)cc4C(=O)O
canonical SMILES	O=C(O)c1cn(C2CC2)c2cc(N3CCNCC3)c(F)cc2c1=O
DeepSMILES	O=CO)ccnCCC3)))cccNCCNCC6))))))cF)cc6c%10=O
SELFIES	[N][C][C][N][Branch1][Branch1][C][C][Ring1][=Branch1][C][Branch1][=Branch1][C][Branch1][C][F][=C][=C][C][=Branch1][=Branch1][=C][Ring1][Ring2][C][=O][N][Branch1][=Branch1][C][C][C][Ring1][Ring1][C][=C][Ring1][Branch2][C][=Branch1][C][=O][O]

Table 1.1 Different encodings of the molecule shown in Figure 1.2. DeepSMILES could not originally encode the SMILES string, we converted the molecule to canon SMILES for it to encode it correctly.

dimensional representation of the molecule. The encoding contains information on both the structure and certain properties of the molecule. This representation was not adopted by the automated molecule discovery research community because of its low readability by both humans and machines. Instead, it has become commonly used in indexing and searching tasks (*e.g.* databases).

SMILES

SMILES is a one-dimensional string representation for molecular encoding [7]. This encoding is much simpler than InChI, only maintaining the structural information required to reconstruct the molecule. However, any lost information can be recovered using different techniques. Admittedly, the process can be difficult and time-intensive to guarantee accurate results.

Do this: Get sources showing which ppl do this

If we picture a molecule as a graph where every vertex is an atom, a SMILES string would be the order in which we explore a tree-representation of the graph using a Depth-First Search (DFS). Since SMILES is a DFS over a tree, any cyles that were in the original graph would be lost. Thankfully, SMILES considers this by designating a specific token to represent broken cycle bonds. This prevents losing the cycles when we convert the graph into a tree. These tokens are represented by number tokens as seen in Figure 1.2B and, later, in D. Similarly, when we reach a branching path in the tree, one side is chosen as the main branch, which is shown in green in Figure 1.2C, while the other side is written between parentheses to indicate that it is a branch.

A very common concept in organic chemistry is aromatic rings. These are usually 5 or 6 atoms in a ring, the ring bonds alternate between single and double bonds. Due to their frequency, the SMILES language has started using a shorthand for it by writing atoms in aromatic rings using lowercase letters. This allows us to omit the alternating single and double bonds during writing and makes aromatic rings much more visible when reading a molecule. Kekulized SMILES keeps these bonds explicit, but non-kekulized SMILES is preferred.

This encoding has gained a lot of popularity in the automated molecule discovery research community due to its age and ease of readability. Since its introduction, it has become the most popular string representation in automated discovery. However it comes with certain issues that we must address. Unlike InChI, SMILES does not offer a unique encoding for each molecule. It is therefore possible to generate two different strings that describe the same molecule. Another prominent issue is linked to SMILES' special tokens. By requiring an opening token, as is needed to describe cycles, branches and isotopes (which we did not describe), any string that does not have corresponding open and close tokens is syntactically invalid.

Gilles commented: À déplacer là où tu parleras des limites de la ML

This can be problematic in token-by-token generation if these rules are not hard constraints, which is the case in ML techniques since they lack the ability to impose long-term structure.

SMILES also has no check on the valence shells of the atoms within it. For example a Carbon atom, which wants to make 4 bonds to complete its valence shell, could be placed in such a way that it has 6 bonds.

There are some ways to generate canonical SMILES strings (*i.e.* unique for a given molecule), however no consensus has been reached on which method to use.

This is the encoding that we use during our work, mainly due to its popularity within the automated molecule discovery community, which allowed us to find documentation and tools that helped during the work. We will present two other molecule encodings that were introduced to resolve issues within SMILES, but they were not used due to their relatively new appearance and, consequently, to their smaller research community.

DeepSMILES

DeepSMILES was introduced to answer some of SMILES' shortcomings [8]. It changes how branches and cycles are represented so that only one token is required. Instead of representing cycles using numbers as tags, they instead use numbers to indicate the size of the cycle and place the number at the end of the cycle. Similarly, branches no longer require opening branch tokens, instead they place as many branch closing tokens as there are atoms in the described branch.

Unfortunately, DeepSMILES is not perfect and sometimes fails to encode a molecule correctly. The example molecule used in Figure 1.2 cannot be directly converted into DeepSMILES from its SMILES format. This is a big issue, since the encoding could fail based on which bond in a cycle we choose to break to convert the graph into a tree. However, this can be avoided by first converting the molecule to canonical SMILES. The molecule still has an encoding in DeepSMILES, as shown in Table 1.1.

SELFIES

Similarly to DeepSMILES, SELF-referencIng Embedded Strings (SELFIES) [9] was introduced specifically for ML applications, its language having been designed to minimize syntax invalidity and simplify the structure for ML models.

To resolve some of SMILES' syntax problems (*i.e.* branch and cycle invalidity), it associates each token to a numeric value. It then overloads the tokens following cycle or branch tokens, replacing them by their numeric value. In the case of branches, they place the token at the start of the branch and the overloaded value tells us how many of the future tokens are a part of this branch. For cycles, the token is placed at the end of the cycle and the overloaded value indicates how many atoms back we have to go to find the start of the cycle.

Another important difference is that all tokens are described between square brackets to remove some ambiguity. In SMILES, the square brackets are omitted for common atoms to improve readability.

1.2.4 Lipinski's Rule of Five

Lipinski's Rule of Five is a set of rules describing properties that orally administered drugs tend to respect. While there are only four rules, each rule contains a value that is a multiple of five, which is where the name comes from.

The rules are as follows:

- The molecular weight must not exceed 500 Daltons.
- There must not be more than 10 Hydrogen-bond acceptors.
- There must not be more than 5 Hydrogen-bond donors.
- The logP must not exceed 5.

The molecular weight is the simplest property to understand. By limiting the weight of the molecule, we tend to avoid molecules that are too large. It is important to note that Daltons are on a one-to-one scale with g/mol, which is the more commonly used unit.

Hydrogen-bond acceptors as seen in section 1.2.2, are electronegative atoms (*e.g.* F, S, N, O) with a free electron pair on their valence shell to act as an acceptor for the Hydrogen-bond.

Hydrogen-bond donors , as seen in section 1.2.2, are electronegative atoms linked to a Hydrogen atom. This Hydrogen atom will allow the Hydrogen-bond with an acceptor.

The logP is an evaluation of how lipophilic or hydrophobic a molecule is, *i.e.* how easily the molecule dissolves in fats as opposed to water. This is relevant when trying to control how a drug is absorbed in the human body.

1.3 Constraint Programming

Constraint Programming is a complete, heuristic guided search method which excels at ensuring the respect of constraints while generating a solution. It is complete in that if a solution exists in a given search space, a CP model is guaranteed to find it. By using heuristics as well as constraint propagation (more on that later), it can be much faster than a simple brute force of all possible solutions.

We will first define how a simple CP model functions. We will cover the initial problem declaration, the constraint declaration to describe the problem and finally the solving process and its intricacies (constraint propagation, branching decisions, backtracking). Once that is covered, we can expand on this topic by introducing CPBP

Do this: cite BP from original paper

which is an improvement over standard constraint propagation and leads to more informed decisions. We use CPBP in our work since it tends to yield better results and allows for the combination with a ML model as we will describe later.

1.3.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined in three parts:

- The variables making up the problem, defined as the finite set $\mathcal{X}$

- The domains of these variables, defined as a finite set of values D. Each variable can have its own domain
- The constraints, each of which is applied to a subset of the variables, defined as a set of constraints C.

There are a finite number of **variables** defined in the set $\mathcal{X}$. Each of these variables has its own **domain** as is defined in the set D, which contains the possible values that a variable may take on. Finally, we define a finite number of **constraints**, each of which is applied on a subset of the variables. Each variable must then be assigned a value from its domain such that it respects all the applied constraints. If such an assignment is possible for all the variables, that is a solution to the problem.

If we take the Sudoku problem as an example, a classic and very commonly seen problem, we can define it as a CSP as follows. Our **variables** will be each tile in the 9x9 grid. While this gives us the layout of our problem, we must define the possible values for each variable to be able to solve this problem. All the variables can take on the same values and so we can define the **domain** as being the integer values between 1 and 9 inclusively.

We could represent this using a 2-dimensional array of variables like so:

$$tile[i][j] \in \{1, 2, \ldots, 9\} \mid i, j \in \{1, 2, \ldots, 9\} \tag{1.1}$$

All that is missing are the constraints, which are the source of the complexity of the problem.

The **constraints** in a Sudoku are fairly straightforward, lines, columns and all 3x3 sub-grids within the total grid may not contain any repeat values. In the CP community, this type of constraint is very common and is called an `alldifferent` constraint. The Sudoku problem would therefore have the following constraints:

$$\texttt{alldifferent}(tile[1][j], tile[2][j], \ldots, tile[9][j]) \ \forall\ j \mid j \in \{1, 2, \ldots, 9\}$$
$$\texttt{alldifferent}(tile[i][1], tile[i][2], \ldots, tile[i][9]) \ \forall\ i \mid i \in \{1, 2, \ldots, 9\}$$
$$\begin{gathered}\texttt{alldifferent}(\\ tile[3u+1][3v+1], tile[3u+1][3v+2], tile[3u+1][3v+3],\\ tile[3u+2][3v+1], tile[3u+2][3v+2], tile[3u+2][3v+3],\\ tile[3u+3][3v+1], tile[3u+3][3v+2], tile[3u+3][3v+3],\\ ) \ \forall\ u, v \mid u, v \in \{0, 1, 2\}\end{gathered}$$

Overall, we would need 81 variables to define this CSP as well as 27 constraints. Each of our variables could take on any of the 9 possible values in their domain.

1.3.2 Domain Filtering

As mentioned in the previous section, each constraint is applied to a subset of the variables in the problem definition. When a constraint is declared, a filtering algorithm that is specific to that constraint will eliminate values that are inconsistent.

The simple example below illustrates how a constraint can filter a variable's domain after being declared.

$$\begin{gathered} x \in \{2, 3, 4\}\\ y \in \{1, 2, 3\}\\ x \leq y\\ x \in \{2, 3, \cancel{4}\}\\ y \in \{\cancel{1}, 2, 3\}\end{gathered}$$

Both variables initially contained a value that would always breach the constraint if chosen. A value such as that one is said to have no support, *i.e.* there are no solutions to the current constraint that contain this value. A visual representation of this can be seen in Figure 1.3, where a constraint is applied to two different variables and both have their domain filtered. While we do not know all the solutions to a problem in all cases, we can use logical processes to determine values that would guarantee a breach of the constraint.

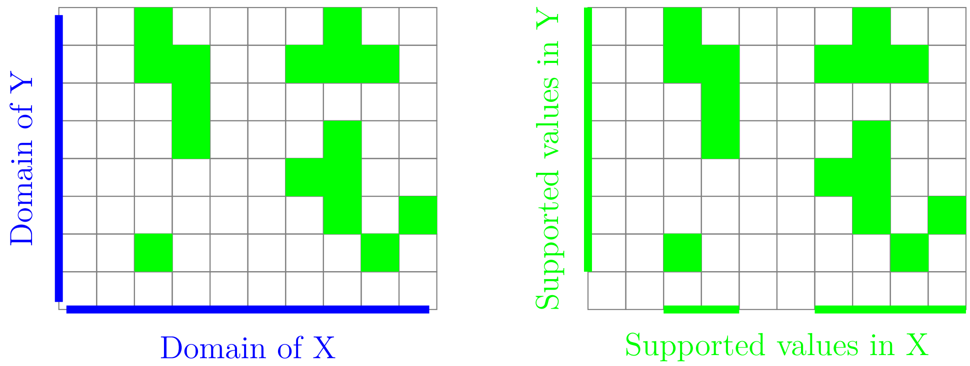

Figure 1.3 A constraint is declared on both variable X and Y. Valid combinations to the constraint are illustrated by green points on the grid. Values that have no support (no solution to this constraint contains these values) are removed from the domain of the variable. This can be seen as a projection of the solution space onto the domain of each variable.

1.3.3 Constraint Propagation

Now that we have declared our constraints, the solver begins propagating the consequences of these constraints. Each constraint in the queue communicates to the variables it affects which values in the domain have to be filtered out. Once a variable's domain has been changed, it notifies the constraints that are affected by the change and those constraints are then added to the queue again.

The solver continues propagating the consequences of the constraints and updating domains until it reaches one of three situations:

1. The queue is empty, but there remain unassigned variables.
2. All variables have been assigned a value, this is a solution to the problem.
3. One of the variables' domain has been completely filtered, there is no solution in the current state of the problem.

In the first case, there is nothing else to deduce with the information currently available and the solver has to make a branching decision from the current state. Any time we reach one of the three cases above, we can consider that state as being a node in the search tree. The solver makes a branching decision from the current node and propagates the consequences of this decision until it reaches another node to handle.

In the second case, the solver has found a solution and can add it it the solution set. Once the solution has been found, we backtrack to the previous node in the search tree and search along the other branches.

Finally, if we reach an unsatisfiable state, the solver backtracks to the previous node and continues its search from there.

Since we have a finite number of values, we know that this process will eventually end and we will either find a value that respects the constraint, or, find that the constraint cannot be satisfied.

To continue with the example of a Sudoku, a classic way humans continue solving, once they reach a dead end in their reasoning, is by assigning a value to a tile and seeing if they reach a contradictory state. If they do, then they know their choice was wrong and they can eliminate that possibility.

1.3.4 Marginals-Augmented Constraint Programming

In an ideal world, if we knew every possible solution to a problem, we could use the values within the solution to inform our search and avoid bad branching decisions. This is especially useful when we consider bigger problems that might have a huge combinatorial space to explore.

Marginals-augmented constraint programming is the idea of guiding our branching decisions by counting the number of solutions to a constraint that contain a given value for a given variable as seen in Figure 1.4. The difficulty of this task is that it requires an efficient algorithm which can predict the number of solutions without finding and enumerating all possible solutions to the constraint.

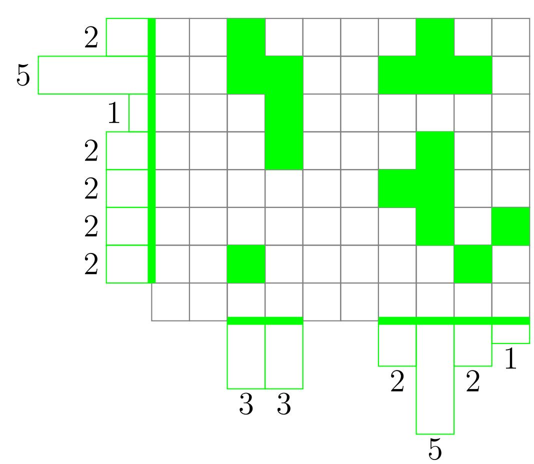

Figure 1.4 Taking the same example as in Figure 1.3, we can count how many valid solutions contain a given value for both variables X and Y. The numbers seen on the left and the bottom of the grid indicate the number of solutions containing the value. This can help guide our branching decisions towards solution-dense regions in the search space.

One use of these marginals is to change standard constraint propagation to contain more information. Belief Propagation (BP) does this by modifying the message that constraints send variables. Instead of sending a message containing a binary representation of which values in the domain have a support, the messages are modified to communicate the probability of a value being contained within a solution as seen in Figure 1.5. This allows the solver to avoid branching on values that have a very small chance of being valid.

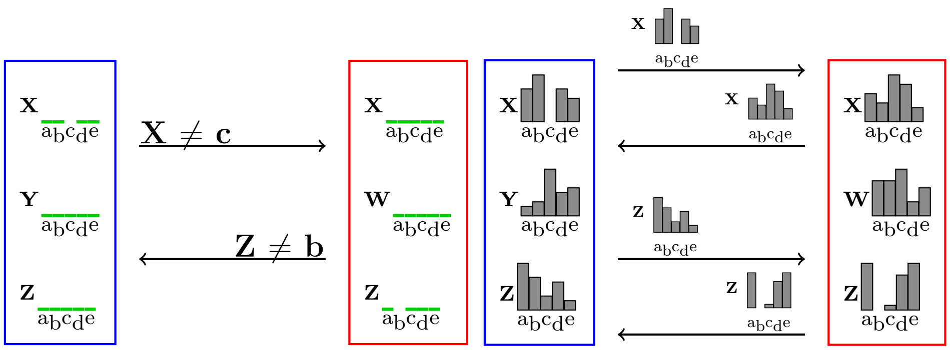

Figure 1.5 BP replaces standard constraint messages, which consist of a binary message indicating which values are supported in the domain, with a probabilistic distribution over the domain. As we can see, instead of communicating that the value c for variable X lacks a support, the blue constraint communicates that $X = c$ has a 0% chance of being in a valid solution. This ensures that we can still communicate what values must be filtered out, but we also gain information on the other values in the domain.

When multiple constraints interact on one variable, they each simultaneously communicate to the variable what they estimate the probability distribution to be. The variable then merges these probabilities into the final values as seen in Figure 1.6.

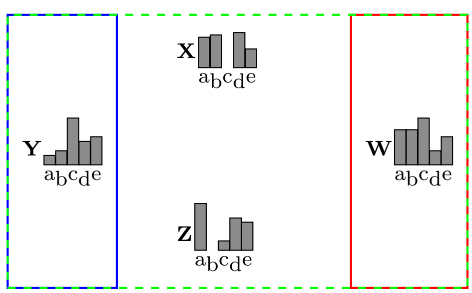

Figure 1.6 The variables which are affected by multiple constraints (X and Z) merge the communicated probabilities that were communicated by the different constraints. X filters out the value c and Z filters out b, as was communicated by the constraints seen in Figure 1.5.

1.3.5 Among Constraint

1.3.6 Element Constraint

1.3.7 Grammar Constraint

1.3.8 Regular Constraint

1.3.9 Sum Constraint

1.3.10 Table Constraint

Short Table Constraint

1.4 Neural Networks for Natural Language Processing

This section will give simplified descriptions of different necessary notions for this work.

1.4.1 Neural Network

Neural Networks are a Machine Learning architecture that get their name from their resemblance to a brain. Similarly to a brain, a Neural Network (NN) has neurons that communicate with each other to learn how to solve the task at hand. The simplest network we can make is made up of one input layer and one output layer. To improve the learning capabilities of this model, we can add hidden layers, which are neuron layers between the input and output ones. A model which has more than 2 hidden layers is called a Deep Neural Network.

The input layer contains as many nodes as the problem has inputs, each node representing one value. Similarly, the output layer contains as many nodes as the problem has. A model can contain any number of hidden layers, each of which is made up of any number of nodes. Each node in a hidden layer takes its inputs from every node in the previous layer and, inversely, sends its output to every node in the next layer.

In a standard model, the node sums up the product of all the inputs and their associated weight before applying an activation function to the sum. This result is the node's output and will be passed on to the next layer where it will be used as the input in a similar operation. For the model to learn complex relations, it is critical that the activation function used is non-linear. If the activation function were linear, the entire model would collapse back into a simple linear equation.

To find the right weights, the model must first be trained on a part of the total dataset. During training, the model computes the error between the expected result and the predicted one and then backpropagates this error from layer to layer. Each layer then recalculates the weights of its inputs based on the obtained error before sending a modified message to the previous layer.

From there, the trained model can be given any problem input and will calculate the predicted output based on its internal weights.

1.4.2 Transformers

Transformers [10] are a ML architecture based on encoders and decoders. The model first passes the input through an encoder, that encoded sequence is then used by the decoder to generate an output one token at a time.

The encoder is made up of multiple identical layers, each composed of two sub-layers: a multi-head attention layer and a feed-forward network. The multi-head attention layer is an improvement over standard attention models and allows the model to learn more complex relations. The input embeddings received by the multi-head attention sub-layer maintain more context during training and generation by encoding both the input sequence as well as positional information.

The decoder is also made up of multiple identical layers, each composed of three sub-layers: a masked multi-head attention layer, a standard multi-head attention layer and a feed-forward network. The masked multi-headed attention layer's output is then fed into the next multi-headed attention layer with the encoded input from the encoder. This is finally passed through a feed-forward network. The input received by the masked multi-headed attention is an embedding which encodes both the current output sequence as well as positional information on the tokens.

Once this final output is calculated, we apply a softmax on it to get the probabilities for the next token.

1.4.3 Large Language Model

LLMs were introduced shortly after the proposal of transformers in 2017. Following transformers, Bidirectional Encoder Representations from Transformers (BERT) [11] was introduced as an encoder-only architecture and can be considered the start of LLMs. However, this type of architecture came into the limelight with the Generative Pre-trained Transformer (GPT) models from OpenAI.

The specific GPT model that interests us is the GPT-2 model [12], it is what we use in our architecture. Similarly to what was introduced for GPT-1 [13], the model is a large decoder layer, as seen in the transformers. An important difference is that the second multi-head attention sub-layer is removed from each of the identical layers in the decoder.

These models are usually trained to complete many different tasks, however by training one on a SMILES dataset, we can get a GPT-2 model to generate molecules based on what it has seen.

1.5 Problem Statement

As mentioned previously, drug discovery is both time-consuming and costly and automated drug discovery has been an important field of research to reduce these costs. While ML methods have been gaining a lot of popularity in the field, those techniques suffer from a lack of long-term structure. To address this, CP is a natural answer since it provides the lacking long-term structure.

Do this: Cite works using CP could help

However, while CP is used in the domain, there isn't much work

Do this: I found none, but to be investigated further

relating to generating molecule candidates using CP.

We believe that a CP model would be beneficial and would reduce the number of invalid molecules generated.

More importantly however, a CP model that generates valid molecules could then be used to target property-specific molecules using constraints to eliminate undesirable options.

The issue of using CP for this problem is the size of the search space to explore. By using BP, we believe that the search will be better guided towards a solution and require less backtracking. However, it remains to be seen if the added cost for the BP increases the overall time to solve.

Finally, we believe that by combining a trained token-by-token generating ML model with our CPBP model, we might get molecules similar to what is being used today (molecules in datasets) while still maintaining the long-term structure of the CP model.

1.6 Research Questions

During our research we will answer the following questions:

1. Can we use CP to model valid molecules in a one-dimensional encoding?
2. Can we use CP to model desirable molecular properties in SMILES molecules?
3. Can Belief Propagation be used to better guide a solver towards a solution?
4. How can we combine a CP model with a NLP model to improve the realism of generated sequences and is it an effective method?

1.7 Thesis Outline

The rest of this thesis is organized in the following chapters:

- Chapter 2 goes over the necessary concepts to understand the rest of the paper.
- Chapter 3 provides a general overview of the different techniques currently in use.
- Chapter 4 presents our base model as well as the methods used to model valid molecules as per our first research question.
- Chapter **??** expands on the previous section and introduces ways to model molecular properties using CP. This section addresses our second research question.
- Chapter 5 details how we combine our CP model with a NLP model.
- Chapter 6 goes over the paper's contributions, its limitations and potential ways to improve this in future work.

CHAPTER 2 BACKGROUND

2.1 Chemistry

2.1.1 Organic Chemistry Notation

2.1.2 Lipinski's Rule of Five

2.2 Constraint Programming

2.3 Natural Language Processing

CHAPTER 3 LITERATURE REVIEW

This chapter will focus on an overview of the current state of the art on this topic of research.

Drug discovery, and molecule design in general, is a vast topic. There are many different methods that are applicable to the problem.

A recent survey by Du et al. [2] presents various representation formalisms. It covers one-dimensional representations such as SMILES and InChI as well as two-dimensional and three-dimensional representations. It describes some of the main problems tackled, and an array of computational methods used to solve them, mostly generative machine learning but also combinatorial solvers.

Among the current challenges for deep generative models, they mention the difficulty of exploring little known/seen areas of the molecular design space (the common out-of-distribution generation issue) and the need for lots of training data (generation in low-data regime issue i.e. high sample complexity). They also mention as opportunity the generation of specialized molecules with more complex structure.

3.1 NLP applied to drug discovery

3.2 CP applied to drug discovery

3.2.1 Combining CP with ML

CHAPTER 4 MODELING VALID MOLECULES USING CP

In this chapter, we will put forward a way to model that can represent valid molecules using CP. As mentioned previously, we choose to use SMILES to encode our molecules. This is a simple and easy-to-read one-dimensional molecule representation which can easily be modelled by CP.

We first describe the grammar that was required to describe the SMILES language. This is the key component that allows our model to generate molecules in the right encoding. We will then give a formal definition for our model before finally explaining our experiments and results.

4.1 SMILES Grammar

SMILES was developed for applications in organic chemistry, this can be seen in some of its rules. For example, the addition of tokens to describe aromatic rings is something that was added to simplify the notation, specifically due to the common occurrence of these rings. Another example is that simple atom tokens with no descriptor tokens have an implied complete valence shell (*i.e.* the atom is in its stable state). SMILES requires an explicit indication when an atom does not respect its valence shell, whether it has more or less than the expected amount. This is why the grammar we chose to use, which is a variation of the one described by Kraev in his work [14], ensures that atom valences are respected.

The original work uses masks in addition to this grammar to completely avoid invalid outputs. The first mask handles numerical assignment for cycles, guaranteeing that cycles are numbered correctly. The second mask avoids making cycles that are too small (*i.e.* cycles of 2 atoms) and cycles that are too long. They limit their cycle length to 8 based on what they observe in their database [14].

Gilles commented: Include important rules here from grammar

We address both of these issues by modifying the base grammar and adding new constraints as will be discussed later. The final grammar used for validity can be seen in Appendix A.

Padding

For the purpose of using this grammar in our CP model, we add padding tokens that can complete the end of a molecule. This will allow our model to generate any molecule up to the size instead of giving it a fixed length, allowing for a more versatile model. We chose "_" as our padding token.

An easy way to make this change is to create a new starting token that can be developed into the old start token and any number of padding tokens (including none). This change was not influential on the performance of the algorithm and allows for more options during generation.

Hydrogen tokens

Some Hydrogen tokens can be included in the molecule. These can be followed by a number to indicate the number of Hydrogen atoms present. We change these tokens to directly include the number. Instead of needing two tokens ("H" and "3") we now use one token ("H3") made up of two characters.

This avoids confusing Hydrogen count tokens for cycle tokens and improves our model's understanding of what it is generating.

Cycle-length limit

The final required modification we make to our grammar is to limit the cycle length. We wish to ensure that cycle lengths remain in the desired range (between 3 and 8 inclusively). In datasets of known drug-like molecules, long cycles are infrequent. In the dataset MOSES [15], containing near two million molecules, no molecule features anything greater than a length-6 cycle. However, in another dataset containing near 250 thousand molecules, Zinc_250k [16], we can find up to length-8 cycles. This seems to indicate that long cycles are either undesirable or lead to chemically unstable molecules (*i.e.* molecules that we cannot synthesize).

We achieve this by limiting the number of tokens that a cycle production can be developed into. This information must be encoded in nonterminals where a larger cycle nonterminal can be rewritten as an atom and a smaller cycle nonterminal as seen in Table 4.1.

This change alone guarantees that any nonterminal "num" will have another nonterminal "num" within an acceptable distance. However, this does not guarantee that the nonterminal "num" will be developed into the same cycle number. Take the unfinished chain "C*num*CCCCC*num*NC*num*CCCCC*num*" as an example. While we would expect the finished chain to be "C1CCCCC1NC2CCCCC2", the current grammar would also accept "C1CCCCC2NC2CCCCC1", which results in both cycles being the wrong size.

This was a problem we ran into fairly quickly after applying the cycle size limit changes to

1	valence_2	→	valence_4_num1 "(" cycle1_n_bond ")"
2	cycle1_n_bond	→	cycle1_7_bond
3	cycle1_7_bond	→	cycle1_6_bond
4	cycle1_6_bond	→	cycle1_5_bond
5	cycle1_5_bond	→	cycle1_4_bond
6	cycle1_4_bond	→	cycle1_3_bond
7	cycle1_3_bond	→	cycle1_2_bond
8	cycle1_7_bond	→	valence_2 cycle1_6_bond
9	cycle1_6_bond	→	valence_2 cycle1_5_bond
		...	
10	cycle1_2_bond	→	valence_2 valence_2_num1

Table 4.1 Cycle degradation example from the grammar. Rule 1 shows how a cycle is started, in this case it is started in a branch. The nonterminal outside the branch, "valence_4_num1", is a part of the cycle and must be taken into account for the length. Rule 2 was added to easily change the starting size of the cycle. Rules 3-7 allow for cycles to get smaller without adding another token, this is how we allow smaller cycles than 8. Rules 8-10 are the development of the cycle, we add a nonterminal and go down to the cycle size down. Rule 10 is special since it is the end of a cycle, so we first place a nonterminal followed by a nonterminal that is numbered to indicate the end of the cycle. The name of the cycle nonterminal contains information on what it will develop into: "cycle1" means it is the cycle identified by the "1" token, "_n_" indicates how many more atoms this nonterminal will develop into, "bond" tells us that it is a simple bond that is expected.

the grammar, resulting in one very long cycle and one small one instead of two appropriate cycles. The solution was to integrate into the left-hand side of the production information about which cycle is being developed as can also be seen in Table **??**.

As Kraev mentions in the original paper [14], this change will make the grammar grow very quickly in size based on the maximum number of cycles allowed (not to be confused with the maximum cycle-length). Therefore, it was critical to limit the number of cycles to avoid drastically increasing the size of our grammar. After examining the two datasets at our disposal, 6 molecules from the ZINC250K dataset [16] and 4 from the MOSES [15] dataset exceed 6 cycles. These datasets contain, respectively, 250K and 2M molecules. Based on this, we decided to limit the number of cycles to 6, seeing as it does not exclude many molecules from the ones observed in the known drugs datasets.

All of these changes ensure that cycles have an appropriate length. However, this does increase the size of the grammar. While the original CFG from Kraev contained 34 terminals, 36 nonterminals and 138 productions, the current CFG now has 32 terminals, 194 nonterminals and 538 productions.

We have two fewer terminals overall because we removed 3 cycle numbering tokens ("7", "8", "9") as well as the token that is used to number cycles using two digits ("%"). However, we did add the padding token, "_", and we added the "H3" token to fully distinguish cycle numbering tokens from other numeric tokens. Notice we did not add the "H2" token as it was not present in the grammar previously.

4.2 Our Model

This section will first describe our model's variables and their domains. We will then go over formal definitions of the constraints used to model valid molecules and break certain easily-identifiable symmetries. Following the validity constraints, we define constraints that target specific structures in our generated molecules. Finally, we will define the constraints required to target desirable properties.

4.2.1 Variables

We chose to limit the size of our molecules to 40 tokens. Since we use padding tokens, this means we can model any molecule of size 40 or less, which represents 83% of all molecules in the two datasets we chose to use in our work (ZINC250K and MOSES). This decision ensures the problem is representative of real-life molecules observed in our datasets and hence provides a meaningful empirical study.

We define 40 variables, one for each token in the molecule such as $\mathcal{X} = X_1, X_2, \ldots, X_{40}$. Each token starts with the same domain, containing every possible terminal in the SMILES grammar alphabet.

@Gilles, I have a question: Oui, on dit plus tot que les atomes qui nous intéresses sont les atomes organiques

This is formally defined as

$D(X_i)$ ={ Br, Cl, F, I, C, N, O, S, c, n, o, s, 1, 2, 3, 4, 5, 6, (,), =, #, [,], +, -, H, H3, /, \, @, _ }

This setup allows any combination of SMILES tokens of size 40, including invalid ones. To ensure validity, we use three constraints as described in the following subsection.

4.2.2 Validity Constraints

This section will answer our first research question: Can we use CP to model valid molecules using a one-dimensional encoding? With the following constraints, our model will be able to model valid SMILES molecules.

Grammar Constraint

The grammar constraint is responsible for guaranteeing the SMILES syntax in our generated molecules. The grammar constraint is a global constraint applied to all variables in the model. It also requires the CFG that we defined earlier in Section 4.1.

$$\texttt{grammar}(\langle X_1, X_2, \ldots, X_{40} \rangle, \mathcal{G}_{\texttt{SMILES}})$$

This constraint does a lot of the work in ensuring that the generated output is a valid SMILES string. It guarantees:

1. No valence mistakes. Atoms used in the molecule will respect the expected number of bonds to complete their valence shell.
2. Opened cycles are appropriately paired to another cycle token to close it.
3. Cycles respect a maximal length to avoid non-sensically large cycles that do not appear in drug-like molecules in known datasets.
4. Any branch token has a corresponding opening/closing branch token.

This constraint on its own would already generate valid SMILES strings. However, we add two more constraints to improve the readability of our generated results and avoid symmetries.

Cycle Parity Constraint

The cycle parity constraint ensures that all cycle tokens in the grammar's alphabet are used either twice or never. This avoids having two cycles with the same cycle identifier. In classic SMILES notation, the same cycle identifier can be reused if there is no ambiguity.

We decided not to allow the reuse of cycle tokens, since adding checks to avoid ambiguity in the grammar would make it much more complex, as we would need to track which cycles are currently open at all times. This simple constraint avoids the generation of ambiguous molecules while avoiding a larger grammar that would have taken a long time to design.

$$\textsc{Among}(\langle X_1, X_2, \ldots, X_{40}\rangle, \{j\}, \{0, 2\})\ \forall\ j \mid 1 \leq j \leq 6$$

Cycle Numbering Constraint

The cycle numbering constraint is a symmetry-breaking constraint. It avoids using larger cycle identification tokens before smaller ones. In other words, the first opened cycle is identified using the token "1", the second will be identified using "2" and so on and so forth. This ensures there is only one possible cycle token choice every time a cycle token is placed.

This constraint is considered to be symmetry-breaking since it avoids exploring branches where a "1" would be replaced by a "2" without any other changes.

We represent this using a Regular constraint which ensures the variables it is placed upon respect a given automaton. The automaton defined in Figure 4.1 ensures that, at any given state, we can freely place any cycle token already encountered. It also guarantees that only the next smallest cycle token can be used, excluding the ones already encountered, and using it transitions us to the next state.

$$\textsc{Regular}(\langle X_1, X_2, \ldots, X_n\rangle, \mathcal{A})$$

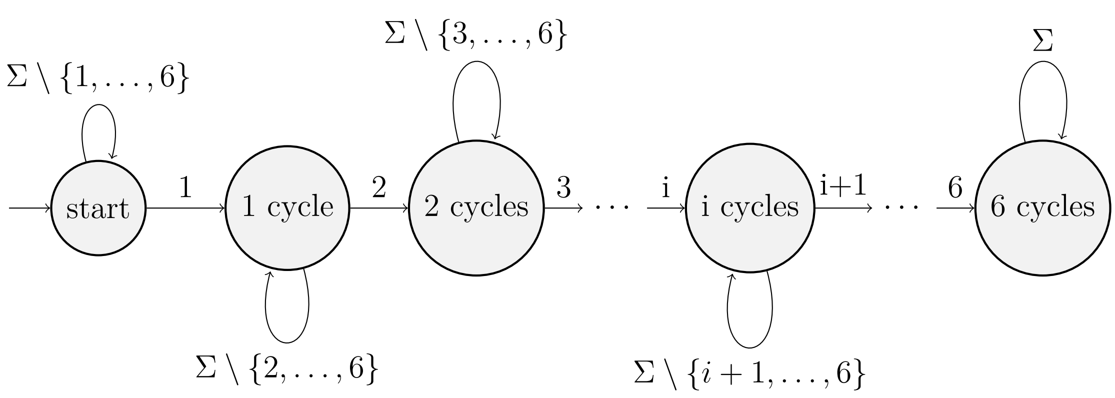

Figure 4.1 Automaton $\mathcal{A}$ which imposes ordinal order on cycle numbering. The starting state has no cycles that have been opened yet and subsequent states each contain one more opened cycle than the last. The Σ character represents all terminals in the grammar's alphabet. Every token, other than certain cycle tokens, lead back to the same state. Starting at the first state after the start state, cycle tokens that have already been seen can be placed freely.

4.2.3 Structural Constraints

This section will present two constraints targeting specific structures in generated molecules. These constraints will be used to get samples of varying difficulty during our experiments. These constraints will never be used on their own, instead always being used in tandem with the validity constraints from Section 4.2.2.

Cycle Count Constraint

This constraint forces our generated output to contain a certain number of cycles. Using SMILES notation, this is very simple to do. By previously placing the cycle numbering constraint, we guarantee that each cycle has its own number token used to identify it. This allows us to use an Among constraint, requiring the presence of the number token equivalent to the number of desired cycles, *e.g.* if we want 4 cycles, we require the presence of the token "4".

While this ensures we get 4 cycles, it actually ensures we get *at least* 4 cycles. To avoid getting more than 4, we have to place a second Among constraint that forbids the use of the next smallest cycle token. In our example where we have 4 cycles, we would forbid the use of the token "5".

Formally this is defined using the two following constraints where c is the number of desired cycles in the chain. It is important we ask for 2 and not 1, since the cycle parity constraint from Section 4.2.2 already restricts the number of appearances to either 0 or 2.

$$\textsc{Among}(\langle X_1, X_2, \ldots, X_{40}\rangle, \{c\}, 2)$$
$$\textsc{Among}(\langle X_1, X_2, \ldots, X_{40}\rangle, \{c+1\}, 0)$$

Branch Count Constraint

Similarly to the previous constraint, the branch count constraint requires a certain number of branches in the generated output. Since the grammar constraint from Section 4.2.2 ensures that any opened branch is closed, we can constrain the number of total branches by placing an Among constraint on either the opening or closing branch tokens. In the following definition, b is the desired number of branches.

$$\textsc{Among}(\langle X_1, X_2, \ldots, X_{40}\rangle, \{\text{“(”}\}, b)$$

4.2.4 Molecular Property Constraints

In this section, we answer our second research question: Can we use CP to model desirable molecular properties in SMILES molecules?

We previously talked about Lipinski's rule of five in Section 1.2.4. We will show how it is possible to describe each of these properties using constraints in our current model.

These constraints will never be used alone, they are always used with the constraints from Section 4.2.2.

Molecular Weight Constraint

The first property constraint is the molecular weight. Since the solver we use only allows for integers, we multiply all weight values by 10 to get more precision for this constraint.

Estimating the weight of the grammar's tokens. Since we are working on a one-dimensional representation, SMILES, we attempt to estimate the weight of the total molecule by estimating the weight of each token in the SMILES string. However, this isn't as simple as linking atoms to their atomic weight, since we have to account for the Hydrogen atoms that are potentially bonded but implicit in SMILES notation.

An intuitive solution to this is to assume that each atom token is making two bonds, one on its left and one on its right in the SMILES chain. This allows us to assume that each atom token is bonded to two less Hydrogen atoms than the number of bonds needed to complete its valence shell, *e.g.* Carbon, which needs to make 4 bonds to complete its valence shell, would have an assumed 2 bonds with Hydrogen atoms.

However, this sometimes results in an overestimation of the molecule's weight. To correct this, we associate a weight, which is sometimes negative, to non-atomic tokens.

Cycle tokens are an extra bond that our current weight model does not account for. Each extra bond is a bond that cannot be made with a Hydrogen atom. For that reason, we associate all our cycle tokens to the negative weight of a Hydrogen atom.

It would seem like branch tokens need special weights for the same reason. However, the opening branch token indicates an extra bond (*i.e.* a negative weight) while the closing branch token indicates a lacking bond that we assumed was present (*i.e.* a positive weight). Overall, these two tokens should cancel out. However, this is only true if the token to the left of the closing branch token was expected to make a bond on its right. If the last atom already has a full valence shell, *e.g.* "...F)" or "...=O)", it cannot bond with another on its right. In such cases, our assumption that the closing branch token "replaced" one of the atom's bonds is wrong and would result in a slightly higher weight than expected.

Bond tokens are also associated to negative weights. When we make a double bond, there are two fewer Hydrogen atoms than we assumed there would be in our atom weights. Similarly, a triple bond means there are four fewer Hydrogen atoms bonded to the two atoms around the bond token. Therefore, the double and triple bond tokens are, respectively, associated to a weight of -20 and -40.

We also had to adjust the weight of aromatic cycle tokens. As we explained earlier in Section 1.2.3, aromatic cycles are a specific type of cycle where single and double bonds alternate. This is common enough to justify a shorthand notation in SMILES. We can assume that these atoms are bonded to 3 other atoms, unlike the 2 bonds for non-aromatic atoms, *e.g.* the aromatic variant of Carbon would have 1 Hydrogen atom bonded to it instead of 2 and its associated weight would reflect this.

Gilles commented: Tu ne dis pas que tu as fait une régression linéaire. Ces poids sur les jetons ne sont quand même pas le "ground truth" mais une approximation statistique. Donc tu es quand même justifié d'avoir utilisé tes poids. Dis simplement que ces tests confirment que les poids que tu as dérivés manuellement donne une bonne approximation du poids réel.

@Gilles, I have a question: Ces tests sont avant la régression linéaire, c'était quand on regardait ce qui cause les erreurs et on a trouvé qu'en mettant un poids de 10 pour le "+", on obtient des meilleurs résultats

Finally, we did some testing to see the accuracy of our estimation. We used the open-source tool RDKit[1] to calculate the true weight. During our tests, we found that associating a positive weight to the "+" token improved the score. Atoms with a charge have a different number of Hydrogen atoms bonded to them.

With this we create a weight array (Table 4.2), $\mathcal{T}^w$, indexed by token IDs.

[1] https://www.rdkit.org/

Token	Human Weight	Linear Regression Weight
C	140	140
c	130	130
N	150	148
n	140	144
O	160	161
o	160	158
S	321	338
s	321	319
F	180	180
Cl	345	346
Br	789	793
I	1259	1259
=	-20	-13
#	-40	-35
+	10	11
-	10	-1
1	-10	-9
2	-10	-9
3	-10	-9
4	-10	-9
5	-10	-9
6	-10	-9
(	0	1
)	0	1
[	0	-4
]	0	-4
H	0	12
H2	0	18
H3	0	19
@	0	-3
/	0	-5
\	0	-9

Table 4.2 Estimated token to weight array $\mathcal{T}^w$. Any token that is not present in this weight map has a weight of 0. The middle column are the weights using our human intuition, the right column are the weights as predicted by the linear regression. While there are some differences, most weights are similar which confirms our intuition. However, we continue to use the human weights in our experiments.

Defining the constraint. To apply this constraint, we first create weight variables, W_i, that will represent the weight of their associated token variables, X_i.

We link the values of these variables using the ELEMENT constraint. It uses the weight array previously defined, $\mathcal{T}^w$ as a lookup table and ensures that W_i is the value associated to index X_i.

Once these variables are defined, we can constrain the sum, S^w, of the variable array, W, to our desired estimated value. To respect Lipinski's rule of five, we would limit the value to 500 Daltons (in our model, we would instead use 5000 since we multiply values by 10 for more significant numbers).

$$\textsc{Element}(\mathcal{T}^w, X_i, W_i)\ \forall\ i \mid 1 \leq i \leq n$$
$$\textsc{Sum}(\langle W_1, W_2, \ldots, W_n \rangle, S^w)$$
$$S^w \leq 500$$

How accurate is this estimation? As mentioned previously, we tested this process on the 2.2M molecules in the two datasets used thus far in our work. On average the error is 1.06% of the molecule's real weight. However, at its maximum, we find errors of 4.83%.

We include a graph of the distribution of relative error frequencies in Figure 4.2. Since it is an estimation, the relative error rate is acceptable and doesn't stop us from targeting a desirable region of the search space.

Can we improve this using a simple linear regression? By using a linear regression, we can see how close our intuition was and get a potential improvement to our current constraint.

We first convert all our molecules into frequency arrays, where each position contains the number of times the associated token shows up in the molecule. We can then use the python library `SKLearn` to do a linear regression and find weights for each token.

This leads to a good improvement on the average error, now of 0.23%, and a massive reduction in the maximum error, now of 2.80%. The results can be seen in Figure 4.3.

While the linear regression's weights do vary from our own, they are mostly similar as can be seen in Table 4.2. This confirms our intuition but goes to show that there is room for improvement.

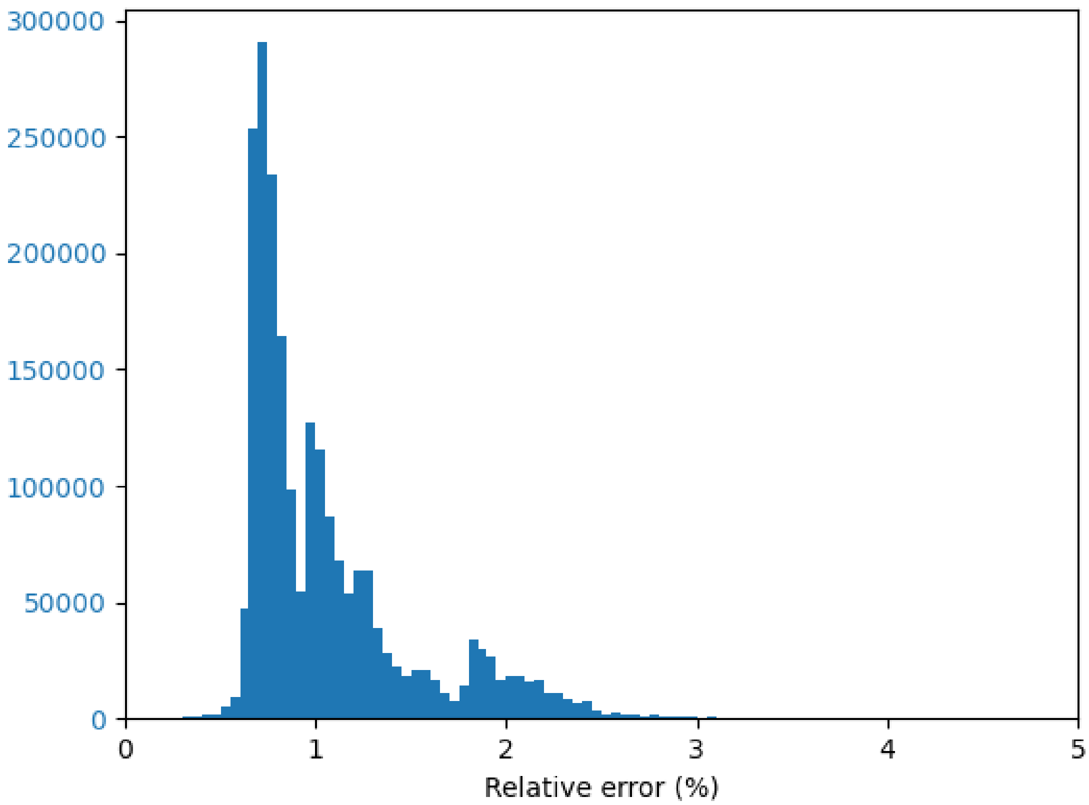

Figure 4.2 Relative error frequency when estimating the weight of molecules using human intuition. Most values are concentrated around 1%, but we see nearly 50k molecules with errors around 2%. We know the maximum error is 4.85%, but it is so infrequent that it does not show up in the graph.

Hydrogen-Bond Acceptors Constraint

This property is simple enough to represent in SMILES notation. As long as one of the atoms of interest (*i.e.* N, O, S) have a free electron pair, they are considered an acceptor. For these atoms to not have a free electron pair would require it to be used to make another bond and for its valence shell to be overloaded (*i.e.* making more bonds than what is expected). This is possible but not common, and so a good estimation of the number of Hydrogen-bond acceptors in a molecule is simply the number of relevant atoms, *i.e.* Nitrogen, Oxygen and Sulfur. The aromatic version of the atoms are included in the constraint.

While Fluorine is an electronegative, Lipinski's rule of five specifically excludes it from the list of potential Hydrogen-bond acceptors [17].

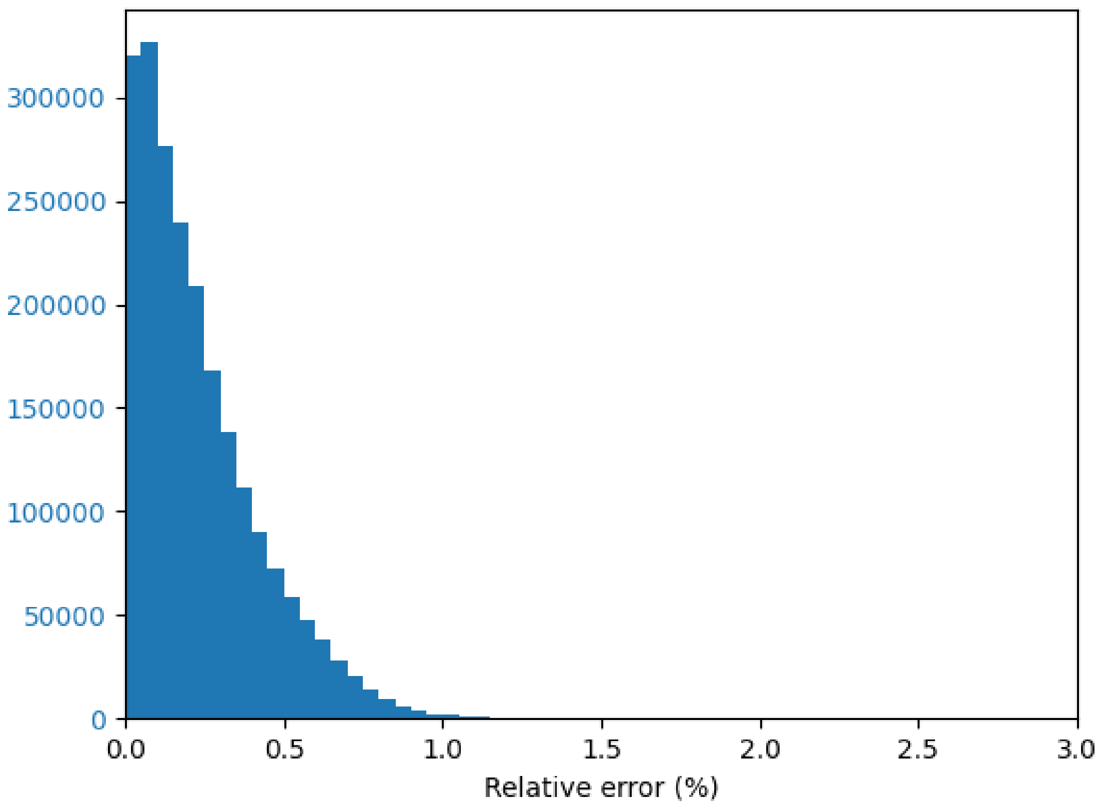

Figure 4.3 Relative error frequency when estimating the weight of molecules using a linear regression.

We note the number of wanted acceptors as N_a in the formal definition below.

$$\textsc{Among}(\langle X_1, X_2, \ldots, X_n\rangle, \{“N”, “O”, “S”, “n”, “o”, “s”\}, N_a)$$
$$N_a \leq 10$$

Hydrogen-Bond Donors Constraint

This property, while similar to the previous one, requires changes to our grammar in order to be represented correctly. The full changes can be seen in Appendix B.

Since our grammar already accounts for the number of bonds that atoms are making, it seemed natural to change certain productions to determine which atoms were donors and which ones weren't. However, this implies the need for new atom tokens to differentiate between the donor and non-donor version of the same atom. We add:

- "N_D" as the donor version of "N"
- "O_D" as the donor version of "O"
- "S_D" as the donor version of "S"

Since the atoms in question have to be bonded to a Hydrogen atom to be Hydrogen-bond donors, we suppose that they cannot be donors if they are a part of an aromatic cycle. For that reason, only the non-aromatic version of the atoms are included in the constraint.

Once we have modified our grammar, it is simply a matter of limiting how many of the donor tokens appear in our molecule. We note the number of wanted donors as N_d as seen below.

$$\begin{aligned}&\textsc{Among}(\langle X_1, X_2, \ldots, X_n\rangle, \{``T", ``X", ``R"\}, N_d)\\&N_d \leq 5\end{aligned}$$

LogP Constraint

To model the *logP* value using only SMILES notation, another of Lipinski's rules, human ingenuity is not enough.

Ridge Regression. Basing ourselves on the work done previously by Vidal et al. [18], we use a linear regression to estimate the partial contribution, positive or negative, of sequences of four tokens (4-*grams*) on the *logP* score. In their work, they used a partial least squares regression to estimate the *logP* value of different molecules. Their results indicate that their model could accurately predict *logP* values. Similarly, we compute the partial contribution of every possible 4-gram in the datasets by using a ridge regression (a linear regression where we add a small value to the input to avoid linearly dependant inputs).

The accuracy of this model is very variable. We did a relative error analysis as well as an absolute error analysis as can be seen in Table 4.3.

Absolute Error Analysis. The first thing to note is that the average error sits at 0.1235, an acceptable value when we are targeting a *logP* score under 5. However, the maximum error is much bigger, at 2.2231. To get a clearer message, we calculate the median error (*i.e.* the second quartile) and find that it is lower than our average, at 0.0861. We decide to exclude outliers in our data by using the IQR filtering method. This filters out 4.56% of our data as outliers and gives us our new average: 0.1063.

Relative Error Analysis. Relative error allows us to get a better picture by comparing the error to the targeted value instead of keeping an absolute scale. With an average error of 12.56%, our method doesn't seem too accurate. However, our maximum error of 108′851.38% seems absurd. Upon further inspection, we found that we were getting these absurdly high relative errors on molecules with very small *logP* scores (*i.e.* smaller than 0.001). Since the relative error has the true value as the denominator, this makes the relative error for this sample grow disproportionately. Once again, we looked at the median value, which is less sensitive to outliers and find that it is four times smaller than our current average at 3.68%. To avoid falsifying our average with outliers, we filter our data using IQR filtering, filtering out 7.86% of our data. This allows us to get a more reliable average of 4.54% relative error, a very acceptable error rate for our estimations.

Calculated Value	Absolute	Relative
Total Average Error	0.1235	12.56%
Total Maximum Error	2.2231	108′851.38%
Q1	0.0388	1.56%
Q2	0.0861	3.68%
Q3	0.1715	8.02%
Inlier Data Coverage	95.44%	92.14%
Inlier Average Error	0.1063	4.54%

Table 4.3 Error analysis on the *logP* estimation constraint. The first two rows are the average and maximum error on all data points. The next three rows are the quartile values. We include a row to detail what percentage of the data points are still included after excluding outliers. The final row is the new average, excluding outliers, this gives us a better representation of our method's efficiency.

Table Implementation. We can then create a table $\mathcal{T}^p$ which links every possible 4-gram to its estimated contribution. In the case where a 4-gram has no or a very small partial contribution, its weight is set to 0, thus having no effect on the final estimation of the *logP* value.

Our model then uses a Table constraint as defined below:

$$\textsc{Table}(\langle X_i, \ldots, X_{i+3}, P_i\rangle, \mathcal{T}^p) \; \forall \; i \mid 1 \leq i \leq n-3$$
$$\textsc{Sum}(\langle P_1, P_2, \ldots, P_{n-3}\rangle, S^p)$$
$$S^p \leq 5$$

However, this table can potentially be quite large (the number of terminal tokens elevated to the power 4), we limit its size through the use of a wildcard token ($\star$) and the ShortTable constraint [19].

Short Table Implementation. Since a $\star$ token can represent any other token, we can use it to map large numbers of zero-weight 4-grams to the appropriate weight. To identify the 4-grams to which we can apply these wildcards, we iterate on each position and on all possible tokens for that position, if no important 4-grams (that have a non-zero weight) contain the prefix, we associate that prefix followed by wildcard tokens to a weight of 0. This wildcard allows us to reduce the table down to 39305 4-grams that have a non-zero weight, which is 3% of all the possible 4-grams. The total table, including zero weight 4-grams, has $168'731$ rows, closer to 12.6% of the size a normal Table constraint would have needed.

The constraint definition doesn't change much, we replace the Table constraint by a ShortTable constraint and use the new table, $\mathcal{T}^{p\star}$:

$$\textsc{ShortTable}(\langle X_i, \ldots, X_{i+3}, P_i\rangle, \mathcal{T}^{p\star}) \; \forall \; i \mid 1 \leq i \leq n-3$$
$$\textsc{Sum}(\langle P_1, P_2, \ldots, P_{n-3}\rangle, S^p)$$
$$S^p \leq 5$$

This representation results in a number of constraints which grows linearly with the number of token variables. However, there is a way to represent this property by using a single CostRegular constraint applied on the entire variable array X.

Cost Regular Implementation To implement this constraint using a CostRegular, we have to remap our $\mathcal{T}^{p\star}$ table into an automaton $\mathcal{A}^p$.

This automaton contains a state for each non-zero weight 4-gram as well as states for every 3-gram, 2-gram and 1-gram necessary to get to the non-zero weight 4-grams. The transition to a 4-gram state is associated to the partial contribution of that 4-gram. All other state transitions have a weight of 0 and all states in the automaton are considered valid final states. See Algorithm 1 to see exactly how we create the transition and weight table for our automaton.

The final automaton has 41′741 states, a considerable reduction in size compared to the previous ShortTable constraint.

$$\textsc{CostRegular}(\langle X_1, X_2, \ldots, X_n\rangle, \mathcal{A}^p, \mathcal{W}_{\mathcal{A}^p}).$$

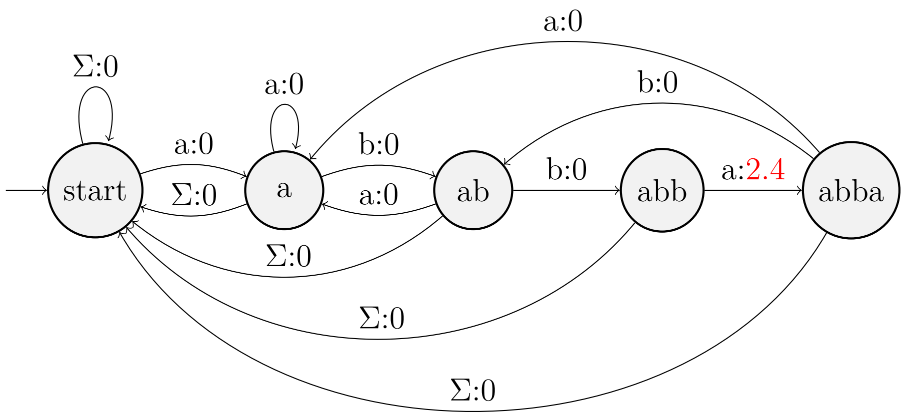

Figure 4.4 Simplified example of automaton $\mathcal{A}^p$, associating a weight to a sequence during generation. This example associates a weight of 2.4 to the 4-gram **`abba`**. For clarity and simplicity, we limit ourselves to one sequence as the graph quickly becomes very connected. The only transition that has an associated weight is the one that completes the 4-gram. Each state has a transition back to the start state in the case where it receives a token that is in the alphabet (Σ) but does not have its own state. For all transitions that have Σ associated to them, we assume any tokens that are in another transition are excluded from Σ.

4.3 Experiments

In the previous section, we've answered our first two research questions:

1. Can we use CP to model valid molecules in a one-dimensional encoding?
2. Can we use CP to model desirable molecular properties in SMILES molecules?

In this section, our experiments will serve to answer the third of our questions: Can BP be

Algorithm 1: regularAutomatonCreation(N, w, g)

Input:
A list of non-zero weight 4-grams: N;
A dictionary mapping each 4-gram to its weight: w;
A list of the tokens in the grammar's alphabet: g
Output:
A transition matrix: $\mathcal{T}_{A^p}$;
A weight matrix: $\mathcal{W}_{A^p}$;

```
// Define a state set that starts with the empty state
S ← {“”}
// Find all relevant states from our 4-grams
foreach ngram ∈ N do
    state ← “”
    // Add each partial sequence of the 4-gram to the state set
    foreach token ∈ ngram do
        state ← state + token
        S ← S ∪ state
// Fill the transition and weight matrix
foreach state ∈ S do
    foreach token ∈ g do
        // Define the next state as the current state’s last three
           tokens plus the given token
        state′ ← state[1 :] + token
        // Remove the first token from the next state until we find a
           valid transition state.  This will always default to the
           start state “” if nothing is found.
        while state′ ∉ S do
            state′ ← state′[1 :]
        T_{A^p}[state][token] ← state′
        if len(state′) = 4 then
            W_{A^p}[state][t] ← w[state]
return (T_{A^p}, W_{A^p})
```

used to better guide a solver towards a solution? We also go over the necessary steps to run the experiments as well as the specific testing conditions.

Do this: All subject to change if we rerun our tests. To be determined

These experiments were run on an AMD Rome 7532 processor (2.4GHz, 256M cache L3) with 1 GB of RAM and using a 30-minute timeout.

For our tests, we will place additional structural constraints on the generated output. We require the presence of a certain number of branches and cycles in the respective range of 2-4 and 1-3. We add these constraints to evaluate the performance of different branching heuristics on problems of varying difficulty.

4.3.1 Chomsky Normal Form

The solver we use, miniCPBP[2], has an implementation of the grammar constraint that requires a grammar in Chomsky Normal Form.

Do this: Potentially insert what is in the intro here

We automated the process of converting the CFG into the right form. This allows us to keep working on the more readable CFG format.

After converting the original grammar, the number of nonterminals and productions, respectively, increase to 169 and 411. Meanwhile, the final grammar grows to 640 nonterminals and 1996 productions.

The complexity of the base propagation algorithm is cubic in regards to the number of variables as well as linear according to the number of productions [20]. The number of variables in our model does not change with the size of the grammar, however we do have nearly five times as many productions. However, seeing as we are using Belief Augmented Constraint Programming, this requires an additional step which is also cubic in relation to the number of variables, linear in relation to the number of productions and linear in relation to the number of nonterminals. This will slow down the total time required for our algorithms to run.

[2]`https://github.com/PesantGilles/MiniCPBP`

4.4 Results

	domWdeg/random		maxMarginalStrength/DFS		maxMarginalStrength/LDS	
instance	time(s)	fails	time(s)	fails	time(s)	fails
c1b2	20.2	103	8.9	0	8.9	0
c1b3	14.0	65	12.0	0	11.7	0
c1b4	61.7	484	12.2	0	12.6	0
c2b2	26.3	105	–	–	16.7	3
c2b3	37.4	253	16.0	0	16.0	0
c2b4	245.3	2083	17.9	12	17.5	6
c3b2	131.2	1389	–	–	32.0	14
c3b3	40.2	106	–	–	–	–
c3b4	101.5	1040	–	–	498.8	247

Table 4.4 Comparing branching heuristics on some structurally-constrained molecule generation instances.

Do this: Move this to when we compare LogP estimation constraint

	Initial Setup (s)	**Solve Time (s)**	Failures
ShortTable with BP	10.707	1.168	13
ShortTable without BP	17.670	24.116	111
Regular	31.917	20.944	4

Table 4.5 Performance comparison between different implementations of the *logP* estimation constraint. The addition of BP clearly reduces the number of failures before a solution is found.

Unfortunately during our testing, the Belief Propagation component of the ShortTable constraint resulted in strange molecules being generated. It would generate as small of a molecule as possible and seemed to lack the variety that previous tests had.

When we disable the BP component of this constraint, we find more varied results that seem more natural based on what we have seen thus far in molecule datasets. However, BP does significantly decrease the solving time as seen in Table 4.5.

Do this: About the regular constraint as opposed to the shortTable

However it has a slower solving time, as can be seen in Table 4.5.

CHAPTER 5 COMBINING CP WITH NLP TO IMPROVE GENERATION

This chapter details how we combine our previous CP with a chosen LLM to try and improve the realism of our generated molecules. By combining a model trained on real drug-like molecules and our CP model, we believe it is possible to get molecules that are informed by what is in current use while still maintaining the guaranteed validity and property targeting.

The first section will present the combined architecture before going in depth on each part. We will then go over the experiments we did and their results in the next two chapters.

5.1 Architecture

To combine the two models, we use a combination at inference time (*i.e.* during generation). Our CPBP model takes the LLM's prediction probabilities over the next token as an input and updates them using BP. The combined model then samples the next token from this new distribution and sends this new string as an input to the LLM.

This combined model, called Generative AI using Belief-Augmented Constraints (GeAI-BlAnC), can be seen in Figure 5.1. We start with the start token and, after each iteration, sample a new token that is informed by both the LLM and our CPBP model.

5.1.1 LLM

We chose to use the GPT model, GPT2-ZINC480M-87M[1] (henceforth referred to simply as GPT). It has 87M parameters and was trined on 480M molecules from the ZINC database[2]. This is the same database that we have partial access to (we can access 250k molecules). It is a transformer model trained to generate molecules in the standard string representation SMILES (Fig. 5.2 gives an example).

The use of a token-by-token generation model allows us to change the probability distribution over the next token before sampling and is necessary for this combined architecture to work.

However, changing this distribution might overpower the LLM's message with the CPBP model's message. This will be studied later in our experiments through the use of the perplexity metric (which will be explained in Section 5.2).

---

[1] `https://huggingface.co/entropy/gpt2_zinc_87m`
[2] `https://zinc.docking.org/`

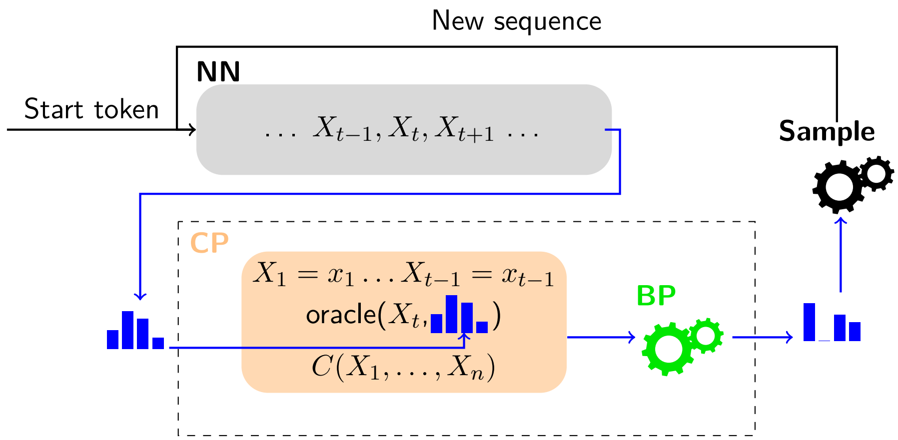

Figure 5.1 Combined CP and token-by-token generator architecture. In our case, the token-by-token generator is a LLM, but any other can be used as long as it was trained on SMILES strings. The NN outputs a probability distribution over the next token, X_t in our example. This distribution is given as the outside belief of the Oracle constraint. After a few iterations of BP, we get our new distribution and sample for the next token. This process starts by inputting the start token to the NN.

5.1.2 Oracle Constraint

The Oracle constraint is a unary constraint defined as follows: Oracle(X, p). X is a finite-domain variable and p is a fixed probability mass function over the domain of X. In our case the variable is X_t, representing the token at step t and p is the NN's probabilities for the current token x_t. Uncharacteristically, this constraint does not enforce a relation but only associates a probability to each domain value. Its sole purpose is to contribute messages to variable X_t during BP in the same way as the other constraints in the CP model. Without the Oracle constraint, the resulting marginals would only take into account the satisfaction of $C(X_1, \ldots, X_n)$ (*i.e.* the satisfaction of the constraints over the variables) and not what was learned from the dataset. Therefore, the Oracle constraint is our way of integrating the NN's knowledge into the process of CPBP.

To balance this integration, we can associate a weight to our constraints, in this case we would adjust the weight of the Oracle constraint (as will be shown later in our experiments). This weight affects the marginals sent by constraints during BP (and not the filtering nor the hardness of the constraint). Given a positive weight w (the default value being 1) each marginal $p_X(v)$ for a value v in the domain of variable X is raised to the power of that weight and normalized, yielding new marginal $(p_X(v))^w / \sum_{d \in D(X)} (p_X(d))^w$. As a result, a weight $w > 1$ accentuates the disparities between marginals while $0 < w < 1$ lessens them and makes them more uniform. We will use such a weight on the Oracle constraint in order

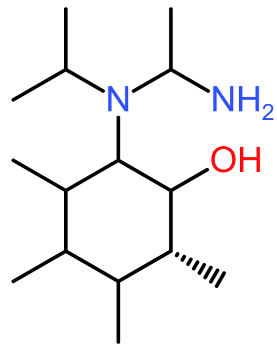

Figure 5.2 "C[C@H]1C(O)C(N(C(N)C)C(C)C)C(C)C(C)C1(C)", a molecule of weight 256.3 Da and PPL=1.8487 generated by GeAI-BLAnC.

to control its importance on the resulting distribution.

5.1.3 Communication

Since the two models are in different programming languages, GPT is in Python while MiniCPBP (the solver from Section 4.3) is in java, we had to set up a way for them to communicate. We could do this by integrating one model into the other language, however we chose to use a minimalistic HyperText Transfer Protocol (HTTP) server approach.

By making the GPT model a server with its own interface, we can easily change the model being used, and we can allocate more resources to it if needed. The GPT model can compute the next token's probability distribution fast enough that it can keep up with multiple client instances making requests.

The client instances in question are the CPBP model. It makes an asynchronous HTTP request to the server with the current molecule in the request. The server decodes the molecule and calculates the next token's probabilities.

An issue we encountered is that our CPBP model's language is made up of relevant single tokens in the SMILES alphabet. However, the GPT model's language is made up of n-grams, which includes single tokens, but also includes n-grams made up of multiple tokens. If we were to return those probabilities as they are, our client instance would be unable to input them into the Oracle constraint. We first convert the result into something our client can understand.

To do this, we pass over every n-gram and associate its probability to the first token that appears in it. If the first token already has an associated weight, we sum the two probabilities into a new one. Since we get the probability of every n-gram in the GPT model, every token from our CPBP model's language is guaranteed to have an associated probability, though that weight may be 0 if every n-gram the token leads has a null-probability. This conversion also allows us to remove any unwanted tokens that our CPBP model does not support (*e.g.* higher cycle number tokens). Finally, during this process, we also change the end-of-sequence token, "</s>", to our padding token, "_", since they functionally represent the same thing. Once we place a padding token, the only thing that can follow are more padding tokens.

With this, we have a functioning HTTP server that takes the current molecule as input and returns the probability distribution over the next token's value in our CPBP model's language.

5.2 Experiments

This section will detail the experimental context and any steps required to reproduce our results.

With our experiments, we wish to show two things. First, we want to further our findings from Section 4.3 and evaluate the advantages of BP when looking for a solution as per our third research question. We also aim to show that our approach more consistently generates sequences exhibiting the desired structure while still reflecting what the NN has learned from the training corpus.

GPT model's settings. As mentioned prior, we use the GPT2-ZINC480M-87M[3] model that was trained on 480M molecules from the ZINC dataset.

During generation, we kept the model's default configuration with the following exceptions: we limited the generation to one new token at a time as per Fig. 5.1, we set the model's temperature to 1.5 which gives more varied results as reported by the model's authors, we decreased the maximum length of the molecule to fit our target length, and we disabled the early stopping parameter.

Changes to the CP solver. During most of our tests, we choose to disable the backtracking ability of our CP solver, MiniCPBP[4]. We do this to allow a more faithful comparison between the models. We keep one test with the backtracking active to evaluate the differences.

We will detail which search heuristics were used during generation later on.

[3] `https://huggingface.co/entropy/gpt2_zinc_87m`
[4] `https://github.com/PesantGilles/MiniCPBP`

Experimental conditions. We attempt to generate 100 molecules. Our experiments were run using an 8-core processor with a core speed of 4.20 GHz and 64GB of Random Access Memory (RAM). All our code and data for these tests are available[5]. As an initial input to the GPT model, we use the start-of-sequence token, "<s>".

5.2.1 Chosen constraints

For these tests, we did not use all the constraints described previously in our CP model. We only need a subset of the constraints for the goals set earlier.

All constraints relating to the validity of the SMILES string are necessary to ensure that any generated final result respects SMILES notation. These are the constraints described in Section 4.2.2.

We do not keep any of the structural constraints described in Section 4.2.3. While they would add complexity to the problem, they do not require long-term structure the way the property constraints do.

While all property constraints from Section 4.2.4 introduce some form of long-term structure to the molecule, only one is required to evaluate it. We chose the molecular weight constraint for three reasons: most tokens in the chain will have an effect on the weight, the GPT model was not trained on this property, and it is more accurate than the *logP* constraint.

Overall, we apply the following constraints. Note that the targeted molecular weight is now between 200 and 275 Daltons. This weight is only achieved by 20.48% of the 40-token molecules observed in the datasets.

Validity Constraints:

$$\texttt{grammar}(\langle X_1, X_2, \ldots, X_{40}\rangle, \mathcal{G}_{\texttt{SMILES}})$$

$$\textsc{Among}(\langle X_1, X_2, \ldots, X_{40}\rangle, \{j\}, \{0, 2\})\ \forall\ j \mid 1 \leq j \leq 6$$

$$\textsc{Regular}(\langle X_1, X_2, \ldots, X_n\rangle, \mathcal{A})$$

Molecular Weight Constraint:

$$\textsc{Element}(\mathcal{T}^w, X_i, W_i)\ \forall\ i \mid 1 \leq i \leq n$$

$$\textsc{Sum}(\langle W_1, W_2, \ldots, W_n\rangle, S^w)$$

$$200 \leq S^w \leq 275$$

[5]https://github.com/cravethedave/MiniCPBP/tree/ijcai-2025

5.2.2 Evaluation metrics

To evaluate our different models, we use three metrics: time, success rate and perplexity.

The time is self-explanatory, we measure the average time to generate a molecule over all successful molecules generated. This means that unsuccessful runs aren't accounted for in this average. We also set a time limit on the CPBP model running with backtracking in case the search takes too long. The time limit was set to 10 minutes.

As for the success rate, it is evaluated as the number of successfully generated molecules out of the 100 attempts. However, we categorize a molecule as "successfully generated" if the generated molecule is valid and respects all constraints that we defined.

Finally, perplexity [21] is a common metric in NLP:

$$\mathrm{PPL}(x_1, \ldots, x_n) = \exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log p(x_t|x_1, \ldots, x_{t-1})\right)$$

The higher the perplexity, the less likely it would be generated by the neural model and the more surprising it is with respect to the training set. This is particularly interesting for our CP model as it may disagree with the probability distribution it receives from GPT and modify it. This could increase the perplexity score for the current token if the chosen value is one that GPT would be less likely to choose. We also consider perplexity for individual tokens, $1/p(x_t|x_1, \ldots, x_{t-1})$, in order to track its behavior across the whole sequence.

5.2.3 Tested model combinations

CPBP with backtrack

This is the default solving method for our CPBP solver. It should have the highest success rate as it can backtrack to fix its mistakes while the other methods will be limited to token-by-token generation. Its perplexity score should be high, but it'll be interesting to see if it differs from the CP model with no backtracking.

To ensure the results are as close to the other models, allowing for a better comparison, we use a lexicographic variable choice (*i.e.* in order from left to right) and a biased wheel selection for the value (*i.e.* weighted random or roulette wheel selection).

All further models will generate the sequence token-by-token.

CPBP no backtrack

By removing the backtracking from our model, it may end up making mistakes. However, this serves as a better comparison to the GPT models as we cannot backtrack to fix our mistakes, which is how the CP model will be used in the combined architecture.

GPT

The first GPT model is run without any input from the CP side of things. This serves as our baseline for the perplexity score. Once we add CP to the setup, we'll be able to see how it affects the success rate of our model.

GPT + CP

By adding the CP component to our GPT model, we expect the success rate to rise in exchange for an increase in perplexity. This model does not use BP and thus the integration between the two models is slightly altered. Instead of modifying the GPT model's probabilities, we simply use CP to eliminate values that would breach one of our constraints and then normalize the probabilities of the possible values that are left. We then proceed normally, using a biased wheel selection.

GPT + CPBP

Finally, this model is the one presented in Figure 5.1. It utilizes BP to modify the probabilities received by the GPT while also getting rid of values breaching contraints.

As mentioned in Section 5.1.2, we can modify the weight of the ORACLE constraint to increase or decrease its influence on the generated sequence. We test this model using a baseline constraint weight of 1, as well as 0.5 and 1.5 to observe the effects it has on the perplexity and the success rate. The time should not be affected by this change.

5.3 Results

This section will go over our results for the different model combinations that we described previously.

CPBP with and without backtrack. As we expected, the backtracking model has the highest success rate at the cost of having a very high perplexity and run time. It takes twice

method	success(%)↑	PPL↓	time(s)↓
CPBP (backtrack)	100	1236.71	125.4
CPBP (no backtrack)	59	503.48	62.4
GPT	7	5.30	1.2
GPT+CP	18	13.50	25.2
GPT+CPBP ($w = 1$)	92	8.35	67.2
GPT+CPBP $w = 0.5$	82	17.30	70.8
GPT+CPBP $w = 1.5$	72	8.14	63.6

Table 5.1 Success rate, average perplexity, and average runtime over 100 attempts to generate weight-constrained 40-token molecules. The arrows near the column heads indicate what the goal is for this column (*i.e.* minimize or maximize).

as long to solve when we use backtracking, but we achieve a perfect success rate. The success rate of the no-backtracking model is much lower than we anticipated. By eliminating values that lead to unsolvable sequences, we thought the model would still perform with very few errors. However, BP is a heuristic, and we are using a biased wheel selection, meaning we might make multiple non-ideal choices which lead to an unsolvable sequence.

What's interesting to note is that backtracking seems to increase the perplexity. We believe this is due to a type of "survivor bias". When we have backtracking enabled, a non-ideal decision, which would have lead to an unsolvable sequence in the no-backtrack model, can still be solved using backtracking and results in a molecule that is atypical and has a high perplexity score. In other words, the perplexity is higher because molecules that have a high perplexity are harder to solve and end in failure when we don't have backtracking.

GPT model alone. Contrary to the previous two models, this one's success rate is the lowest, but it also achieves the best perplexity score in the least time. Since the model is used to calculate perplexity, it holds that molecules generated purely with this model would have the lowest score.

A reminder that a successful molecule is one that respects validity *and* the molecular weight. The model generates more valid molecules, but as it was not trained to target the specified molecular weight range, it does not perform well in terms of success.

GPT with an added layer of CP. Just by adding a layer of CP at inference time, we double our success rate. Unfortunately, as we do a biased wheel selection using the GPT model's probabilities, we still end up making mistakes. This model's increased success rate does show that CP filters out some wrong choices, however there is a significant increase in both time and perplexity.

GPT + CPBP, the full combined architecture. This model is the final one we proposed, a combined architecture that utilizes both the learned information from a GPT model and the BP of our CP model.

We ran tests with three different Oracle constraint weights: 0.5, 1 and 1.5. The success rate of the baseline model (weight of 1) is the highest of the three. As was expected, by increasing the weight of the constraint, we decrease the perplexity as we are giving more weight to the GPT model's message (and inversely when we decrease the weight of the constraint). However, what was unexpected is the decrease in success rate regardless of which way we shift the weight. Intuitively, one would expect the success rate to increase as we decrease the Oracle constraint's weight, giving priority to the hard constraints. But, as we can see, it appears that the GPT model contributes to the validity of the model. When we decrease the constraint's weight, we see a clear increase in perplexity: giving more weight to the constraint model over the GPT model would have that effect.

Overall, this final model performs incredibly well in terms of success rate, firmly beating GPT and CPBP without backtracking. It takes more time, roughly one minute per molecule, far more than GPT. We can see that a large part of that time comes from the added BP calculations (note the difference between GPT+CP and GPT+CPBP). We will discuss ways to mitigate this in our future work. The perplexity score stays pretty low even after adding the CPBP component, far lower than CPBP on its own and not too far above the GPT model alone.

Something very interesting is how the addition of BP lowers the perplexity. We expected the opposite to happen, where adding BP would change the GPT model's probabilities during sampling and lead to a higher perplexity. This could be explained by high-cost decisions the no BP model has to make. Since BP guides the search towards probable solutions, we believe the no BP model ends up in bottlenecks where it has to make multiple high-cost decisions to maintain solvability. Meanwhile, the BP model anticipates these problems and guides the search towards a more likely solution, making fewer high-cost decisions earlier to avoid multiple high-cost ones later. This can be seen in Figure 5.3, where the model with no BP starts off with very low perplexity choices and slowly gets worse. Meanwhile, the BP variants make a high-cost decision early, allowing future decisions to remain low.

@Gilles, I have a question: Que pensez-vous de cette explication?

The combined architecture successfully targets the molecular weight desired, even though

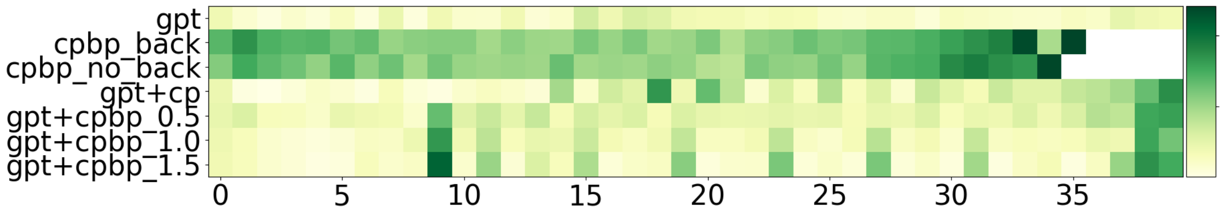

Figure 5.3 Average perplexity indexed by token (darker is higher).

the base GPT model was not trained to do so, while still maintaining a low perplexity. Admittedly, the time is high, but we believe this can be improved and does not prevent an answer to our fourth research question: How can we combine a CP model with a NLP model to improve the realism of generated sequences and is it an effective method? We have shown how to combine the two models with the help of the Oracle constraint and the results indicate that it is effective at targeting the specified constraints without overpowering the information learned by the GPT model.

CHAPTER 6 CONCLUSION

Texte / Text.

6.1 Synthèse des travaux / Summary of Works

Texte / Text.

6.2 Limitations de la solution proposée / Limitations

6.3 Améliorations futures / Future Research

Texte / Text.

REFERENCES

[1] P. et al., "Estimation of the size of drug-like chemical space based on gdb-17 data," *Journal of Computer-Aided Molecular Design*, 2013.

[2] Y. Du, T. Fu, J. Sun, and S. Liu, "Molgensurvey: A systematic survey in machine learning models for molecule design," 2022.

[3] D. Deutsch, S. Upadhyay, and D. Roth, "A general-purpose algorithm for constrained sequential inference," in *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, 2019, pp. 482–492.

[4] J. Y. Lee, S. V. Mehta, M. Wick, J.-B. Tristan, and J. Carbonell, "Gradient-based inference for networks with output constraints," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4147–4154.

[5] W. Commons, "Smiles.png," online, accessed July 12, 2023. [Online]. Available: https://commons.wikimedia.org/wiki/File:SMILES.png

[6] S. R. Heller, A. McNaught, I. Pletnev, S. Stein, and D. Tchekhovskoi, "InChI, the IUPAC International Chemical Identifier," *Journal of Cheminformatics*, vol. 7, no. 1, p. 23, Dec. 2015. [Online]. Available: https://jcheminf.biomedcentral.com/articles/10.1186/s13321-015-0068-4

[7] D. Weininger, "SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules," *Journal of Chemical Information and Computer Sciences*, vol. 28, no. 1, pp. 31–36, Feb. 1988. [Online]. Available: https://pubs.acs.org/doi/abs/10.1021/ci00057a005

[8] N. O'Boyle and A. Dalke, "DeepSMILES: An Adaptation of SMILES for Use in Machine-Learning of Chemical Structures," Sep. 2018. [Online]. Available: https://chemrxiv.org/engage/chemrxiv/article-details/60c73ed6567dfe7e5fec388d

[9] M. Krenn, Q. Ai, S. Barthel, N. Carson, A. Frei, N. C. Frey, P. Friederich, T. Gaudin, A. A. Gayle, K. M. Jablonka, R. F. Lameiro, D. Lemm, A. Lo, S. M. Moosavi, J. M. Nápoles-Duarte, A. Nigam, R. Pollice, K. Rajan, U. Schatzschneider, P. Schwaller, M. Skreta, B. Smit, F. Strieth-Kalthoff, C. Sun, G. Tom, G. Falk Von Rudorff, A. Wang, A. D. White, A. Young, R. Yu, and A. Aspuru-Guzik, "SELFIES and the future

of molecular string representations," *Patterns*, vol. 3, no. 10, p. 100588, Oct. 2022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2666389922002069

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[12] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[13] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[14] E. Kraev, "Grammars and reinforcement learning for molecule optimization," 2018.

[15] P. et al., "Molecular sets (moses): A benchmarking platform for molecular generation models," *Frontiers in Pharmacology*, vol. 11, 2020, iSSN: 1663-9812. [Online]. Available: https://doi.org/10.3389/fphar.2020.565644

[16] T. Akhmetshin, A. I. Lin, D. Mazitov, E. Ziaikin, T. Madzhidov, and A. Varnek, "ZINC 250K data sets," 12 2021. [Online]. Available: https://figshare.com/articles/dataset/ZINC_250K_data_sets/17122427

[17] C. A. Lipinski, F. Lombardo, B. W. Dominy, and P. J. Feeney, "Experimental and computational approaches to estimate solubility and permeability in drug discovery and development settings1pii of original article: S0169-409x(96)00423-1. the article was originally published in advanced drug delivery reviews 23 (1997) 3–25.1," *Advanced Drug Delivery Reviews*, vol. 46, no. 1, pp. 3–26, 2001, special issue dedicated to Dr. Eric Tomlinson, Advanced Drug Delivery Reviews, A Selection of the Most Highly Cited Articles, 1991-1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169409X00001290

[18] D. Vidal, M. Thormann, and M. Pons, "LINGO, an Efficient Holographic Text Based Method To Calculate Biophysical Properties and Intermolecular Similarities," *Journal of Chemical Information and Modeling*, vol. 45, no. 2, pp. 386–393, Mar. 2005. [Online]. Available: https://pubs.acs.org/doi/10.1021/ci0496797

[19] H. Verhaeghe, C. Lecoutre, and P. Schaus, “Extending compact-table to negative and short tables,” in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 3951–3957. [Online]. Available: https://doi.org/10.1609/aaai.v31i1.11127

[20] C.-G. Quimper and T. Walsh, “Global Grammar Constraints,” in *Principles and Practice of Constraint Programming - CP 2006*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and F. Benhamou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4204, pp. 751–755, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/11889205_64

[21] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, “Perplexity—a measure of the difficulty of speech recognition tasks,” *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 08 2005. [Online]. Available: https://doi.org/10.1121/1.2016299

APPENDIX A GRAMMAR FOR VALIDITY

Grammar used to model SMILES validity in Appendix A.

empty_smiles	→	smiles
empty_smiles	→	smiles void
void	→	“_” void
void	→	“_”
smiles	→	simple_bond
smiles	→	atom_valence_1 simple_bond
smiles	→	atom_valence_2 double_bond
smiles	→	atom_valence_3 triple_bond
atom_valence_1	→	“F”
atom_valence_1	→	“Cl”
atom_valence_1	→	“Br”
atom_valence_1	→	“I”
atom_valence_1	→	“[” “O” “-” “]”
atom_valence_1	→	“[” “N” hydrogen_3 “+” “]”
atom_valence_2	→	“O”
atom_valence_2	→	“S”
atom_valence_3	→	“N”
atom_valence_3	→	“[” “C” “@” hydrogen_1 “]”
atom_valence_3	→	“[” “C” “@” “@” hydrogen_1 “]”
atom_valence_3	→	“[” “N” hydrogen_1 “+” “]”
atom_valence_4	→	“C”
atom_valence_4	→	“[” “C” “@” “]”
atom_valence_4	→	“[” “C” “@” “@” “]”
atom_valence_4	→	“[” “N” “+” “]”
hydrogen_1	→	“H”
hydrogen_3	→	“H3”
simple_bond	→	valence_1
simple_bond	→	valence_2 simple_bond
simple_bond	→	valence_3 double_bond
simple_bond	→	valence_4 triple_bond
simple_bond	→	valence_2 slash valence_3 “=” valence_3 slash valence_2
slash	→	“/”
slash	→	“\”
valence_1	→	atom_valence_1
valence_1	→	valence_2
valence_1	→	valence_2 “(” simple_bond “)”
valence_1	→	valence_3 “(” double_bond “)”
valence_1	→	valence_4 “(” triple_bond “)”

valence_2	→	atom_valence_2
valence_2	→	"S" "(" "=" "O" ")" "(" "=" "O" ")"
valence_2	→	valence_3
valence_2	→	valence_3 "(" simple_bond ")"
valence_2	→	valence_4 "(" double_bond ")"
valence_3	→	atom_valence_3
valence_3	→	valence_4
valence_3	→	valence_4 "(" simple_bond ")"
valence_4	→	atom_valence_4
double_bond	→	"=" valence_2
double_bond	→	"=" valence_3 simple_bond
double_bond	→	"=" valence_4 double_bond
triple_bond	→	"#" valence_3
triple_bond	→	"#" valence_4 simple_bond
simple_bond	→	valence_3_num1 cycle1_n_bond
simple_bond	→	valence_3_num2 cycle2_n_bond
simple_bond	→	valence_3_num3 cycle3_n_bond
simple_bond	→	valence_3_num4 cycle4_n_bond
simple_bond	→	valence_3_num5 cycle5_n_bond
simple_bond	→	valence_3_num6 cycle6_n_bond
simple_bond	→	valence_4_num1 cycle1_n_double_bond
simple_bond	→	valence_4_num2 cycle2_n_double_bond
simple_bond	→	valence_4_num3 cycle3_n_double_bond
simple_bond	→	valence_4_num4 cycle4_n_double_bond
simple_bond	→	valence_4_num5 cycle5_n_double_bond
simple_bond	→	valence_4_num6 cycle6_n_double_bond
simple_bond	→	ring_n_segment
simple_bond	→	ring_n_segment simple_bond
valence_2	→	valence_4_num1 "(" cycle1_n_bond ")"
valence_2	→	valence_4_num2 "(" cycle2_n_bond ")"
valence_2	→	valence_4_num3 "(" cycle3_n_bond ")"
valence_2	→	valence_4_num4 "(" cycle4_n_bond ")"
valence_2	→	valence_4_num5 "(" cycle5_n_bond ")"
valence_2	→	valence_4_num6 "(" cycle6_n_bond ")"
cycle1_n_bond	→	cycle1_7_bond
cycle1_n_double_bond	→	cycle1_7_double_bond
cycle1_n-1_bond	→	cycle1_6_bond
cycle1_n-1_double_bond	→	cycle1_6_double_bond
cycle1_n-2_bond	→	cycle1_5_bond
cycle1_n-2_double_bond	→	cycle1_5_double_bond
cycle1_7_bond	→	cycle1_6_bond
cycle1_6_bond	→	cycle1_5_bond
cycle1_5_bond	→	cycle1_4_bond
cycle1_4_bond	→	cycle1_3_bond

cycle1_3_bond	→	cycle1_2_bond
cycle1_7_double_bond	→	cycle1_6_double_bond
cycle1_6_double_bond	→	cycle1_5_double_bond
cycle1_5_double_bond	→	cycle1_4_double_bond
cycle1_4_double_bond	→	cycle1_3_double_bond
cycle1_3_double_bond	→	cycle1_2_double_bond
cycle1_7_bond	→	valence_2 cycle1_6_bond
cycle1_7_bond	→	valence_3 cycle1_6_double_bond
cycle1_7_bond	→	ring_n_segment cycle1_6_bond
cycle1_7_double_bond	→	"=" valence_3 cycle1_6_bond
cycle1_6_bond	→	valence_2 cycle1_5_bond
cycle1_6_bond	→	valence_3 cycle1_5_double_bond
cycle1_6_bond	→	ring_n_segment cycle1_5_bond
cycle1_6_double_bond	→	"=" valence_3 cycle1_5_bond
cycle1_5_bond	→	valence_2 cycle1_4_bond
cycle1_5_bond	→	valence_3 cycle1_4_double_bond
cycle1_5_bond	→	ring_n_segment cycle1_4_bond
cycle1_5_double_bond	→	"=" valence_3 cycle1_4_bond
cycle1_4_bond	→	valence_2 cycle1_3_bond
cycle1_4_bond	→	valence_3 cycle1_3_double_bond
cycle1_4_bond	→	ring_n_segment cycle1_3_bond
cycle1_4_double_bond	→	"=" valence_3 cycle1_3_bond
cycle1_3_bond	→	valence_2 cycle1_2_bond
cycle1_3_bond	→	valence_3 cycle1_2_double_bond
cycle1_3_bond	→	ring_n_segment cycle1_2_bond
cycle1_3_double_bond	→	"=" valence_3 cycle1_2_bond
cycle1_2_bond	→	valence_2 valence_2_num1
cycle1_2_bond	→	valence_3 "=" valence_3_num1
cycle1_2_bond	→	ring_n_segment valence_2_num1
cycle1_2_double_bond	→	"=" valence_3 valence_2_num1
cycle2_n_bond	→	cycle2_7_bond
cycle2_n_double_bond	→	cycle2_7_double_bond
cycle2_n-1_bond	→	cycle2_6_bond
cycle2_n-1_double_bond	→	cycle2_6_double_bond
cycle2_n-2_bond	→	cycle2_5_bond
cycle2_n-2_double_bond	→	cycle2_5_double_bond
cycle2_7_bond	→	cycle2_6_bond
cycle2_6_bond	→	cycle2_5_bond
cycle2_5_bond	→	cycle2_4_bond
cycle2_4_bond	→	cycle2_3_bond
cycle2_3_bond	→	cycle2_2_bond
cycle2_7_double_bond	→	cycle2_6_double_bond
cycle2_6_double_bond	→	cycle2_5_double_bond
cycle2_5_double_bond	→	cycle2_4_double_bond

cycle2_4_double_bond	→	cycle2_3_double_bond
cycle2_3_double_bond	→	cycle2_2_double_bond
cycle2_7_bond	→	valence_2 cycle2_6_bond
cycle2_7_bond	→	valence_3 cycle2_6_double_bond
cycle2_7_bond	→	ring_n_segment cycle2_6_bond
cycle2_7_double_bond	→	"=" valence_3 cycle2_6_bond
cycle2_6_bond	→	valence_2 cycle2_5_bond
cycle2_6_bond	→	valence_3 cycle2_5_double_bond
cycle2_6_bond	→	ring_n_segment cycle2_5_bond
cycle2_6_double_bond	→	"=" valence_3 cycle2_5_bond
cycle2_5_bond	→	valence_2 cycle2_4_bond
cycle2_5_bond	→	valence_3 cycle2_4_double_bond
cycle2_5_bond	→	ring_n_segment cycle2_4_bond
cycle2_5_double_bond	→	"=" valence_3 cycle2_4_bond
cycle2_4_bond	→	valence_2 cycle2_3_bond
cycle2_4_bond	→	valence_3 cycle2_3_double_bond
cycle2_4_bond	→	ring_n_segment cycle2_3_bond
cycle2_4_double_bond	→	"=" valence_3 cycle2_3_bond
cycle2_3_bond	→	valence_2 cycle2_2_bond
cycle2_3_bond	→	valence_3 cycle2_2_double_bond
cycle2_3_bond	→	ring_n_segment cycle2_2_bond
cycle2_3_double_bond	→	"=" valence_3 cycle2_2_bond
cycle2_2_bond	→	valence_2 valence_2_num2
cycle2_2_bond	→	valence_3 "=" valence_3_num2
cycle2_2_bond	→	ring_n_segment valence_2_num2
cycle2_2_double_bond	→	"=" valence_3 valence_2_num2
cycle3_n_bond	→	cycle3_7_bond
cycle3_n_double_bond	→	cycle3_7_double_bond
cycle3_n-1_bond	→	cycle3_6_bond
cycle3_n-1_double_bond	→	cycle3_6_double_bond
cycle3_n-2_bond	→	cycle3_5_bond
cycle3_n-2_double_bond	→	cycle3_5_double_bond
cycle3_7_bond	→	cycle3_6_bond
cycle3_6_bond	→	cycle3_5_bond
cycle3_5_bond	→	cycle3_4_bond
cycle3_4_bond	→	cycle3_3_bond
cycle3_3_bond	→	cycle3_2_bond
cycle3_7_double_bond	→	cycle3_6_double_bond
cycle3_6_double_bond	→	cycle3_5_double_bond
cycle3_5_double_bond	→	cycle3_4_double_bond
cycle3_4_double_bond	→	cycle3_3_double_bond
cycle3_3_double_bond	→	cycle3_2_double_bond
cycle3_7_bond	→	valence_2 cycle3_6_bond
cycle3_7_bond	→	valence_3 cycle3_6_double_bond

cycle3_7_bond	→	ring_n_segment cycle3_6_bond
cycle3_7_double_bond	→	“=” valence_3 cycle3_6_bond
cycle3_6_bond	→	valence_2 cycle3_5_bond
cycle3_6_bond	→	valence_3 cycle3_5_double_bond
cycle3_6_bond	→	ring_n_segment cycle3_5_bond
cycle3_6_double_bond	→	“=” valence_3 cycle3_5_bond
cycle3_5_bond	→	valence_2 cycle3_4_bond
cycle3_5_bond	→	valence_3 cycle3_4_double_bond
cycle3_5_bond	→	ring_n_segment cycle3_4_bond
cycle3_5_double_bond	→	“=” valence_3 cycle3_4_bond
cycle3_4_bond	→	valence_2 cycle3_3_bond
cycle3_4_bond	→	valence_3 cycle3_3_double_bond
cycle3_4_bond	→	ring_n_segment cycle3_3_bond
cycle3_4_double_bond	→	“=” valence_3 cycle3_3_bond
cycle3_3_bond	→	valence_2 cycle3_2_bond
cycle3_3_bond	→	valence_3 cycle3_2_double_bond
cycle3_3_bond	→	ring_n_segment cycle3_2_bond
cycle3_3_double_bond	→	“=” valence_3 cycle3_2_bond
cycle3_2_bond	→	valence_2 valence_2_num3
cycle3_2_bond	→	valence_3 “=” valence_3_num3
cycle3_2_bond	→	ring_n_segment valence_2_num3
cycle3_2_double_bond	→	“=” valence_3 valence_2_num3
cycle4_n_bond	→	cycle4_7_bond
cycle4_n_double_bond	→	cycle4_7_double_bond
cycle4_n-1_bond	→	cycle4_6_bond
cycle4_n-1_double_bond	→	cycle4_6_double_bond
cycle4_n-2_bond	→	cycle4_5_bond
cycle4_n-2_double_bond	→	cycle4_5_double_bond
cycle4_7_bond	→	cycle4_6_bond
cycle4_6_bond	→	cycle4_5_bond
cycle4_5_bond	→	cycle4_4_bond
cycle4_4_bond	→	cycle4_3_bond
cycle4_3_bond	→	cycle4_2_bond
cycle4_7_double_bond	→	cycle4_6_double_bond
cycle4_6_double_bond	→	cycle4_5_double_bond
cycle4_5_double_bond	→	cycle4_4_double_bond
cycle4_4_double_bond	→	cycle4_3_double_bond
cycle4_3_double_bond	→	cycle4_2_double_bond
cycle4_7_bond	→	valence_2 cycle4_6_bond
cycle4_7_bond	→	valence_3 cycle4_6_double_bond
cycle4_7_bond	→	ring_n_segment cycle4_6_bond
cycle4_7_double_bond	→	“=” valence_3 cycle4_6_bond
cycle4_6_bond	→	valence_2 cycle4_5_bond
cycle4_6_bond	→	valence_3 cycle4_5_double_bond

cycle4_6_bond	→	ring_n_segment cycle4_5_bond
cycle4_6_double_bond	→	"=" valence_3 cycle4_5_bond
cycle4_5_bond	→	valence_2 cycle4_4_bond
cycle4_5_bond	→	valence_3 cycle4_4_double_bond
cycle4_5_bond	→	ring_n_segment cycle4_4_bond
cycle4_5_double_bond	→	"=" valence_3 cycle4_4_bond
cycle4_4_bond	→	valence_2 cycle4_3_bond
cycle4_4_bond	→	valence_3 cycle4_3_double_bond
cycle4_4_bond	→	ring_n_segment cycle4_3_bond
cycle4_4_double_bond	→	"=" valence_3 cycle4_3_bond
cycle4_3_bond	→	valence_2 cycle4_2_bond
cycle4_3_bond	→	valence_3 cycle4_2_double_bond
cycle4_3_bond	→	ring_n_segment cycle4_2_bond
cycle4_3_double_bond	→	"=" valence_3 cycle4_2_bond
cycle4_2_bond	→	valence_2 valence_2_num4
cycle4_2_bond	→	valence_3 "=" valence_3_num4
cycle4_2_bond	→	ring_n_segment valence_2_num4
cycle4_2_double_bond	→	"=" valence_3 valence_2_num4
cycle5_n_bond	→	cycle5_7_bond
cycle5_n_double_bond	→	cycle5_7_double_bond
cycle5_n-1_bond	→	cycle5_6_bond
cycle5_n-1_double_bond	→	cycle5_6_double_bond
cycle5_n-2_bond	→	cycle5_5_bond
cycle5_n-2_double_bond	→	cycle5_5_double_bond
cycle5_7_bond	→	cycle5_6_bond
cycle5_6_bond	→	cycle5_5_bond
cycle5_5_bond	→	cycle5_4_bond
cycle5_4_bond	→	cycle5_3_bond
cycle5_3_bond	→	cycle5_2_bond
cycle5_7_double_bond	→	cycle5_6_double_bond
cycle5_6_double_bond	→	cycle5_5_double_bond
cycle5_5_double_bond	→	cycle5_4_double_bond
cycle5_4_double_bond	→	cycle5_3_double_bond
cycle5_3_double_bond	→	cycle5_2_double_bond
cycle5_7_bond	→	valence_2 cycle5_6_bond
cycle5_7_bond	→	valence_3 cycle5_6_double_bond
cycle5_7_bond	→	ring_n_segment cycle5_6_bond
cycle5_7_double_bond	→	"=" valence_3 cycle5_6_bond
cycle5_6_bond	→	valence_2 cycle5_5_bond
cycle5_6_bond	→	valence_3 cycle5_5_double_bond
cycle5_6_bond	→	ring_n_segment cycle5_5_bond
cycle5_6_double_bond	→	"=" valence_3 cycle5_5_bond
cycle5_5_bond	→	valence_2 cycle5_4_bond
cycle5_5_bond	→	valence_3 cycle5_4_double_bond

cycle5_5_bond	→	ring_n_segment cycle5_4_bond
cycle5_5_double_bond	→	“=” valence_3 cycle5_4_bond
cycle5_4_bond	→	valence_2 cycle5_3_bond
cycle5_4_bond	→	valence_3 cycle5_3_double_bond
cycle5_4_bond	→	ring_n_segment cycle5_3_bond
cycle5_4_double_bond	→	“=” valence_3 cycle5_3_bond
cycle5_3_bond	→	valence_2 cycle5_2_bond
cycle5_3_bond	→	valence_3 cycle5_2_double_bond
cycle5_3_bond	→	ring_n_segment cycle5_2_bond
cycle5_3_double_bond	→	“=” valence_3 cycle5_2_bond
cycle5_2_bond	→	valence_2 valence_2_num5
cycle5_2_bond	→	valence_3 “=” valence_3_num5
cycle5_2_bond	→	ring_n_segment valence_2_num5
cycle5_2_double_bond	→	“=” valence_3 valence_2_num5
cycle6_n_bond	→	cycle6_7_bond
cycle6_n_double_bond	→	cycle6_7_double_bond
cycle6_n-1_bond	→	cycle6_6_bond
cycle6_n-1_double_bond	→	cycle6_6_double_bond
cycle6_n-2_bond	→	cycle6_5_bond
cycle6_n-2_double_bond	→	cycle6_5_double_bond
cycle6_7_bond	→	cycle6_6_bond
cycle6_6_bond	→	cycle6_5_bond
cycle6_5_bond	→	cycle6_4_bond
cycle6_4_bond	→	cycle6_3_bond
cycle6_3_bond	→	cycle6_2_bond
cycle6_7_double_bond	→	cycle6_6_double_bond
cycle6_6_double_bond	→	cycle6_5_double_bond
cycle6_5_double_bond	→	cycle6_4_double_bond
cycle6_4_double_bond	→	cycle6_3_double_bond
cycle6_3_double_bond	→	cycle6_2_double_bond
cycle6_7_bond	→	valence_2 cycle6_6_bond
cycle6_7_bond	→	valence_3 cycle6_6_double_bond
cycle6_7_bond	→	ring_n_segment cycle6_6_bond
cycle6_7_double_bond	→	“=” valence_3 cycle6_6_bond
cycle6_6_bond	→	valence_2 cycle6_5_bond
cycle6_6_bond	→	valence_3 cycle6_5_double_bond
cycle6_6_bond	→	ring_n_segment cycle6_5_bond
cycle6_6_double_bond	→	“=” valence_3 cycle6_5_bond
cycle6_5_bond	→	valence_2 cycle6_4_bond
cycle6_5_bond	→	valence_3 cycle6_4_double_bond
cycle6_5_bond	→	ring_n_segment cycle6_4_bond
cycle6_5_double_bond	→	“=” valence_3 cycle6_4_bond
cycle6_4_bond	→	valence_2 cycle6_3_bond
cycle6_4_bond	→	valence_3 cycle6_3_double_bond

cycle6_4_bond	→	ring_n_segment cycle6_3_bond
cycle6_4_double_bond	→	“=” valence_3 cycle6_3_bond
cycle6_3_bond	→	valence_2 cycle6_2_bond
cycle6_3_bond	→	valence_3 cycle6_2_double_bond
cycle6_3_bond	→	ring_n_segment cycle6_2_bond
cycle6_3_double_bond	→	“=” valence_3 cycle6_2_bond
cycle6_2_bond	→	valence_2 valence_2_num6
cycle6_2_bond	→	valence_3 “=” valence_3_num6
cycle6_2_bond	→	ring_n_segment valence_2_num6
cycle6_2_double_bond	→	“=” valence_3 valence_2_num6
ring_n_segment	→	valence_3 “(” cycle1_n-2_bond “)” valence_3_num1
ring_n_segment	→	valence_4 “(” cycle1_n-2_bond “)” “=” valence_4_num1
ring_n_segment	→	valence_4 “(” cycle1_n-2_double_bond “)” valence_3_num1
ring_n_segment	→	valence_3 “(” cycle2_n-2_bond “)” valence_3_num2
ring_n_segment	→	valence_4 “(” cycle2_n-2_bond “)” “=” valence_4_num2
ring_n_segment	→	valence_4 “(” cycle2_n-2_double_bond “)” valence_3_num2
ring_n_segment	→	valence_3 “(” cycle3_n-2_bond “)” valence_3_num3
ring_n_segment	→	valence_4 “(” cycle3_n-2_bond “)” “=” valence_4_num3
ring_n_segment	→	valence_4 “(” cycle3_n-2_double_bond “)” valence_3_num3
ring_n_segment	→	valence_3 “(” cycle4_n-2_bond “)” valence_3_num4
ring_n_segment	→	valence_4 “(” cycle4_n-2_bond “)” “=” valence_4_num4
ring_n_segment	→	valence_4 “(” cycle4_n-2_double_bond “)” valence_3_num4
ring_n_segment	→	valence_3 “(” cycle5_n-2_bond “)” valence_3_num5
ring_n_segment	→	valence_4 “(” cycle5_n-2_bond “)” “=” valence_4_num5
ring_n_segment	→	valence_4 “(” cycle5_n-2_double_bond “)” valence_3_num5
ring_n_segment	→	valence_3 “(” cycle6_n-2_bond “)” valence_3_num6
ring_n_segment	→	valence_4 “(” cycle6_n-2_bond “)” “=” valence_4_num6
ring_n_segment	→	valence_4 “(” cycle6_n-2_double_bond “)” valence_3_num6
valence_2_num1	→	atom_valence_2 “1”
valence_2_num1	→	“S” “1” “(” “=” “O” “)” “(” “=” “O” “)”
valence_2_num1	→	valence_3_num1
valence_2_num1	→	valence_3_num1 “(” simple_bond “)”
valence_2_num1	→	valence_4_num1 “(” double_bond “)”
valence_3_num1	→	atom_valence_3 “1”
valence_3_num1	→	valence_4_num1
valence_3_num1	→	valence_4_num1 “(” simple_bond “)”
valence_4_num1	→	atom_valence_4 “1”
valence_2_num2	→	atom_valence_2 “2”
valence_2_num2	→	“S” “2” “(” “=” “O” “)” “(” “=” “O” “)”
valence_2_num2	→	valence_3_num2
valence_2_num2	→	valence_3_num2 “(” simple_bond “)”
valence_2_num2	→	valence_4_num2 “(” double_bond “)”
valence_3_num2	→	atom_valence_3 “2”
valence_3_num2	→	valence_4_num2

valence_3_num2	→	valence_4_num2 “(” simple_bond “)”
valence_4_num2	→	atom_valence_4 “2”
valence_2_num3	→	atom_valence_2 “3”
valence_2_num3	→	“S” “3” “(” “=” “O” “)” “(” “=” “O” “)”
valence_2_num3	→	valence_3_num3
valence_2_num3	→	valence_3_num3 “(” simple_bond “)”
valence_2_num3	→	valence_4_num3 “(” double_bond “)”
valence_3_num3	→	atom_valence_3 “3”
valence_3_num3	→	valence_4_num3
valence_3_num3	→	valence_4_num3 “(” simple_bond “)”
valence_4_num3	→	atom_valence_4 “3”
valence_2_num4	→	atom_valence_2 “4”
valence_2_num4	→	“S” “4” “(” “=” “O” “)” “(” “=” “O” “)”
valence_2_num4	→	valence_3_num4
valence_2_num4	→	valence_3_num4 “(” simple_bond “)”
valence_2_num4	→	valence_4_num4 “(” double_bond “)”
valence_3_num4	→	atom_valence_3 “4”
valence_3_num4	→	valence_4_num4
valence_3_num4	→	valence_4_num4 “(” simple_bond “)”
valence_4_num4	→	atom_valence_4 “4”
valence_2_num5	→	atom_valence_2 “5”
valence_2_num5	→	“S” “5” “(” “=” “O” “)” “(” “=” “O” “)”
valence_2_num5	→	valence_3_num5
valence_2_num5	→	valence_3_num5 “(” simple_bond “)”
valence_2_num5	→	valence_4_num5 “(” double_bond “)”
valence_3_num5	→	atom_valence_3 “5”
valence_3_num5	→	valence_4_num5
valence_3_num5	→	valence_4_num5 “(” simple_bond “)”
valence_4_num5	→	atom_valence_4 “5”
valence_2_num6	→	atom_valence_2 “6”
valence_2_num6	→	“S” “6” “(” “=” “O” “)” “(” “=” “O” “)”
valence_2_num6	→	valence_3_num6
valence_2_num6	→	valence_3_num6 “(” simple_bond “)”
valence_2_num6	→	valence_4_num6 “(” double_bond “)”
valence_3_num6	→	atom_valence_3 “6”
valence_3_num6	→	valence_4_num6
valence_3_num6	→	valence_4_num6 “(” simple_bond “)”
valence_4_num6	→	atom_valence_4 “6”
simple_bond	→	aromatic_ring1_5
simple_bond	→	aromatic_ring2_5
simple_bond	→	aromatic_ring3_5
simple_bond	→	aromatic_ring4_5
simple_bond	→	aromatic_ring5_5
simple_bond	→	aromatic_ring6_5

simple_bond	→	aromatic_ring1_6
simple_bond	→	aromatic_ring2_6
simple_bond	→	aromatic_ring3_6
simple_bond	→	aromatic_ring4_6
simple_bond	→	aromatic_ring5_6
simple_bond	→	aromatic_ring6_6
simple_bond	→	double_aromatic_ring1
simple_bond	→	double_aromatic_ring2
simple_bond	→	double_aromatic_ring3
simple_bond	→	double_aromatic_ring4
simple_bond	→	double_aromatic_ring5
aromatic_os	→	side_aliphatic_ring1
aromatic_os	→	side_aliphatic_ring2
aromatic_os	→	side_aliphatic_ring3
aromatic_os	→	side_aliphatic_ring4
aromatic_os	→	side_aliphatic_ring5
aromatic_os	→	side_aliphatic_ring6
full_aromatic_segment	→	side_aliphatic_ring1_segment
full_aromatic_segment	→	side_aliphatic_ring2_segment
full_aromatic_segment	→	side_aliphatic_ring3_segment
full_aromatic_segment	→	side_aliphatic_ring4_segment
full_aromatic_segment	→	side_aliphatic_ring5_segment
full_aromatic_segment	→	side_aliphatic_ring6_segment
full_aromatic_segment	→	aromatic_atom aromatic_atom
aromatic_atom	→	“n”
aromatic_atom	→	“c”
aromatic_atom	→	“c” “(” simple_bond “)”
aromatic_os	→	“o”
aromatic_os	→	“s”
aromatic_os	→	“n” “(” simple_bond “)”
aromatic_os	→	“[” “n” hydrogen_1 “]”
starting_aromatic_c_num1	→	“c” “1”
aromatic_atom_num1	→	“n” “1”
aromatic_atom_num1	→	“c” “1”
aromatic_atom_num1	→	“c” “1” simple_bond
aromatic_os_num1	→	“o” “1”
aromatic_os_num1	→	“s” “1”
aromatic_os_num1	→	“n” “1” simple_bond
aromatic_ring1_6	→	starting_aromatic_c_num1 aromatic_atom full_aromatic_segment aromatic_atom aromatic_atom_num1
aromatic_ring1_6	→	starting_aromatic_c_num1 full_aromatic_segment full_aromatic_segment aromatic_atom_num1
aromatic_ring1_5	→	starting_aromatic_c_num1 aromatic_os full_aromatic_segment aromatic_atom_num1

aromatic_ring1_5	→	starting_aromatic_c_num1 aromatic_atom aromatic_os aromatic_atom aromatic_atom_num1
aromatic_ring1_5	→	starting_aromatic_c_num1 full_aromatic_segment aromatic_os aromatic_atom_num1
aromatic_ring1_5	→	starting_aromatic_c_num1 full_aromatic_segment aromatic_atom aromatic_os_num1
aromatic_ring1_5	→	starting_aromatic_c_num1 aromatic_atom full_aromatic_segment aromatic_os_num1
double_aromatic_ring1	→	"c" "1" aromatic_atom aromatic_atom aromatic_atom "c" "2" "c" "1" aromatic_atom aromatic_atom aromatic_atom aromatic_atom_num2
double_aromatic_ring1	→	"c" "1" aromatic_atom aromatic_atom aromatic_atom "c" "2" "n" "1" aromatic_atom aromatic_atom aromatic_atom_num2
double_aromatic_ring1	→	"c" "1" aromatic_atom aromatic_atom aromatic_atom "n" "2" "c" "1" aromatic_atom aromatic_atom aromatic_atom_num2
side_aliphatic_ring1	→	"c" "1" "(" cycle1_n_bond ")"
side_aliphatic_ring1_segment	→	"c" "1" "c" "(" cycle1_n-1_bond ")"
side_aliphatic_ring1_segment	→	"c" "(" cycle1_n-1_bond ")" "c" "1"
starting_aromatic_c_num2	→	"c" "2"
aromatic_atom_num2	→	"n" "2"
aromatic_atom_num2	→	"c" "2"
aromatic_atom_num2	→	"c" "2" simple_bond
aromatic_os_num2	→	"o" "2"
aromatic_os_num2	→	"s" "2"
aromatic_os_num2	→	"n" "2" simple_bond
aromatic_ring2_6	→	starting_aromatic_c_num2 aromatic_atom full_aromatic_segment aromatic_atom aromatic_atom_num2
aromatic_ring2_6	→	starting_aromatic_c_num2 full_aromatic_segment full_aromatic_segment aromatic_atom_num2
aromatic_ring2_5	→	starting_aromatic_c_num2 aromatic_os full_aromatic_segment aromatic_atom_num2
aromatic_ring2_5	→	starting_aromatic_c_num2 aromatic_atom aromatic_os aromatic_atom aromatic_atom_num2
aromatic_ring2_5	→	starting_aromatic_c_num2 full_aromatic_segment aromatic_os aromatic_atom_num2
aromatic_ring2_5	→	starting_aromatic_c_num2 full_aromatic_segment aromatic_atom aromatic_os_num2
aromatic_ring2_5	→	starting_aromatic_c_num2 aromatic_atom full_aromatic_segment aromatic_os_num2
double_aromatic_ring2	→	"c" "2" aromatic_atom aromatic_atom aromatic_atom "c" "3" "c" "2" aromatic_atom aromatic_atom aromatic_atom aromatic_atom_num3
double_aromatic_ring2	→	"c" "2" aromatic_atom aromatic_atom aromatic_atom "c" "3" "n" "2" aromatic_atom aromatic_atom aromatic_atom_num3
double_aromatic_ring2	→	"c" "2" aromatic_atom aromatic_atom aromatic_atom "n" "3" "c" "2" aromatic_atom aromatic_atom aromatic_atom_num3

side_aliphatic_ring2	→	“c” “2” “(” cycle2_n_bond “)”
side_aliphatic_ring2_segment	→	“c” “2” “c” “(” cycle2_n-1_bond “)”
side_aliphatic_ring2_segment	→	“c” “(” cycle2_n-1_bond “)” “c” “2”
starting_aromatic_c_num3	→	“c” “3”
aromatic_atom_num3	→	“n” “3”
aromatic_atom_num3	→	“c” “3”
aromatic_atom_num3	→	“c” “3” simple_bond
aromatic_os_num3	→	“o” “3”
aromatic_os_num3	→	“s” “3”
aromatic_os_num3	→	“n” “3” simple_bond
aromatic_ring3_6	→	starting_aromatic_c_num3 aromatic_atom full_aromatic_segment aromatic_atom aromatic_atom_num3
aromatic_ring3_6	→	starting_aromatic_c_num3 full_aromatic_segment full_aromatic_segment aromatic_atom_num3
aromatic_ring3_5	→	starting_aromatic_c_num3 aromatic_os full_aromatic_segment aromatic_atom_num3
aromatic_ring3_5	→	starting_aromatic_c_num3 aromatic_atom aromatic_os aromatic_atom aromatic_atom_num3
aromatic_ring3_5	→	starting_aromatic_c_num3 full_aromatic_segment aromatic_os aromatic_atom_num3
aromatic_ring3_5	→	starting_aromatic_c_num3 full_aromatic_segment aromatic_atom aromatic_os_num3
aromatic_ring3_5	→	starting_aromatic_c_num3 aromatic_atom full_aromatic_segment aromatic_os_num3
double_aromatic_ring3	→	“c” “3” aromatic_atom aromatic_atom aromatic_atom “c” “4” “c” “3” aromatic_atom aromatic_atom aromatic_atom aromatic_atom_num4
double_aromatic_ring3	→	“c” “3” aromatic_atom aromatic_atom aromatic_atom “c” “4” “n” “3” aromatic_atom aromatic_atom aromatic_atom_num4
double_aromatic_ring3	→	“c” “3” aromatic_atom aromatic_atom aromatic_atom “n” “4” “c” “3” aromatic_atom aromatic_atom aromatic_atom_num4
side_aliphatic_ring3	→	“c” “3” “(” cycle3_n_bond “)”
side_aliphatic_ring3_segment	→	“c” “3” “c” “(” cycle3_n-1_bond “)”
side_aliphatic_ring3_segment	→	“c” “(” cycle3_n-1_bond “)” “c” “3”
starting_aromatic_c_num4	→	“c” “4”
aromatic_atom_num4	→	“n” “4”
aromatic_atom_num4	→	“c” “4”
aromatic_atom_num4	→	“c” “4” simple_bond
aromatic_os_num4	→	“o” “4”
aromatic_os_num4	→	“s” “4”
aromatic_os_num4	→	“n” “4” simple_bond
aromatic_ring4_6	→	starting_aromatic_c_num4 aromatic_atom full_aromatic_segment aromatic_atom aromatic_atom_num4
aromatic_ring4_6	→	starting_aromatic_c_num4 full_aromatic_segment full_aromatic_segment aromatic_atom_num4

aromatic_ring4_5	→	starting_aromatic_c_num4 aromatic_os full_aromatic_segment aromatic_atom_num4
aromatic_ring4_5	→	starting_aromatic_c_num4 aromatic_atom aromatic_os aromatic_atom aromatic_atom_num4
aromatic_ring4_5	→	starting_aromatic_c_num4 full_aromatic_segment aromatic_os aromatic_atom_num4
aromatic_ring4_5	→	starting_aromatic_c_num4 full_aromatic_segment aromatic_atom aromatic_os_num4
aromatic_ring4_5	→	starting_aromatic_c_num4 aromatic_atom full_aromatic_segment aromatic_os_num4
double_aromatic_ring4	→	“c” “4” aromatic_atom aromatic_atom aromatic_atom “c” “5” “c” “4” aromatic_atom aromatic_atom aromatic_atom aromatic_atom_num5
double_aromatic_ring4	→	“c” “4” aromatic_atom aromatic_atom aromatic_atom “c” “5” “n” “4” aromatic_atom aromatic_atom aromatic_atom_num5
double_aromatic_ring4	→	“c” “4” aromatic_atom aromatic_atom aromatic_atom “n” “5” “c” “4” aromatic_atom aromatic_atom aromatic_atom_num5
side_aliphatic_ring4	→	“c” “4” “(” cycle4_n_bond “)”
side_aliphatic_ring4_segment	→	“c” “4” “c” “(” cycle4_n-1_bond “)”
side_aliphatic_ring4_segment	→	“c” “(” cycle4_n-1_bond “)” “c” “4”
starting_aromatic_c_num5	→	“c” “5”
aromatic_atom_num5	→	“n” “5”
aromatic_atom_num5	→	“c” “5”
aromatic_atom_num5	→	“c” “5” simple_bond
aromatic_os_num5	→	“o” “5”
aromatic_os_num5	→	“s” “5”
aromatic_os_num5	→	“n” “5” simple_bond
aromatic_ring5_6	→	starting_aromatic_c_num5 aromatic_atom full_aromatic_segment aromatic_atom aromatic_atom_num5
aromatic_ring5_6	→	starting_aromatic_c_num5 full_aromatic_segment full_aromatic_segment aromatic_atom_num5
aromatic_ring5_5	→	starting_aromatic_c_num5 aromatic_os full_aromatic_segment aromatic_atom_num5
aromatic_ring5_5	→	starting_aromatic_c_num5 aromatic_atom aromatic_os aromatic_atom aromatic_atom_num5
aromatic_ring5_5	→	starting_aromatic_c_num5 full_aromatic_segment aromatic_os aromatic_atom_num5
aromatic_ring5_5	→	starting_aromatic_c_num5 full_aromatic_segment aromatic_atom aromatic_os_num5
aromatic_ring5_5	→	starting_aromatic_c_num5 aromatic_atom full_aromatic_segment aromatic_os_num5
double_aromatic_ring5	→	“c” “5” aromatic_atom aromatic_atom aromatic_atom “c” “6” “c” “5” aromatic_atom aromatic_atom aromatic_atom aromatic_atom_num6
double_aromatic_ring5	→	“c” “5” aromatic_atom aromatic_atom aromatic_atom “c” “6” “n” “5” aromatic_atom aromatic_atom aromatic_atom_num6

double_aromatic_ring5	→	“c” “5” aromatic_atom aromatic_atom aromatic_atom “n” “6” “c” “5” aromatic_atom aromatic_atom aromatic_atom_num6
side_aliphatic_ring5	→	“c” “5” “(” cycle5_n_bond “)”
side_aliphatic_ring5_segment	→	“c” “5” “c” “(” cycle5_n-1_bond “)”
side_aliphatic_ring5_segment	→	“c” “(” cycle5_n-1_bond “)” “c” “5”
starting_aromatic_c_num6	→	“c” “6”
aromatic_atom_num6	→	“n” “6”
aromatic_atom_num6	→	“c” “6”
aromatic_atom_num6	→	“c” “6” simple_bond
aromatic_os_num6	→	“o” “6”
aromatic_os_num6	→	“s” “6”
aromatic_os_num6	→	“n” “6” simple_bond
aromatic_ring6_6	→	starting_aromatic_c_num6 aromatic_atom full_aromatic_segment aromatic_atom aromatic_atom_num6
aromatic_ring6_6	→	starting_aromatic_c_num6 full_aromatic_segment full_aromatic_segment aromatic_atom_num6
aromatic_ring6_5	→	starting_aromatic_c_num6 aromatic_os full_aromatic_segment aromatic_atom_num6
aromatic_ring6_5	→	starting_aromatic_c_num6 aromatic_atom aromatic_os aromatic_atom aromatic_atom_num6
aromatic_ring6_5	→	starting_aromatic_c_num6 full_aromatic_segment aromatic_os aromatic_atom_num6
aromatic_ring6_5	→	starting_aromatic_c_num6 full_aromatic_segment aromatic_atom aromatic_os_num6
aromatic_ring6_5	→	starting_aromatic_c_num6 aromatic_atom full_aromatic_segment aromatic_os_num6
side_aliphatic_ring6	→	“c” “6” “(” cycle6_n_bond “)”
side_aliphatic_ring6_segment	→	“c” “6” “c” “(” cycle6_n-1_bond “)”
side_aliphatic_ring6_segment	→	“c” “(” cycle6_n-1_bond “)” “c” “6’

APPENDIX B GRAMMAR FOR LIPINSKI'S RULE OF 5

Grammar used to model molecular properties in Appendix B.

-	smiles	→	simple_bond
-	smiles	→	atom_valence_1 simple_bond
-	smiles	→	atom_valence_2 double_bond
-	smiles	→	atom_valence_3 triple_bond
+	smiles	→	valence_1
+	smiles	→	valence_1 simple_bond
+	smiles	→	valence_2 double_bond
+	smiles	→	valence_3 triple_bond
+	smiles	→	valence_1 slash valence_3 "=" valence_3 slash valence_2
+	smiles	→	valence_2_num1 cycle1_n_bond
+	smiles	→	valence_2_num2 cycle2_n_bond
+	smiles	→	valence_2_num3 cycle3_n_bond
+	smiles	→	valence_2_num4 cycle4_n_bond
+	smiles	→	valence_2_num5 cycle5_n_bond
+	smiles	→	valence_2_num6 cycle6_n_bond
+	smiles	→	valence_3_num1 cycle1_n_double_bond
+	smiles	→	valence_3_num2 cycle2_n_double_bond
+	smiles	→	valence_3_num3 cycle3_n_double_bond
+	smiles	→	valence_3_num4 cycle4_n_double_bond
+	smiles	→	valence_3_num5 cycle5_n_double_bond
+	smiles	→	valence_3_num6 cycle6_n_double_bond
+	smiles	→	ring_n_start
+	smiles	→	ring_n_start simple_bond
+	smiles	→	aromatic_ring1_5
+	smiles	→	aromatic_ring2_5
+	smiles	→	aromatic_ring3_5
+	smiles	→	aromatic_ring4_5
+	smiles	→	aromatic_ring5_5
+	smiles	→	aromatic_ring6_5
+	smiles	→	aromatic_ring1_6
+	smiles	→	aromatic_ring2_6
+	smiles	→	aromatic_ring3_6
+	smiles	→	aromatic_ring4_6
+	smiles	→	aromatic_ring5_6
+	smiles	→	aromatic_ring6_6
+	smiles	→	double_aromatic_ring1
+	smiles	→	double_aromatic_ring2
+	smiles	→	double_aromatic_ring3

+	smiles	$\rightarrow$	double_aromatic_ring4
+	smiles	$\rightarrow$	double_aromatic_ring5
-	atom_valence_1	$\rightarrow$	"[" "N" hydrogen_3 "+" "]"
+	atom_valence_1	$\rightarrow$	"[" "N_D" hydrogen_3 "+" "]"
+	atom_valence_1	$\rightarrow$	"O_D"
+	atom_valence_1	$\rightarrow$	"S_D"
+	atom_valence_1	$\rightarrow$	"N_D"
+	atom_valence_1	$\rightarrow$	"[" "C" "@" hydrogen_1 "]"
+	atom_valence_1	$\rightarrow$	"[" "C" "@" "@" hydrogen_1 "]"
+	atom_valence_1	$\rightarrow$	"[" "N_D" hydrogen_1 "+" "]"
+	atom_valence_1	$\rightarrow$	"C"
+	atom_valence_1	$\rightarrow$	"[" "C" "@" "]"
+	atom_valence_1	$\rightarrow$	"[" "C" "@" "@" "]"
+	atom_valence_1	$\rightarrow$	"[" "N" "+" "]"
+	atom_valence_2	$\rightarrow$	"N_D"
+	atom_valence_2	$\rightarrow$	"[" "C" "@" hydrogen_1 "]"
+	atom_valence_2	$\rightarrow$	"[" "C" "@" "@" hydrogen_1 "]"
+	atom_valence_2	$\rightarrow$	"[" "N_D" hydrogen_1 "+" "]"
+	atom_valence_2	$\rightarrow$	"C"
+	atom_valence_2	$\rightarrow$	"[" "C" "@" "]"
+	atom_valence_2	$\rightarrow$	"[" "C" "@" "@" "]"
+	atom_valence_2	$\rightarrow$	"[" "N" "+" "]"
-	atom_valence_3	$\rightarrow$	"[" "N" hydrogen_1 "+" "]"
+	atom_valence_3	$\rightarrow$	"[" "N_D" hydrogen_1 "+" "]"
+	atom_valence_3	$\rightarrow$	"C"
+	atom_valence_3	$\rightarrow$	"[" "C" "@" "]"
+	atom_valence_3	$\rightarrow$	"[" "C" "@" "@" "]"
+	atom_valence_3	$\rightarrow$	"[" "N" "+" "]"
-	valence_1	$\rightarrow$	valence_2
-	valence_2	$\rightarrow$	valence_3
-	valence_3	$\rightarrow$	valence_4
+	ring_n_start	$\rightarrow$	valence_2 "(" cycle1_n-2_bond ")" valence_3_num1
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle1_n-2_bond ")" "=" valence_4_num1
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle1_n-2_double_bond ")" valence_3_num1
+	ring_n_start	$\rightarrow$	valence_2 "(" cycle2_n-2_bond ")" valence_3_num2
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle2_n-2_bond ")" "=" valence_4_num2
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle2_n-2_double_bond ")" valence_3_num2
+	ring_n_start	$\rightarrow$	valence_2 "(" cycle3_n-2_bond ")" valence_3_num3
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle3_n-2_bond ")" "=" valence_4_num3
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle3_n-2_double_bond ")" valence_3_num3
+	ring_n_start	$\rightarrow$	valence_2 "(" cycle4_n-2_bond ")" valence_3_num4
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle4_n-2_bond ")" "=" valence_4_num4
+	ring_n_start	$\rightarrow$	valence_3 "(" cycle4_n-2_double_bond ")" valence_3_num4
+	ring_n_start	$\rightarrow$	valence_2 "(" cycle5_n-2_bond ")" valence_3_num5

+	ring_n_start	→	valence_3 “(” cycle5_n-2_bond “)” “=” valence_4_num5
+	ring_n_start	→	valence_3 “(” cycle5_n-2_double_bond “)” valence_3_num5
+	ring_n_start	→	valence_2 “(” cycle6_n-2_bond “)” valence_3_num6
+	ring_n_start	→	valence_3 “(” cycle6_n-2_bond “)” “=” valence_4_num6
+	ring_n_start	→	valence_3 “(” cycle6_n-2_double_bond “)” valence_3_num6