Grammar-Constrained Decoding for Large Language Models

Saibo Geng, Arina Rak, Robert West

Data Science Lab, EPFL

June 18, 2024

Outline of talk

Table of Contents

Table of Contents

What is Large Language Model (LLM)?

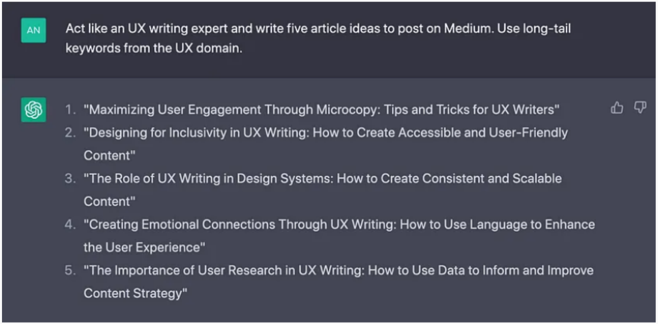


Figure: Example of ChatGPT convseration

Large Language Model = ChatGPT?

For many people, LLMs are synonymous with ChatGPT because ChatGPT is the first LLM that has been widely used by the public. It is also **by far** the most popular LLM.

Large Language Model

Large Language Model is auto-completer

From an academic perspective, LLMs are probabilistic models that gives the probability of the next word given the previous words, i.e. $P(w_i|w_1, w_2, ..., w_{i-1})$.

It's not wrong to say that

- LLMs are a **super powerful autocompleter**.
- the autocomplete system on your smartphone is a **tiny LLM**.

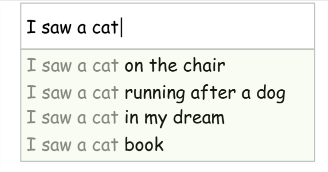


Figure: Example of autocompleter, taken from Voita [2024]

Why is Large Language Model Powerful?

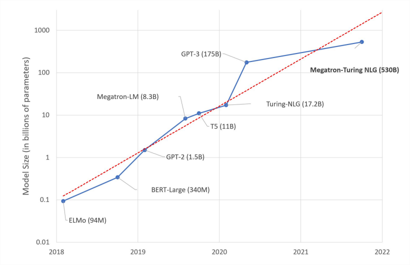


Figure: LLM's Moore's Law, from Huggingface

Langauge models are fundamentally autocompleters, why they suddenly become so powerful?

- **better** neural network architectures and training algorithms
- **much bigger** model
- **much much more** data
- **much much much .... more** compute (GPUs, TPUs, etc.)

GCD is an orthogonal research direction

Grammar-Constrained Decoding is orthogonal to the following aspects of the model:

1. Model architecture, size
2. Data quality, quantity
3. Training
4. Computation

Rather, we focus on the **generation process** itself, which is

1. **Agnostic** to the model architecture -> general
2. **Deterministic** -> robust
3. **Algorithm-based** -> Interpretable

What the heck is Grammar-Constrained Decoding?

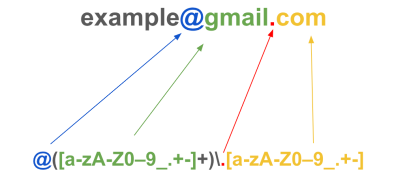


Figure: Example of Regular Expression

GCD = Regular Expression for LLM

The best way to explain Grammar-Constrained Decoding is to think of it as a regular expression for LLMs.

1. **Regular Expression**: an expression that defines a pattern; return strings that match the pattern.
2. **Grammar-Constrained Decoding**: an expression that define a pattern for LLMs; guide LLMs to generate strings that match the pattern.
3. **Regular Expression** is indeed a special type of **Grammar**.

Grammar-Constrained Decoding is behind OpenAI JSON mode

JSON mode

A common way to use Chat Completions is to instruct the model to always return a JSON object that makes sense for your use case, by specifying this in the system message. While this does work in some cases, occasionally the models may generate output that does not parse to valid JSON objects.

To prevent these errors and improve model performance, when using `gpt-4o`, `gpt-4-turbo`, or `gpt-3.5-turbo`, you can set response_format to `{ "type": "json_object" }` to enable JSON mode. When JSON mode is enabled, the model is constrained to only generate strings that parse into valid JSON object.

Important notes:

- When using JSON mode, **always** instruct the model to produce JSON via some message in the conversation, for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit. To help ensure you don't forget, the API will throw an error if the string `"JSON"` does not appear somewhere in the context.
- The JSON in the message the model returns may be partial (i.e. cut off) if `finish_reason` is `length`, which indicates the generation exceeded `max_tokens` or the conversation exceeded the token limit. To guard against this, check `finish_reason` before parsing the response.
- JSON mode will not guarantee the output matches any specific schema, only that it is valid and parses without errors.

Figure: OpenAI JSON mode from the OpenAI website

As LLMs are fundamentally autocompleters, they **iteratively** estimate the probability of the next word given the previous words.
The **generation** process is as follows:

1. **LLM** estimate the probability of the next word
2. **choose wisely** the next word
3. **repeat** until LLM thinks the sentence is finished

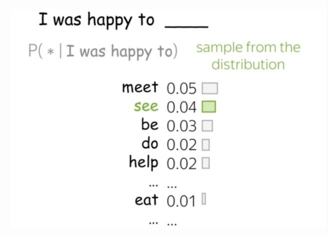


Figure: How LLM generate the next words, from Voita [2024]

Generation Process

- The generation process of LLM can be viewed as a **tree structure**
- This is true for **all LLMs** regardless of their implementation.
- This process is also known as **decoding**
- Different approaches to get better text from LLMs are called **decoding strategies**

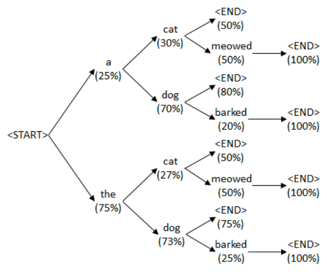


Figure: Generation process visualized as a tree structure

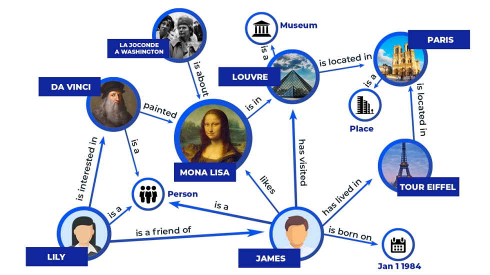
Table of Contents

Knowledge Graph

Definition

A knowledge graph is a graph-structured knowledge base that represents real-world entities and their relationships.

- **Nodes**: Entities
- **Edges**: Relationships
- **Triples**: (subject, relation, object)
- Example: Wikidata, DBpedia, YAGO
- Size: 10M+ entities



Task Description

Given a text (article, sentence, etc.), extract a set of facts under the form of triplets (subject, relation, object).

Input: `MONA LISA is a painting by Leonardo da Vinci.`

Expected Output: `{[DA VINCI, painted, MONA LISA]}`

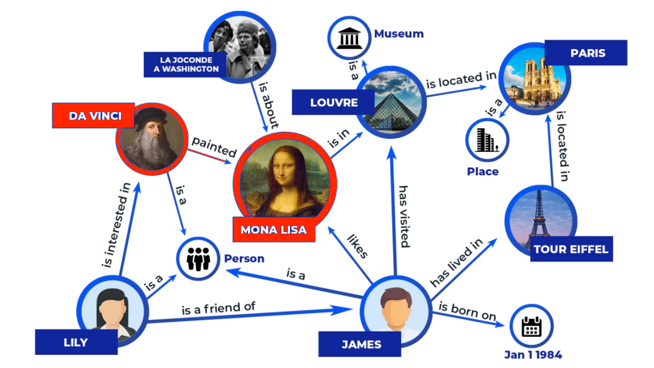


Figure: A subset of knowledge graph

Challenge of Reliable Triplets Extraction

Consistency Constraint

1. The extracted entities are linked to a knowledge graph, e.g.

MONA LISA is a painting by Leonardo da Vinci.

$\rightarrow$

[MONA LISA(Q12418), painted(R17), Da Vinci(Q762)]

$\neq$

[MONA LISA(Q12418), painted by(?), Leonardo da Vinci(?)]

Challenges

- Output needs to be **strictly** matched to the given knowledge graph
- Size of the knowledge graph can be large, i.e. Wikidata has **10M+** entities

How can we ensure the triplets extracted are consistent with the knowledge graph?

Can we constrain the generation process?

High-level idea

At some generation steps, we should **constrain** the token selection to **specific tokens** to improve the generation quality.
For example, as **Leonardo Da Vinci** is not in KG, we should **remove** it from the generation process.
→ This is called **Constrained Decoding**.

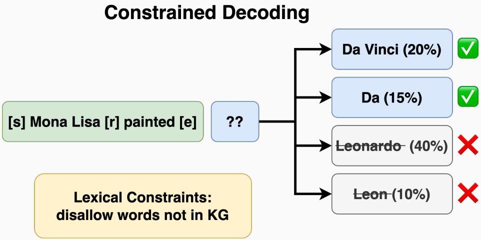


Figure: Constrainted Decoding for Information Triplet Extraction

Table of Contents

Essential Problem of Constrained Decoding: Is the next token valid?

Essential Problem

The essential problem of constrained decoding is to design a **constraint specific checker function** which can determine whether a candidate token is valid or not.
Once we have this function, we can remove the invalid tokens from the vocabulary.
By repeating this process, we can ensure that the generated sequence satisfies the constraints.
So we need a function:
`IsTokenValid`: token $->$ bool.

Constraint Type	Checker Function
Prohibited Words List	Check if the token is in the set
No Word Repetition	Count token occurrence
Valid JSON object	Rather complex, discuss in the next slide

Table: Different Constraint Types

Table of Contents

Table of Contents

Language and Grammar

1. In computer scientist's eyes, a **language** is a set of sentences that has a common structure.
2. For example, the **English language** is the set of sentences that follow the rules of **English grammar**.
3. A sentence that does not follow the rules of **English grammar** is **not** part of the **English language**.

Formal Language and Formal Grammar

In the context of computer science, a **formal language** is a set of strings that can be described by a **formal grammar**.
For example, the set of all balanced parentheses strings is a **formal language**(it has even a name, the **Dyck language**)

- `(())` is a balanced parentheses string
- `(()` is not a balanced parentheses string

What is Formal Grammar ?

Formal Grammar

- **Formalism**: Provides a systematic way to define the language structure.
- **Computational**: Enables the use of efficient algorithms to validate the language structure.

Grammar for Balanced Parentheses Language

$S \rightarrow \text{``(''}\ S\ \text{``)''} \mid \epsilon$

Use grammar to generate valid sentences

Given a grammar, we can generate all valid sentences in the language. For example, we try to generate (()) using the grammar above.

$$
\begin{aligned}
S &\Rightarrow \text{``(''}\ S\ \text{``)''} \\
&\Rightarrow \text{``(''}\ \text{``(''}\ S\ \text{``)''}\ \text{``)''} \\
&\Rightarrow \text{``(''}\ \text{``(''}\ \text{``)''}\ \text{``)''}
\end{aligned}
$$

More useful example: grammar for JSON(Simplified)

JSON

- JSON is a simple data interchange format.
- It consists of objects and arrays.
- Objects are collections of key-value pairs.

$$
\begin{aligned}
S &\rightarrow \text{object} \mid \text{array} \\
\text{object} &\rightarrow \{\ \} \mid \{\ \text{pair}\ (\ ,\ \text{pair}\ )*\ \} \\
\text{pair} &\rightarrow \text{string}\ :\ \text{value} \\
\text{array} &\rightarrow [\ ] \mid [\ \text{value}\ (\ ,\ \text{value}\ )*\ ] \\
\text{value} &\rightarrow \text{string} \mid \text{number} \mid \text{object} \mid \\
&\quad \text{array} \mid \textit{true} \mid \textit{false} \mid \textit{null} \\
\text{string} &\rightarrow \text{[a-zA-Z0-9]}\star
\end{aligned}
$$

Example

Let's derive a simple JSON object:
{"key": "value"}.
Derivation:

$$
\begin{aligned}
S &\rightarrow \text{object} \\
\text{object} &\rightarrow \{\ \text{pair}\ \} \\
\text{pair} &\rightarrow \text{string}\ :\ \text{value} \\
\text{string} &\rightarrow \text{chars} \quad (\text{where chars} \rightarrow \text{"key"}) \\
\text{value} &\rightarrow \text{string} \quad (\text{where string} \rightarrow \text{"value"})
\end{aligned}
$$

Done in 5 steps !

Grammar for Closed Information Extraction (cIE)

Grammar for Closed Information Extraction

We now try to write a grammar to describe the structure of triplets.

$$S \rightarrow (\epsilon \mid \text{“[}s\text{]”}\, \alpha\, \text{“[}r\text{]”}\, \beta\, \text{“[}o\text{]”}\, \alpha\; S)$$
$$\alpha = (\text{Entity-1} \mid \dots \mid \text{Entity-N}),$$
$$\beta = (\text{Relation-1} \mid \dots \mid \text{Relation-M})$$

Derivation

Given the sentence: *“[s] Mona Lisa [r] painted [o] Da Vinci"*

$S \Rightarrow$ “[s]” α “[r]” β “[o]” α S	Use production rule 1
$\Rightarrow$ “[s]” Mona Lisa “[r]” β “[o]” α S	Derive the subject
$\Rightarrow$ “[s]” Mona Lisa “[r]” painted “[o]” α S	Derive the relation
$\Rightarrow$ “[s]” Mona Lisa “[r]” painted “[o]” Da Vinci S	Derive the object
$\Rightarrow$ “[s]” Mona Lisa “[r]” painted “[o]” Da Vinci ϵ	Derive the empty string

Table of Contents

Fundamental Concept: Parsing

What is parsing?

Given a string and a grammar, parsing is the process of determining **whether the string is consistent with the grammar**. Example (balanced parentheses):
Input: `(()())` $\rightarrow$ **Output**: `True`
Input: `(()(` $\rightarrow$ **Output**: `False`

Takeaway from Parsing

- Any well-formed grammar(context-free grammar) **can be parsed** and the parsing be done **efficiently**.

This is theory exactly answers the fundamental question in GCD. Recall that in GCD, our fundamental task is to know if the generated string is consistent with the grammar.

From classical parsing to grammar-constrained decoding

Assume parsing works

Parsing let us know if a sentence is valid according to a grammar.
It provides a `IsSentenceValid` function: str $->$ bool.

How GCD works

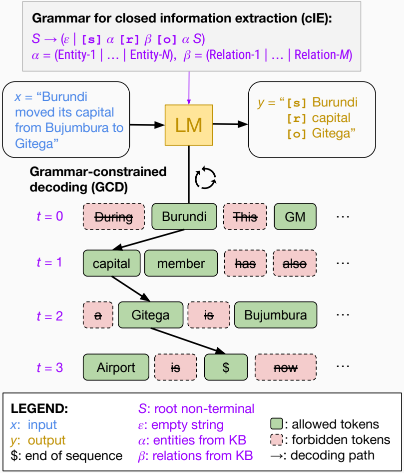
The high-level algorithm of GCD is:

1. Given an existing sentence(not necessarily complete) s
2. Get a probability distribution over the next token $P(w_i|s)$
3. For each candidate token w_i in the distribution:
4. Check if the sentence $s + w_i$ is valid according to the parser
5. If valid, add w_i to the whitelist
6. sample from the whitelist
7. Repeat until the sentence is complete

Main Components of GCD

Three Main Components of GCD:

1. A grammar G , whcih is provided by the user, task specific.
2. A GCD library, which will takes care of the parsing.
3. A LLM, which will work as an engine to generate the next token.



Specificity with Tokenization in LLM

Tokens of LLM are rather messy

In LLM, the minimal unit of generation is a token, which is a chunk of characters, usually a part of a word.

Depth	String	Tokenization	Tokens
0	""	[1]	BOS = 1
1	"[]"	[1, 5159]	␣[= 518
2	"[[]]"	[1, 518, 2636, 29962]	[] = 2636
3	"[[[]]]"	[1, 5519, 2636, 5262]	␣[[= 5519
4	"[[[[]]]]"	[1, 5519, 29961, 2636, 5262, 29962]	[[= 29961
5	"[[[[[]]]]]"	[1, 5519, 8999, 2636, 5262, 5262]	[[[= 8999
6	"[[[[[[]]]]]]"	[1, 5519, 8999, 29961, 2636, 5262, 5262, 29962]	] = 29962
7	"[[[[[[[]]]]]]]"	[1, 5519, 8999, 8999, 2636, 5262, 5262, 5262]	]] = 5262
8	"[[[[[[[[]]]]]]]]"	[1, 5519, 8999, 8999, 29961, 2636, 5262, 5262, 5262, 29962]	␣[] = 5159

Figure: Tokenization Output for Nested Brackets Using LLaMA Tokenizer

The token IDs are not aligned with the original characters

As shown in Fig. 10, tokenizing a few strings with a very simple structure, such as balanced parentheses, results in a **non-trivial** sequence of token IDs

Table of Contents

Table of Contents

Libraries with GCD support

Many libraries for controlled generation

- epfl-dlab/transformers-CFG
- guidance-ai/guidance
- outlines-dev/outlines
- sgl-project/sglang
- eth-sri/lmql
- microsoft/aici
- noamgat/lm-format-enforcer
- stanfordnlp/dspy
- jxnl/instructor
- paralleldrive/sudolang-llm-support

But only a few support CFGs

- **epfl-dlab/transformers-CFG**
- guidance-ai/guidance(Microsoft)
- outlines-dev/outlines(Hugging Face)

Differences among them:

- Grammar interface: EBNF, Custom, etc.
- Parsing algorithm: Earley, recursive descent, etc.
- Other features: Unicode support, LLM inference Engine, etc.

Table of Contents

Applications of GCD

Machine-Oriented Generation

One can categorize the applications of LLM into two main categories:

- **Human-Oriented Generation**
- **Machine-Oriented Generation**

GCD is towards the **second**.

Examples of GCD tasks

- Reliable JSON generation
- Domain-specific language generation
- Strong structured data generation

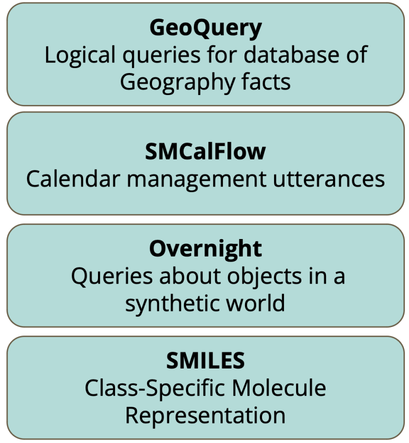


Figure: Examples of code generation tasks

Table of Contents

Thank You!

Questions?

Saibo Geng
EPFL
saibo.geng@epfl.ch

References

Lena Voita. Natural language processing course, 2024. URL https://lena-voita.github.io/nlp_course.html. Accessed: 2024-06-17.