Language Models of Code are Few-Shot Commonsense Learners

Aman Madaan[♠], **Shuyan Zhou**[♠], **Uri Alon**[♠],
Yiming Yang[♠], **Graham Neubig**[♠ †]
[♠] Language Technologies Institute, Carnegie Mellon University, USA
[†] Inspired Cognition, USA
{amadaan,shuyanzh,ualon,yiming,gneubig}@cs.cmu.edu

Abstract

We address the general task of *structured* commonsense reasoning: given a natural language input, the goal is to generate a *graph* such as an event or a reasoning-graph. To employ large language models (LMs) for this task, existing approaches "serialize" the output graph as a flat list of nodes and edges. Although feasible, these serialized graphs strongly deviate from the natural language corpora that LMs were pre-trained on, hindering LMs from generating them correctly. In this paper, we show that when we instead frame structured commonsense reasoning tasks as *code generation* tasks, pre-trained LMs of *code* are *better* structured commonsense reasoners than LMs of natural language, even when the downstream task does not involve source code at all. We demonstrate our approach across three diverse structured commonsense reasoning tasks. In all these *natural language* tasks, we show that using our approach, a *code* generation LM (CODEX) outperforms natural-LMs that are fine-tuned on the target task (*e.g.,* T5) and other strong LMs such as GPT-3 in the few-shot setting. Our code and data are available at https://github.com/madaan/CoCoGen.

1 Introduction

The growing capabilities of large pre-trained language models (LLMs) for generating text have enabled their successful application in a variety of tasks, including summarization, translation, and question-answering (Wang et al., 2019; Raffel et al., 2019; Brown et al., 2020; Chowdhery et al., 2022).

Nevertheless, while employing LLMs for natural language (NL) tasks is straightforward, a major remaining challenge is how to leverage LLMs for *structured commonsense reasoning*, including tasks such as generating event graphs (Tandon et al., 2019), reasoning graphs (Madaan et al., 2021a), scripts (Sakaguchi et al., 2021), and argument explanation graphs (Saha et al., 2021). Unlike traditional commonsense reasoning tasks such as reading comprehension or question answering, *structured* commonsense aims to generate structured output given a natural language input. This family of tasks relies on the natural language knowledge learned by the LLM, but it also requires complex structured prediction and generation.

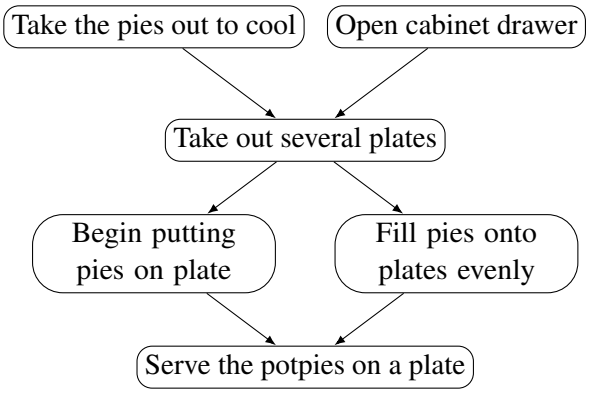
To leverage LLMs, existing structured commonsense generation models modify the *output format* of a problem. Specifically, the structure to be generated (*e.g.,* a graph or a table) is converted, or "serialized", into text. Such conversions include "flattening" the graph into a list of node pairs (Figure 1d), or into a specification language such as DOT (Figure 1c; Gansner et al., 2006).

While converting the structured output into text has shown promising results (Rajagopal et al., 2021; Madaan and Yang, 2021), LLMs struggle to generate these "unnatural" outputs: LMs are primarily pre-trained on free-form text, and these serialized structured outputs strongly diverge from the majority of the pre-training data. Further, for natural language, semantically relevant words are typically found within a small span, whereas neighboring nodes in a graph might be pushed farther apart when representing a graph as a flat string.

Thus, a language model which was trained on natural language text is likely to fail to capture the topology of the graph. Consequently, using LLMs for graph generation typically requires a large amount of task-specific training data, and their generated outputs show structural errors and semantic inconsistencies, which need to be further fixed either manually or by using a secondary downstream model (Madaan et al., 2021b).

Despite these struggles, the recent success of large-language models of *code* (Code-LLMs; Chen et al., 2021b; Xu et al., 2022) for tasks such as code generation from natural language (Austin et al., 2021; Nijkamp et al., 2022), code completion (Fried et al., 2022), and code translation (Wang

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 1384 - 1403
December 7-11, 2022



(a) The script $\mathcal{G}$

```python
class Tree:

  goal = "serve the potpies on a plate"

  def __init__(self):
    # nodes
    take_pies_out_to_cool = Node()
    open_cabinet_drawer = Node()
    take_out_several_plates = Node()
    ...
    # edges
    take_pies_out_to_cool.children =
        [take_out_several_plates]
    open_cabinet_drawer.children =
        [take_out_several_plates]
    ...
```

(b) $\mathcal{G}$ converted to Python code $\mathcal{G}_c$ using our approach

```
digraph G {
  begin -> take_pies_out_to_cool;
  begin -> open_cabinet_drawer;
  take_pies_out_to_cool ->
      take_out_several_plates;
  open_cabinet_drawer ->
      take_out_several_plates;
  take_out_several_plates ->
      begin_putting_pies_on_plates;
  begin_putting_pies_on_plates ->
      serve_potpies_on_plate;
  fill_pies_onto_plates_evenly ->
      serve_potpies_on_plate;
  serve_potpies_on_plate -> end;
}
```

(c) Straightforward encodings of the graph using the "DOT"

```
[
  (take_pies_out_to_cool,
      take_out_several_plates),
  (open_cabinet_drawer,
      take_out_several_plates),
  (take_out_several_plates,
      begin_putting_pies_on_plates),
  (take_out_several_plates,
      fill_pies_onto_plates_evenly),
  (begin_putting_pies_on_plates,
      serve_potpies_on_plate),
  (fill_pies_onto_plates_evenly,
      serve_potpies_on_plate),
  (serve_potpies_on_plate, end)
]
```

(d) Text format, or as a list of edges (node pairs)

Figure 1: An illustration of CoCoGen for the task of script generation. An input graph (1a) is typically represented using the DOT format (1c) or as a list of edges (1d), which allows modeling the graph using standard language models. These popular choices are sufficient in principle; however, these formats are loosely structured, verbose, and not common in text corpora, precluding language models from effectively generating them. In contrast, CoCoGen converts structures into Python code (1b), allowing to model them using large-scale language models of *code*.

et al., 2021), show that Code-LLMs are able to perform complex reasoning on structured data such as programs. Thus, instead of forcing LLMs of natural language (NL-LLMs) to be fine-tuned on structured commonsense data, an easier way to close the discrepancy between the pre-training data (free-form *text*) and the task-specific data (commonsense reasoning *graphs*) is to adapt LLMs that were pre-trained on *code* to structured commonsense reasoning in *natural language*.

Thus, our main insight is that *large language models of code are good structured commonsense reasoners*. Further, we show that Code-LLMs can be even better structured reasoners than NL-LLMs, when converting the desired output graph into a format similar to that observed in the code pre-training data. We call our method CoCoGen: models of **Co**de for **Co**mmonsense **Gen**eration, and it is demonstrated in Figure 1.

Our contributions are as follows:

1. We highlight the insight that Code-LLMs are better structured commonsense reasoners than NL-LLMs, when representing the desired graph prediction as code.
2. We propose CoCoGen: a method for leveraging LLMs of **co**de for structured **co**mmonsense **gen**eration.
3. We perform an extensive evaluation across three structured commonsense generation tasks and demonstrate that CoCoGen vastly outperforms NL-LLMs, either fine-tuned or few-shot tested, while controlling for the number of downstream task examples.
4. We perform a thorough ablation study, which

shows the role of data formatting, model size, and the number of few-shot examples.

2 CoCoGen: Representing Commonsense structures with code

We focus on tasks of structured commonsense generation. Each training example for such tasks is in the form $(\mathcal{T}, \mathcal{G})$, where $\mathcal{T}$ is a text input, and $\mathcal{G}$ is the structure to be generated (typically a graph). The key idea of CoCoGen is transforming an output graph $\mathcal{G}$ into a semantically equivalent program $\mathcal{G}_c$ written in a general-purpose programming language. In this work, we chose Python due to its popularity in the training data of modern Code-LLMs (Xu et al., 2022), but our approach is agnostic to the programming language. The code-transformed graphs are similar in their format to the pre-training data of Code-LLMs, and thus serve as easier to generalize training or few-shot examples than the original raw graph. CoCoGen uses Code-LLMs to generate $\mathcal{G}_c$ given $\mathcal{T}$, which we eventually convert back into the graph $\mathcal{G}$.

We use the task of script generation (ProScript, Figure 1) as a running example to motivate our method: script generation aims to create a script ($\mathcal{G}$) to achieve a given high-level goal ($\mathcal{T}$).

2.1 Converting $(\mathcal{T}, \mathcal{G})$ into Python code

We convert a $(\mathcal{T}, \mathcal{G})$ pair into a Python class or function. The general procedure involves adding the input text $\mathcal{T}$ in the beginning of the code as a class attribute or descriptive comment, and encoding the structure $\mathcal{G}$ using standard constructs for representing structure in code (*e.g.,* hashmaps, object attributes) or function calls. The goal here is to compose Python code that represents a $(\mathcal{T}, \mathcal{G})$ pair, but retains the syntax and code conventions of typical Python code.

For example, for the script generation task, we convert the $(\mathcal{T}, \mathcal{G})$ pair into a `Tree` class (Figure 1b). The goal $\mathcal{T}$ is added as class attribute (`goal`), and the script $\mathcal{G}$ is added by listing the nodes and edges separately. We first instantiate the list of nodes as objects of class `Node`. Then, the edges are added as an attribute `children` for each node (Figure 1b). For example, we instantiate the node "*Take out several plates*" as `take_out_several_plates = Node()`, and add it as a child of the node `take_pies_out_to_cool`.

While there are multiple ways of representing a training example as a Python class, we found empirically that this relatively simple format is the most effective, especially with larger models. We analyze the choice of format and its connection with the model size in Section 4.

2.2 Few-shot prompting for generating $\mathcal{G}$

We focus on large-language models of the scale of Codex (Chen et al., 2021a). Due to their prohibitively expensive cost to fine-tune, these large models are typically used in a *few-shot prompting* mode. Few-shot prompting uses k input-output examples $\{(x_i, y_i)\}_{i=1}^{k}$ to create an in-context prompt: $p = x_1 \oplus y_1 \cdot x_2 \oplus y_2 \cdot \ldots \cdot x_k \oplus y_k$, where $\oplus$ is a symbol that separates an input from its output, and $\cdot$ separates different examples.

A new (test) input x is appended to the prompt p (that is: $p \cdot x$), and $p \cdot x \oplus$ is fed to the model for completion. As found by Brown et al. (2020), large language models show impressive few-shot capabilities in generating a completion $\hat{y}$ given the input $p \cdot x \oplus$. The main question is how to construct the prompt?

In all experiments in this work, the prompt p consists of k Python classes, each representing a $(\mathcal{T}, \mathcal{G}_c)$ pair. For example, for script generation, each Python class represents a goal $\mathcal{T}$ and a script $\mathcal{G}_c$ from the training set. Given a new goal $\mathcal{T}$ for inference, a partial Python class (i.e., only specifying the goal) is created and appended to the prompt. Figure 2 shows such a partial class. Here, the code generation model is expected to complete the class by generating the definition for `Node` objects and their dependencies for the goal *make hot green tea*.

```
class Tree:
  goal = "make hot green tea."

  def __init__(self):
    # generate
```

Figure 2: CoCoGen uses a prompt consisting of k (5-10) Python classes. During inference, the test input is converted to a partial class, as shown above, appended to the prompt, and completed by a code generation model such as Codex.

In our experiments, we used Codex (Chen et al., 2021a) and found that it nearly always generates syntactically valid Python. Thus, the generated code can be easily converted back into a graph and evaluated using the dataset's standard, original,

metrics. Appendix F lists sample prompts for each of the tasks we experimented with.

3 Evaluation

We experiment with three diverse structured commonsense generation tasks: (i) script generation (PROSCRIPT, Section 3.2), (ii) entity state tracking (PROPARA, Section 3.3), and (iii) explanation graph generation (EXPLAGRAPHS, Section 3.4) Dataset details are included in Appendix D. Despite sharing the general goal of structured commonsense generation, the three tasks are quite diverse in terms of the generated output and the kind of required reasoning.

3.1 Experimental setup

Model As our main Code-LLM for CoCoGen, we experiment with the latest version of CODEX `code-davinci-002` from OpenAI[1] in few-shot prompting mode.

Baselines We experimented with the following types of baselines:

1. **Text few-shot:** Our hypothesis is that code-generation models can be repurposed to generate structured output better. Thus, natural baselines for our approach are NL-LLMs – language models trained on natural language corpus. We experiment with the latest versions of CURIE (`text-curie-001`) and DAVINCI (`text-davinci-002`), the two GPT-3 based models by OpenAI (Brown et al., 2020). For both these models, the prompt consists of $(\mathcal{T}, \mathcal{G})$ examples, where $\mathcal{G}$ is simply flattened into a string (as in Figure 1c). DAVINCI is estimated to be much larger in size than CURIE, as our experiments also reveal (Appendix A). DAVINCI, popularly known as GPT-3, is the strongest text-generation model available through OpenAI APIs.[2]

2. **Fine-tuning:** we fine-tune a T5-large model for EXPLAGRAPHS, and use the results from Sakaguchi et al. (2021) on T5-xxl for PROSCRIPT tasks. In contrast to the few-shot setup where the model only has access to a few examples, fine-tuned models observe the *entire* training data of the downstream task.

[1] As of June 2022

[2] `https://beta.openai.com/docs/models/gpt-3`

Choice of prompt We created the prompt p by randomly sampling k examples from the training set. As all models have a bounded input size (*e.g.,* 4096 tokens for CODEX `code-davinci-002` and 4000 for GPT-3 `text-davinci-002`), the exact value of k is task dependent: more examples can fit in a prompt in tasks where $(\mathcal{T}, \mathcal{G})$ is short. In our experiments, k varies between 5 and 30, and the GPT-3 baseline is always fairly given the same prompts as CODEX. To control for the variance caused by the specific examples selected into p, we repeat each experiment with at least 3 different prompts, and report the average. We report the mean and standard deviations in Appendix I.

CoCoGen: We use CoCoGen to refer to setups where a CODEX is used with a Python prompt. In Section 4, we also experiment with dynamically creating a prompt for each input example, using a NL-LLMs with code prompts, and using Code-LLMs with textual prompts.

3.2 Script generation: PROSCRIPT

Given a high-level goal (*e.g., bake a cake*), the goal of script generation is to generate a graph where each node is an action, and edges capture dependency between the actions (Figure 1a). We use the PROSCRIPT (Sakaguchi et al., 2021) dataset, where the scripts are directed acyclic graphs, which were collected from a diverse range of sources including ROCStories (Mostafazadeh et al., 2016), Descript (Wanzare et al., 2016), and Virtual home (Puig et al., 2018).

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a script for a high-level goal $\mathcal{T}$ with node and edge sets $\mathcal{V}$ and $\mathcal{E}$, respectively. Following Sakaguchi et al. (2021), we experiment with two sub-tasks: (i) **script generation:** generating the entire script $\mathcal{G}(\mathcal{V}, \mathcal{E})$ given a goal $\mathcal{T}$, and (ii) **edge prediction:** predicting the edge set $\mathcal{E}$ **given** the nodes $\mathcal{V}$ and the goal $\mathcal{T}$.

Figure 1 shows an input-output example from PROSCRIPT, and our conversion of the graph into Python code: we convert each node $v \in \mathcal{V}$ into an instance of a `Node` class; we create the edges by adding `children` attribute for each of the nodes. Additional examples are present in Figure 6

To represent a sample for edge prediction, we list the nodes in a random order (specified after the comment `# nodes` in Figure 1b). The model then completes the class by generating the code below the comment `# edges`.

	BLEU	ROUGE-L	BLEURT	ISO	GED	Avg(d)	Avg($\|V\|$)	Avg($\|E\|$)
$\mathcal{G}$ (reference graph)		-	-	1.00	0.00	1.84	7.41	6.80
T5 (fine-tuned)	23.80	35.50	-0.31	0.51	1.89	**1.79**	7.46	**6.70**
CURIE (15)	11.40	27.00	-0.41	0.15	3.92	1.47	8.09	6.16
DAVINCI (15)	23.11	36.51	-0.27	**0.64**	**1.44**	1.74	7.58	6.59
COCOGEN (15)	**25.24**	**38.28**	**-0.26**	0.53	2.10	**1.79**	**7.44**	**6.70**

Table 1: Semantic and structural metrics for the script generation task on PROSCRIPT. T5 is fine-tuned on the entire dataset, while the few-shot models (CURIE, DAVINCI, CODEX) use 15 examples in the prompt.

	Method	*prec*	*rec*	F_1
fine-tuned	T5 (100)	52.26	52.91	51.89
	T5 (1k)	60.55	61.24	60.15
	T5 (4k)	**75.71**	**75.93**	**75.72**
few-shot	CURIE (15)	10.19	11.61	10.62
	DAVINCI (15)	50.62	49.30	48.92
	COCOGEN (15)	**57.34**	**55.44**	**56.24**

Table 2: Precision, recall, and F_1 for PROSCRIPT edge-prediction task. COCOGEN with 15 samples outperforms strong few-shot models, and T5 trained on 1k samples.

Script Generation metrics We denote the script that was generated by the model as $\hat{\mathcal{G}}$, and evaluate $\hat{\mathcal{G}}$ vs. $\mathcal{G}$ for both semantic and structural similarity. To evaluate semantic similarity, we use BLEU, ROUGE-L, and the learned metric BLEURT to determine the content overlap. Following Sakaguchi et al. (2021), we use the following metrics for structural evaluation of generated scripts:

- Graph edit distance (GED): the number of required edits (node/edge removal/additions) to transform $\hat{\mathcal{G}}$ to $\mathcal{G}$ (Abu-Aisheh et al., 2015);
- Graph isomorphism (ISO; Cordella et al., 2001): determines whether $\hat{\mathcal{G}}$ and $\mathcal{G}$ are isomorphic based on their structure, disregarding the textual content of nodes;
- Graph size: average number of nodes and edges, $(|\mathcal{G}(V)|, |\mathcal{G}(E)|, |\hat{\mathcal{G}}(V)|, |\hat{\mathcal{G}}(V))$ and the average degree $(\mathrm{d}(\mathcal{G}(V)))$, where the high-level goal is for $\hat{\mathcal{G}}$ to have as close measures to $\mathcal{G}$ as possible.

Edge Prediction metrics For the edge prediction task, the set of nodes is given, and the goal is to predict the edges between them. Following Sakaguchi et al. (2021), we measure precision, recall, and F_1 comparing the true and predicted edges. Specifically, $p = \frac{|\boldsymbol{E} \cap \hat{\boldsymbol{E}}|}{|\hat{\boldsymbol{E}}|}$, $r = \frac{|\boldsymbol{E} \cup \hat{\boldsymbol{E}}|}{|\boldsymbol{E}|}$, and $F_1 = \frac{2pr}{p+r}$.

Results Table 1 shows the results for script generation. The main results are that COCOGEN (based on CODEX), with just 15 prompt examples, outperforms the fine-tuned model T5 which has been fine-tuned on *all* 3500 samples. Further, COCOGEN outperforms the few-shot NL-LM CURIE across all semantic metrics and structural metrics. COCOGEN outperforms DAVINCI across all semantic metrics, while DAVINCI performs slightly better in two structural metrics.

Table 2 shows the results for edge prediction: COCOGEN significantly outperforms the NL-LLMs CURIE and DAVINCI. When comparing with T5, which was fine-tuned, COCOGEN with only 15 examples outperforms the fine-tuned T5 which was fine-tuned on 100 examples. The impressive performance in the edge-generation task allows us to highlight the better ability of COCOGEN in capturing structure, while factoring out all models' ability to generate the NL content.

3.3 Entity state tracking: PROPARA

The text inputs $\mathcal{T}$ of entity state tracking are a sequence of actions in natural language about a particular topic (*e.g.,* photosynthesis) and a collection of entities (*e.g.,* water). The goal is to predict the state of each entity after the executions of an action. We use the PROPARA dataset (Dalvi et al., 2018) as the test-bed for this task.

We construct the Python code $\mathcal{G}_c$ as follows, and an example is shown in Figure 3. First, we define the `main` function and list all n actions as comments inside the `main` function. Second, we create k variables named as `state_k` where k is the number of participants of the topic. The semantics of each variable is described in the comments as well. Finally, to represent the state change after each step, we define n functions where each function corresponds to an action. We additionally define an `init` function to represent the initial-

Action	Entity		
	water	light	CO2
Initial states	soil	sun	-
Roots absorb water from soil	roots	sun	?
The water flows to the leaf	leaf	sun	?

```
def main():
    # init
    # roots absorb water from soil
    # the water flows to the leaf
    # state_0 tracks the location/state water
    # state_1 tracks the location/state light
    # state_2 tracks the location/state CO2
    def init():
        state_0 = "soil"
        state_1 = "sun"
        state_2 = None
    def roots_absorb_water_from_soil():
        state_0 = "roots"
        state_1 = "sun"
        state_2 = "UNK"
    def water_flows_to_leaf():
        state_0 = "leaf"
        state_1 = "sun"
        state_2 = "UNK"
```

Figure 3: A PROPARA example (left) and its corresponding Python code (right). We use a string to represent a concrete location (*e.g.,* `soil`), `UNK` to represent an unknown location, and `None` to represent non-existence.

Model	*prec*	*rec*	F_1
CURIE	**95.1**	22.3	36.1
DAVINCI	75.5	47.1	58.0
COCOGEN	80.0	**53.6**	**63.0**

Table 3: 3-shots results on PROPARA. All numbers are averaged among five runs with different randomly sampled prompts. COCOGEN significantly outperforms CURIE and DAVINCI.

ization of entity states. Inside each function, the value of each variable tells the state of the corresponding entity after the execution of that action. Given a new test example where only the actions and the entities are give, we construct the input string until the `init` function, and we append it to the few-shot prompts for predictions.

Metrics We follow Dalvi et al. (2018) and measure precision, recall and F_1 score of the predicted entity states. We randomly sampled three examples from the training set as the few-shot prompt.

Results As shown in Table 3, COCOGEN achieves a significantly better F_1 score than DAVINCI. Across the five prompts, COCOGEN achieves 5.0 higher F_1 than DAVINCI on average. In addition, COCOGEN yields stronger performance than CURIE, achieving F_1 of 63.0, which is 74% higher than CURIE (36.1).[3]

In PROPARA, COCOGEN will be ranked 6^{th} on the leaderboard.[4] However, all the methods above COCOGEN require fine-tuning on the entire training corpus. In contrast, COCOGEN uses only *3 examples* in the prompt and has a gap of less than 10 F_1 points vs. the current state-of-the-art (Ma et al., 2022). In the few-shot settings, COCOGEN is state-of-the-art in PROPARA.

3.4 Argument graph generation: EXPLAGRAPHS

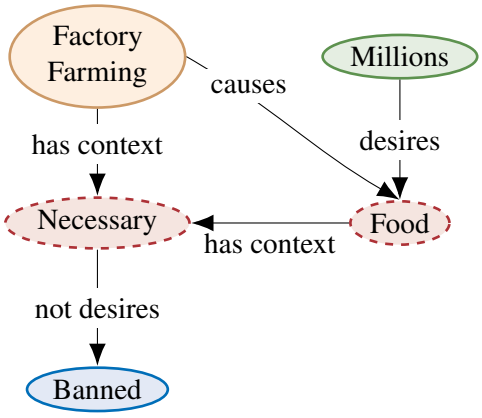
Given a belief (*e.g., factory farming should not be banned*) and an argument (*e.g., factory farming feeds millions*), the goal of this task is to generate a graph that uses the argument to either *support* or *counter* the belief (Saha et al., 2021). The text input to the task is thus a tuple of (*belief*, *argument*, *"supports"/"counters"*), and the structured output is an explanation graph (Figure 4).

We use the EXPLAGRAPHS dataset for this task (Saha et al., 2021). Since we focus on generating the argument graph, we take the stance as given and use the stance that was predicted by a stance prediction model released by Saha et al..

To convert an EXPLAGRAPHS to Python, the belief, argument, and stance are instantiated as string variables. Next, we define the graph structure by specifying the edges. Unlike PROSCRIPT, the edges in EXPLAGRAPHS are typed. Thus, each edge is added as an `add_edge(source, edge_type, destination)` function call. We also list the starting nodes in a list instantiated

[3] CURIE often failed to produce output with the desired format, and thus its high precision and low recall.

[4] As of 10/11/2022, https://leaderboard.allenai.org/propara/submissions/public




```python
class ExplanationDAG:

  def __init__(self):
    belief = "factory farming should not be
        banned."
    argument = "Factory farming feeds millions."
    stance = "support"

    # Edges
    begin = ["factory farming", "millions"]
    add_edge("factory farming", "causes", "food")
    add_edge("factory farming", "has context",
        "necessary")
    add_edge("food", "has context", "necessary")
    add_edge("necessary", "not desires", "banned")
    add_edge("millions", "desires", "food")
```

Figure 4: An explanation graph (left) and the corresponding Python code (right)

		StCA (↑)	SeCA (↑)	G-BS (↑)	GED (↓)	EA (↑)
fine-tuned	T5 (150)	12.56	6.03	9.54	91.06	7.77
	T5 (1500)	38.19	21.86	29.37	73.09	23.41
	T5 (2500)	43.22	**29.65**	33.71	69.14	**26.38**
few-shot	CURIE (30)	5.03	1.26	3.95	96.74	2.60
	DAVINCI (30)	23.62	10.80	18.46	83.83	11.84
	COCOGEN (30)	**45.20**	**23.74**	**34.68**	**68.76**	**23.58**

Table 4: Results for EXPLAGRAPHS (eval split). COCOGEN with only 30 examples outperforms the T5 model which was fine-tuned on 1500 examples, across all metrics.

with a `begin` variable (Figure 4). Given a test example, we construct the input until the line of `# Edges` and let a model complete the remaining.

Metrics We use the metrics defined by Saha et al. (2021) (see Section 6 of Saha et al. (2021) for a detailed description of the mechanisms used to calculate these metrics):

- Structural accuracy (StCA): fraction of graphs that are connected DAGs with two concepts each from belief and the argument.
- Semantic correctness (SeCA): a learned metric that evaluates if the correct stance is inferred from a (belief, graph) pair.
- G-BERTScore (G-BS): measures BERTscore-(Zhang et al., 2020) based overlap between generated and reference edges .
- GED (GED): avg. edits required to transform the generated graph to the reference graph.
- Edge importance accuracy (EA): measures the importance of each edge in predicting the target stance. A high EA implies that each edge in the generated output contains unique semantic information, and removing any edge will hurt.

Results Table 4 shows that COCOGEN with only 30 examples outperforms the T5 model that was fine-tuned using 1500 examples, across all metrics. Further, COCOGEN outperforms the NL-LLMs DAVINCI and CURIE with a text-prompt across all metrics by about 50%-100%.

4 Analysis

In this section, we analyze the effect of three important components of COCOGEN: (i) the contributions of Code-LLMs and structured prompt $\mathcal{G}_c$; (ii) the selection of examples in the in-context prompt; and (iii) the design of the Python class.

Structured Prompts vs. Code-LLMs Which component is more important, using a Code-LLMs or the structured formatting of the input as code? To answer this, we experimented with a text prompt with a Code-LLM CODEX, and a code prompt with an NL-LLM, DAVINCI. Table 5 shows that both contributions are indeed important: performance improves for the NL-LLM DAVINCI both when we use a code prompt, *and* when we use a Code-LLM. However when using both a Code-LLM and a code prompt – the improvement is greater than the sum

	EXPLAGRAPHS					PROSCRIPT (edge-prediction)		
	StCA (↑)	SeCA (↑)	G-BS (↑)	GED (↓)	EA (↑)	p	r	F_1
DAVINCI + text	**33.16**	7.14	25.91	77.45	15.9	43.06	41.52	43.06
DAVINCI + code	33.00	**15.37**	**26.15**	**76.91**	**16.68**	**50.62**	**48.27**	**49.3**
CODEX + text	38.02	18.23	29.46	73.68	19.54	45.31	43.95	44.47
COCOGEN (CODEX + code)	**45.20**	**23.74**	**34.68**	**68.76**	**23.58**	**57.34**	**55.44**	**56.52**

Table 5: Teasing apart the contributions of a code generation model and a structured prompt. The experiments show that both are helpful. DAVINCI, a text generation model, shows marginal improvements with a code prompt (top two rows). Similarly, CODEX, a code generation model, significantly benefits from a code prompt. Overall, CODEX with code prompt performs better than the alternatives, across all metrics.

of each of these solely.

Dynamic prompt selection The prompts for all experiments in Section 3 were created by *random* sampling of examples from the training set. Specifically, a set of k $(\mathcal{T}, \mathcal{G})$ pairs are sampled and concatenated into a prompt p, which we used for inference over all examples x_{test} in the test set. As an alternative to creating prompts, there is now a growing interest in customizing the in-context examples each example x_{test}. Popular techniques typically train a retriever, which is used to fetch the closest examples (Liu et al., 2021; Rubin et al., 2021; Poesia et al., 2021). We also experimented with such *dynamic* creation of the prompt, that depends on the particular test example. Specifically, following Poesia et al. (2021), we performed knowledge similarity tuning (KST): we trained a retriever model to retrieve the k closest examples for a given input.

Setup	p	r	F_1
COCOGEN	57.34	55.44	56.52
COCOGEN + KST	**67.11**	**64.57**	**65.71**

Table 6: Our retrieval mechanism is highly effective for edge prediction: the closest examples are from similar domains and the model is able to leverage the information for better performance.

The results indicate that the efficacy of dynamic prompts depends on both the training data and task. In the edge-prediction sub-task of PROSCRIPT, edges between events in similar scripts are helpful, and Table 6 shows that the model was able to effectively leverage this information. In the script generation sub-task of PROSCRIPT, Table 8 shows that KST provides gains as well (Appendix B).

In EXPLAGRAPHS, we observed that the training data had multiple examples which were nearly identical, and thus dynamically created prompts often included such duplicate examples, effectively reducing diversity and prompt size (Table 9).

Python Formatting We performed an extensive study of the effect of the Python format on the downstream task performance in Appendix G. We find that: (i) there are no clear task-agnostic Python class designs that work uniformly well; and that (ii) larger models are less sensitive to prompt (Python class) design. In general, our approach benefits the most from code formats that as similar as possible to the conventions of typical code.

Human evaluation We conduct human evaluation of the graphs generated by COCOGEN and DAVINCI to supplement automated metrics. The results (Appendix C) indicate that human evaluation is closely correlated with the automated metrics: for EXPLAGRAPHS, graphs generated by COCOGEN are found to be more relevant and correct. For PROSCRIPT generation, both DAVINCI and COCOGEN have complementary strengths, but COCOGEN is generally better in terms of relevance.

5 Related work

Structured commonsense reasoning using LLMs Existing methods for structured commonsense generation typically flatten the output graphs as strings (Madaan and Yang, 2021; Madaan et al., 2021a; Sakaguchi et al., 2021). Consequently, these methods struggle with generation of well-formed outputs (Sakaguchi et al., 2021; Madaan et al., 2021b). In contrast, we address the problem of structured generation by (1) translating the task into Python code, and (2) generating code using large-code generation models.

Code representation for procedural knowledge reasoning Programs inherently encode rich structures, and they can efficiently represent task procedures. Existing works leverage the control-flows, nested functions and API calls of a programming language such as Python to control the situated agents in the embodied environment (Sun et al.,

2019; Zhou et al., 2022; Singh et al., 2022). In this work, we go beyond these procedural tasks and show the effectiveness of using Code-LLMs on broader structured commonsense tasks.

Adapting Code-LLMs for reasoning As code-generation models (Code-LLMs) are getting increasingly popular, there is a growing interest in adapting them for a wide range reasoning tasks. Wu et al. (2022) use CODEX and PaLM (Chowdhery et al., 2022) for converting mathematical statements written in natural language into a formal structure that can be used for theorem provers, with moderate success. The task is challenging, as it involves understanding the concepts used in the theorem (*e.g.,* set of real numbers) and the complex relationship between them. Our work is similar in spirit to Wu et al. (2022), and seeks to leverage the dual abilities of Code-LLMs for text and symbolic reasoning. However, differently from their work, we close the gap between the pre-training data and our tasks by translating our output into Python code. As our experiments show, this step is crucial in outperforming text-only and fine-tuned models. To the best of our knowledge, our work is the first to transform a natural-language reasoning problem into code to successfully leverage code generation methods.

Symbolic reasoning using LLMs The use of programming languages like LISP (Tanimoto, 1987) and Prolog (Colmerauer and Roussel, 1996) to process natural language has a long history in AI. However, the recent progress in large language models has obviated the need for specialized methods for symbolic processing. Cobbe et al. (2021) and Chowdhery et al. (2022) address middle-school level algebra problem solving using large-language models in a few-shot setup. These problems require a model to understand the order in which a set of operations should be performed over symbols (typically small integers). In contrast, structured commonsense reasoning requires *broader* information than supplied in the prompt, while utilizing the models' structural generation capabilities for generating output effectively. Thus, the tasks in our work push a model to use *both* its reasoning and symbolic manipulation capabilities.

6 Conclusion

We present the first work to employ large language models of code for structured commonsense generation. By converting the output commonsense structures to Python code, COCOGEN provides a simple and effective method for leveraging the code-generation abilities of Code-LLMs for structured generation. These results open a promising direction for structural commonsense reasoning. We believe that the principles and the methods presented in this paper are applicable to additional NLP tasks that require "language understanding" *and* structured prediction.

Acknowledgments

We thank Kaixin Ma, Keisuke Sakaguchi and Niket Tandon for thoughtful discussion and helping with PROSCRIPT datasets and the anonymous reviewers for valuable feedback. This material is partly based on research sponsored in part by the Air Force Research Laboratory under agreement number FA8750-19-2-0200. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government. This project was also partially supported by a gift from AWS AI.

Limitations

Some experiments in this work are performed with language models that are not open-sourced, namely DAVINCI, CURIE, and CODEX. Existing documentation (Brown et al., 2020; Chen et al., 2021b) does not fully describe the details of these models, such as the pretraining corpus, model size, and model biases. Therefore, we can only provide educational guesses on these details (analysis in Appendix A). In addition, even though CODEX is free to use for research as of June 2022, we are unsure whether the research community will continue to have free access in the future. Nonetheless, we release our code and model outputs to ensure the reproducibility of our work. Furthermore, in cases where the models we experiment with reveal any issue, the publicly available code will allow future investigations.

Another limitation of our work is that we exclusively experiment with datasets in English. Exploring the efficacy of structured generation methods in cross-lingual settings is an interesting and important future work.

References

Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An exact graph edit distance algorithm for solving pattern recognition problems. In *An exact graph edit distance algorithm for solving pattern recognition problems*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374 [cs]*. ArXiv: 2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Alain Colmerauer and Philippe Roussel. 1996. The birth of prolog. In *History of programming languages—II*, pages 331–367.

Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2001. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159.

Bhavana Dalvi, Lifu Huang, Niket Tandon, Wen-tau Yih, and Peter Clark. 2018. Tracking state changes in procedural text: a challenge dataset and models for process paragraph comprehension. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1595–1604, New Orleans, Louisiana. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Emden Gansner, Eleftherios Koutsofios, and Stephen North. 2006. Drawing graphs with dot.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? *arXiv:2101.06804 [cs]*. ArXiv: 2101.06804.

Kaixin Ma, Filip Ilievski, Jonathan Francis, Eric Nyberg, and Alessandro Oltramari. 2022. Coalescing global and local information for procedural text understanding. *arXiv preprint arXiv:2208.12848*.

Aman Madaan, Dheeraj Rajagopal, Niket Tandon, Yiming Yang, and Eduard Hovy. 2021a. Could you give me a hint ? generating inference graphs for defeasible reasoning. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 5138–5147, Online. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Dheeraj Rajagopal, Peter Clark, Yiming Yang, and Eduard Hovy. 2021b. Think about it! improving defeasible reasoning by first modeling the question scenario. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6291–6310.

Aman Madaan and Yiming Yang. 2021. Neural language modeling for contextualized temporal graph generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 864–881, Online. Association for Computational Linguistics.

Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. 2016. A corpus and evaluation framework for deeper understanding of commonsense stories. *arXiv preprint arXiv:1604.01696*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2021. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. 2018. Virtualhome: Simulating household activities via programs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 8494–8502. IEEE Computer Society.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Dheeraj Rajagopal, Aman Madaan, Niket Tandon, Yiming Yang, Shrimai Prabhumoye, Abhilasha Ravichander, Peter Clark, and Eduard Hovy. 2021. Curie: An iterative querying approach for reasoning about situations.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning To Retrieve Prompts for In-Context Learning. *arXiv:2112.08633 [cs]*. ArXiv: 2112.08633.

Swarnadeep Saha, Prateek Yadav, Lisa Bauer, and Mohit Bansal. 2021. ExplaGraphs: An Explanation Graph Generation Task for Structured Commonsense Reasoning. *arXiv:2104.07644 [cs]*. ArXiv: 2104.07644.

Keisuke Sakaguchi, Chandra Bhagavatula, Ronan Le Bras, Niket Tandon, Peter Clark, and Yejin Choi. 2021. proScript: Partially Ordered Scripts Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2138–2149, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2022. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*.

Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. 2019. Program guided agent. In *International Conference on Learning Representations*.

Niket Tandon, Bhavana Dalvi, Keisuke Sakaguchi, Peter Clark, and Antoine Bosselut. 2019. WIQA: A dataset for "what if..." reasoning over procedural text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6076–6085, Hong Kong, China. Association for Computational Linguistics.

Steven L Tanimoto. 1987. *The elements of artificial intelligence: an introduction using LISP*. Computer Science Press, Inc.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 3261–3275.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Lilian D. A. Wanzare, Alessandra Zarcone, Stefan Thater, and Manfred Pinkal. 2016. A crowdsourced database of event sequence descriptions for the acquisition of high-quality script knowledge. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 3494–3501, Portorož, Slovenia. European Language Resources Association (ELRA).

Yuhuai Wu, Albert Q Jiang, Wenda Li, Markus N Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *arXiv preprint arXiv:2205.12615*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Shuyan Zhou, Pengcheng Yin, and Graham Neubig. 2022. Hierarchical control of situated agents through natural language. In *Workshop on Structured and Unstructured Knowledge Integration (SUKI)*, Seattle, USA.

A Few-shot models size estimates

As OpenAI has not released any details of the size of their few-shot models, we estimate the relative strengths and weaknesses on code and text generation by calculating the average loss per token. To calculate the avg. loss of each of these models on code, we use the implementation provided by Xu et al. (2022).[5] The perplexity on text corpus was evaluated on 30 random wikipedia pages from Wikiplots[6] following a similar procedure The structure and text generation capabilities of the models are apparent from the results in Table 7; DAVINCI outperforms CODEX on text generation but is worse on code-generation and vice-versa. CURIE underperforms both DAVINCI and CODEX significantly. Importantly, these results show that CODEX and DAVINCI are of comparable capacities, making their comparison fair.

Model	CODE	TEXT
CODEX	**0.46**	2.71
DAVINCI	0.63	**2.25**
CURIE	1.17	3.32

Table 7: Average loss per token of the three few-shot models used in this work. TEXT refers to the average loss over 30 Wikipedia pages, and CODE is the loss over Python scripts in the evaluation split of Polycoder.

B Dynamic prompt Creation

As an alternative to creating prompts, there is now a growing interest in customizing the in-context examples each example $\mathcal{T}_{test}$. Popular techniques typically train a retriever, which is used to fetch the examples in the training set that are closest to $\mathcal{T}_{test}$ (Liu et al., 2021; Rubin et al., 2021; Poesia et al., 2021).

Specifically Poesia et al. (2021) train a retriever with a *target-similarity tuning* (TST) objective over a corpus of $\mathcal{D}$ of (x, y) examples. TST learns an embedding function f such that for a pair of examples (x_i, y_i) and (x_j, y_j), if $y_i \sim y_j \implies f(x_i) \sim f(x_j)$. For a new x, $f(x)$ is used to retrieve the closest examples from $\mathcal{D}$.

We follow Poesia et al. (2021), and train a knowledge-similarity tuner (KST). We use mpnet-base[7] with SentenceTransformers (Reimers and Gurevych, 2019) to fine-tune a retrieval function f by minimizing the following loss:

$$L_\theta = \left(\cos(f_\theta(\mathcal{T}_i), f_\theta(\mathcal{T}_j)) - \texttt{sim}(\mathcal{G}_i, \mathcal{G}_j)\right)^2 \quad (1)$$

where f_θ is parameterized using a transformer.

Results on using KST with PROSCRIPT (Table 8) and EXPLAGRAPHS (Table 9). While KST is highly effective for edge-prediction 6, the results are mixed for EXPLAGRAPHS and PROSCRIPT. For PROSCRIPT, KST yields marginal gains. However, for EXPLAGRAPHS, a number of training examples have overlapping theme (Table 10), and thus creating a prompt dynamically reduces the effective information in the prompt.

C Human Evaluation

Out of the four tasks used in this work, PROSCRIPT edge prediction and PROPARA have only one possible correct value. Thus, following prior work, we report the automated, standard metrics for these tasks. For EXPLAGRAPHS, we use model-based metrics proposed by Saha et al. (2021), which were found to have a high correlation with human judgments. For PROSCRIPT graph generation, we conducted an exhaustive automated evaluation that separately scores the correctness of the nodes and the correctness of the edges.

However, automated metrics are limited in their ability to evaluate model-generated output. Thus, to further investigate the quality of outputs, we conduct a human evaluation to compare the outputs generated by COCOGEN and DAVINCI. We sampled 20 examples, and three of the authors performed the evaluation. Annotators were shown two graphs (generated by COCOGEN and DAVINCI) and were asked to select one they thought was better regarding relevance and correctness. The selection for each criterion was made independently: the same graph could The annotations were done separately: the same graph could have more relevant nodes (higher relevance) but may not be correct. The identity of the model that generated each graph (COCOGEN or DAVINCI) was shuffled and unknown to the evaluators.

The results in Table 11 indicate that human evaluation is closely correlated with the automated metrics: for EXPLAGRAPHS, annotators found the

[5] https://github.com/VHellendoorn/Code-LMs#evaluation

[6] https://github.com/markriedl/WikiPlots

[7] https://huggingface.co/microsoft/mpnet-base

	ISO	GED	Avg(d)	Avg($\|\boldsymbol{V}\|$)	Avg($\|\boldsymbol{E}\|$)	BLEU	ROUGE-L	BLEURT	
$\mathcal{G}$	1.0	0.0	1.84	7.41	6.8	-	-	-	-
CoCoGen + 002 (15)	0.53	2.1	1.79	**7.44**	**6.7**	25.24	38.28	-0.26	
CoCoGen + 002 (15) + KST	0.52	1.99	**1.8**	7.45	**6.7**	**25.4**	**38.4**	**-0.25**	

Table 8: KST on PROSCRIPT generation: Dynamically creating a prompt leads to marginal improvements.

	StCA (↑)	SeCA (↑)	G-BS (↑)	GED (↓)	EA (↑)
CoCoGen + 002	**45.2**	**23.74**	**34.68**	**68.76**	**23.58**
CoCoGen + 002 + KST	37.47	18.46	29.41	73.76	19.15

Table 9: KST on EXPLAGRAPHS: We find that EXPLAGRAPHS contains multiple examples that are similar to each other in the training set. Thus, dynamically creating a prompt by selecting examples that are closest to the input actually hurts performance.

belief: all religions need to be respected, and able to practice. *argument*: religion is behind many wars.
belief: every religion needs to be respected and allowed to be practiced. *argument*: religion is behind most wars.
belief: school prayer should not be allowed. *argument*: many people would prefer to keep religion out of their lives.
belief: people should follow whichever religion they choose. *argument*: this country has freedom of religion.
belief: people are free to practice the religion they choose*argument*: society's right to be free to practice religion should not be limited.
belief: the church of scientology should be allowed, because everyone has a right to follow their religion. *argument*: the church of scientology doesn't have a religious doctrine.
belief: we should avoid discussing religion in schools.*argument*: some schools are religious in nature, and have regular discussions on the topic.
belief: freedom of religion is paramount. *argument*: not all religions are worth it.
belief: people don't follow the same religion. *argument*: the world has many different religions.
belief: people should follow whatever religion they desire. *argument*: people have the right to adhere to the religion of their choice.
belief: people should follow whichever religion they choose. *argument*: some religions are better than others.
belief: people should be able to practice whatever religion they choose. *argument*: some religions are not okay to pursue.
belief: students have a right to express themselves any way possible, including faith. *argument*: religion is a personal choice.
belief: people should be able to do missionary work if they desire. *argument*: people should have right to missionary work.
belief: students are free to express faith. *argument*: one should go to church to express their religious beliefs.

Table 10: The closest examples in the training set corresponding to the test input: *belief*: religion causes many fights. and *argument*: *There would be less fights without religious conflicts.*. As the table shows, the examples are overlapping which reduces the diversity in the prompt, effectively reducing the number of examples in a prompt creating using nearest neighbors (Section 4.

	Dataset	CoCoGen	DAVINCI	No preference
Relevance	EXPLAGRAPHS	28.3%	16.7%	46.7%
	PROSCRIPT (script generation)	26.7%	18.3%	55%
Correctness	EXPLAGRAPHS	38.3%	18.3%	31.7%
	PROSCRIPT (script generation)	26.7%	23.3%	50%

Table 11: Human evaluation of graphs generated by CoCoGen and DAVINCI. The evaluators were shown graphs generated by CoCoGen and DAVINCI, and were asked to select one that is more relevant to the input and correct. In case of no preference, the evaluators could pick the No preference. The table shows the % of times graphs from each model were preferred.

graphs generated by CoCoGen to be more relevant and correct. We find that DAVINCI often fails to recover semantic relations between nodes in the argument graphs. For example, consider a belief (B) *urbanization harms natural habitats for the animals in the world*. We want to generate a graph that can **counter** this belief with the argument (A) *urbanization causes increase in jobs.*

For the same prompt, CoCoGen generated *(urbanization; causes; increase in jobs); (increase in jobs; has context; good); (good; not capable of; harms)* whereas DAVINCI generated *(jobs; not harms; natural habitats)* → *(natural habitats; not part of; animals)*. Note that DAVINCI successfully recovered relevant events ("natural habitat" "animals") but arranged them in incorrect relations. For PROSCRIPT, the human evaluation shows that CoCoGen and DAVINCI have complementary strengths, while CoCoGen generally produces more relevant and correct outputs.

D Dataset statistics

Dataset statistics are shown in Table 12. The test split for EXPLAGRAPHS is not available, so we evaluate on the validation split. For PROSCRIPT, we obtained the test splits from the authors.

Corpus	#Train	#Val	#Test
PROSCRIPT	3252	1085	2077
PROPARA	387	43	54
EXPLAGRAPHS	2368	398	-

Table 12: Corpus Statistics for the tasks used in this work.

E Sample outputs

Sample outputs from CoCoGen for all the tasks are located at https://github.com/madaan/CoCoGen/tree/main/outputs. Representative examples from each task are presented in Figure 5. Surprisingly, CoCoGen (Codex with a Python prompt) generates syntactically valid Python graphs that are similar to the task graphs/tables in nearly 100% of the cases.

F Prompts

The prompts for each tasks are present at this anonymous URL:

1. PROSCRIPT script-generation: https://github.com/madaan/CoCoGen/tree/main/data/proscript_script_generation/prompt.txt
2. PROSCRIPT edge-prediction: https://github.com/madaan/CoCoGen/tree/main/data/proscript_edge_prediction/prompt.txt
3. PROPARA: https://github.com/madaan/CoCoGen/tree/main/data/explagraphs/prompt.txt
4. EXPLAGRAPHS: https://github.com/madaan/CoCoGen/tree/main/data/explagraphs/prompt.txt

These prompts are also present in the attached supplementary material, and can be found in the `data` folder under respective task sub-directories.

G Designing Python class for a structured task

Figure 7 shows three different designs for Explagraphs. For PROSCRIPT, the various formats include representing proscript as a Networkx[8] class (8), DOT-like class 9, and as a Tree (10).

H Impact of Model size

The Codex model released by OpenAI is available in two versions[9]: `code-davinci-001` and `code-davinci-002`. While the exact sizes of the models are unknown because of their proprietary nature, OpenAI API states that `code-davinci-002` is the *Most capable Codex model* Tables 16 and **??** compares CoCoGen +`code-davinci-001` with CoCoGen +`code-davinci-002`. Note that both `code-davinci-001` and `code-davinci-002` can fit 4000 tokens, so the number of in-context examples was identical for the two settings. The results show that for identical prompts, CoCoGen +`code-davinci-002` vastly outperforms CoCoGen +`code-davinci-001`, showing the importance of having a better underlying code generation model.

[8] https://networkx.org/

[9] as of June 2022

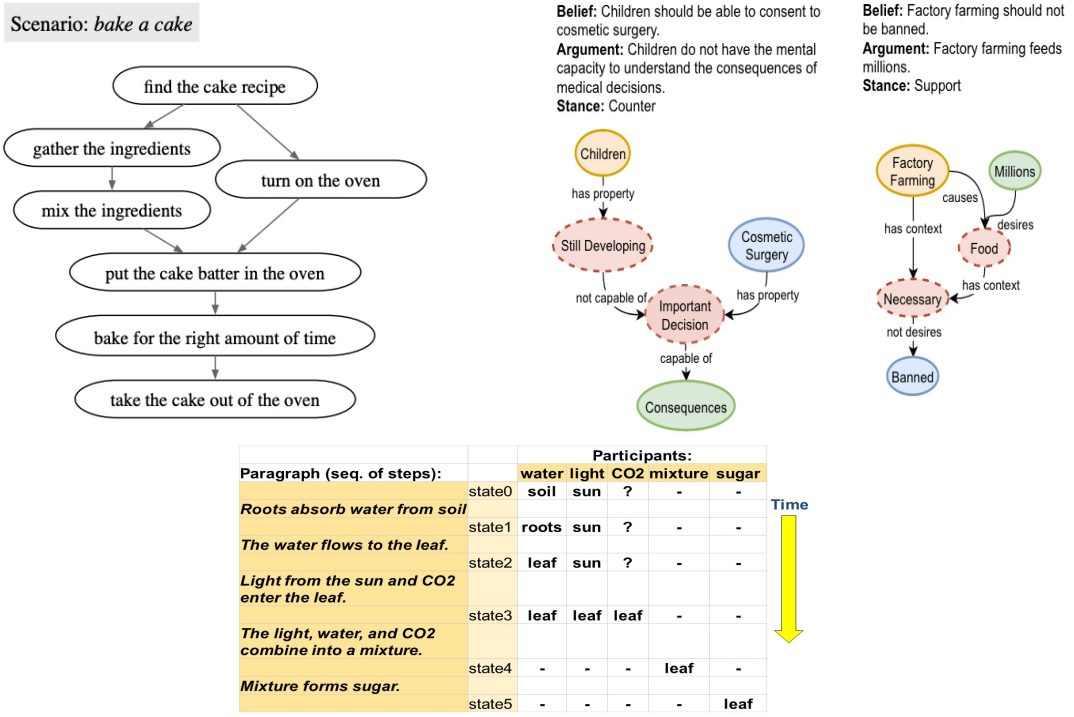


Figure 5: Example graphs for each of the tasks used for CoCoGen: PROSCRIPT (top-left), EXPLAGRAPHS (top-right), and PROPARA (bottom).

Model	Format	StCA (↑)	SeCA (↑)	G-BS (↑)	GED (↓)	EA (↑)
CODEX-002	Literal	**45.2**	**23.74**	**34.68**	**68.76**	**23.58**
CODEX-002	Tree	39.24	15.95	30.49	73.85	18.24
CODEX-002	Relation	42.82	23.68	33.38	70.23	21.16

Table 13: Performance of CODEX on the three different formats present in Figure 7 for EXPLAGRAPHS.

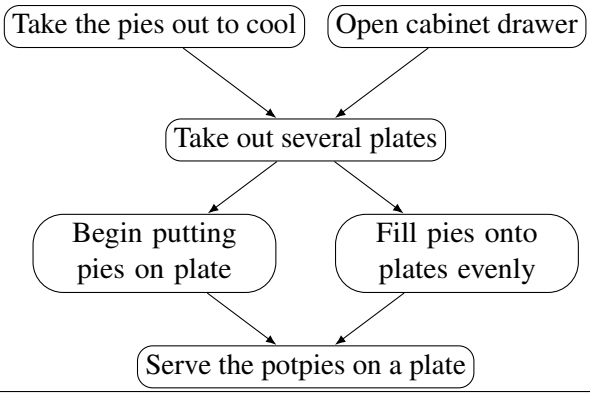
Model	Format	F_1
CODEX-001	Literal	15.9
CODEX-001	Tree	29.7
CODEX-002	Literal (Figure 9)	52.0
CODEX-002	Tree (Figure 10)	56.5

Table 14: Performance of CODEX-001 and CODEX-002 on the the different formats present in Figure 10 and 9 for PROSCRIPT edge prediction. We find that the literal format that combines structure with literally Figure output performs the best for CODEX-002.

Model size vs. sensitivity to the prompt In Table 14 shows the performance of CODEX-001 (smaller) and CODEX-002 (larger, also see Appendix A) on identical prompts. Our experiments show that as model size increases, the sensitivity of the model on the prompt reduces. This indicates that for very large models, prompt design might get progressively easier.

I Variation in prompts

We run each experiment with 3 different random seeds, where the random seeds decides the order of examples in the prompt. We find minimal variance between runs using different fixed prompts between 3 runs. Further, as shown in the Tables 18,19, 20, and 21, all improvements of CoCoGen over DAVINCI are statistically significant (p-value < 0.001).




```
class Tree:

  goal = "serve the potpies on a plate"

  def __init__(self):
    # nodes
    begin = Node()
    take_pies_out_to_cool = Node()
    take_out_several_plates = Node()
    open_cabinet_drawer = Node()
    fill_pies_onto_plates_evenly = Node()
    begin_putting_pies_on_plates = Node()
    serve_potpies_on_plate = Node()

    # edges
    begin.children = [take_pies_out_to_cool, open_cabinet_drawer]
    take_pies_out_to_cool.children = [take_out_several_plates]
    open_cabinet_drawer.children = [take_out_several_plates]
    take_out_several_plates.children = [begin_putting_pies_on_plates,
        fill_pies_onto_plates_evenly]
    begin_putting_pies_on_plates.children = [serve_potpies_on_plate]
    fill_pies_onto_plates_evenly.children = [serve_potpies_on_plate]
    serve_potpies_on_plate.children = [end]
```

Figure 6: A PROSCRIPT plan (top) and the corresponding Python code (bottom).

	BLEU	ROUGE-L	BLEURT
DAVINCI	23.1±2.7	36.5±2.7	-0.27±0.06
COCOGEN	25.3±0.1	38.3±0.1	-0.25±0.01

Table 18: PROSCRIPT script generation: mean and standard deviation across three different random seeds.

	F1
DAVINCI	48.9 ± 2.8
COCOGEN	**56.2±2.1**

Table 19: PROSCRIPT edge prediction: mean and standard deviation across three different random seeds.

	F1
DAVINCI	56.9 ± 2.4
COCOGEN	**62.8** ± 2.4

Table 21: PROPARA: mean and standard deviation across three different random seeds.

Model	Format	ISO	GED	Avg(d)	Avg($\|V\|$)	Avg($\|E\|$)	BLEU	ROUGE-L	BLEURT
$\mathcal{G}$	-	1.0	0.0	1.84	7.41	6.8	-	-	-
CODEX-001	Literal (Figure 9)	**0.55**	**1.8**	1.74	7.45	6.5	22.9	36.2	-0.36
CODEX-001	Tree (Figure 10)	0.35	3	**1.79**	7.45	6.65	17.8	30.7	-0.45
CODEX-001	NetworkX (Figure 8)	0.51	1.81	1.69	7.49	6.32	23.7	35.9	-0.37
CODEX-002	Literal (Figure 9)	0.53	2.1	**1.79**	**7.44**	**6.7**	**25.24**	**38.28**	**-0.26**
CODEX-002	Tree (Figure 10)	0.35	2.46	1.61	7.46	5.74	18.96	32.92	-0.38
CODEX-002	NetworkX (Figure 8)	0.5	2.46	**1.79**	**7.38**	6.61	23.88	36.89	-0.33

Table 15: CODEX results on PROSCRIPT generation for various Python source formats.

```
class Relation:

    def __init__(self):
        belief = "Cannabis should be legal."
        argument = "It's not a bad thing to make marijuana more available."
        stance = "support"

        # create a DAG to support belief using argument
        begin = ["cannabis"]
        add_edge("cannabis", "synonym of", "marijuana")
        add_edge("legal", "causes", "more available")
        add_edge("marijuana", "capable of", "good thing")
        add_edge("good thing", "desires", "legal")

class Tree:
    def __init__(self):
        self.belief = "Cannabis should be legal."
        self.argument = "It's not a bad thing to make marijuana more available."
        self.stance = "support"

        # tree for support in support of belief
        root_nodes = cannabis
        cannabis = Node()
        cannabis.add_edge("synonym of", "marijuana")
        legal = Node()
        legal.add_edge("causes", "more available")
        marijuana = Node()
        marijuana.add_edge("capable of", "good thing")
        good_thing = Node()
        good_thing.add_edge("desires", "legal")


 class Literal:
    def __init__(self):
        self.belief = "Cannabis should be legal."
        self.argument = "It's not a bad thing to make marijuana more available."
        self.stance = "support"
        self.graph = """\
(cannabis; synonym of; marijuana)(legal; causes; more available)
(marijuana; capable of; good thing)
(good thing; desires; legal)"""
```

Figure 7: Templates tried for explagraph.

	ISO	GED	Avg(d)	Avg($\|V\|$)	Avg($\|E\|$)	BLEU	ROUGE-L	BLEURT	
$\mathcal{G}$	1.0	0.0	0.0	1.84	7.41	6.8	-	-	-
COCOGEN + 001 (15)	0.55	1.8	1.74	7.45	6.5	22.9	36.2	-0.36	
COCOGEN + 002 (15)	0.53	2.1	1.79	**7.44**	**6.7**	**25.24**	**38.28**	**-0.26**	

Table 16: CODEX-001 vs 002 on PROSCRIPT script generation

```
class Plan:

    goal = "create a video game"
    num_steps = 7

    def __init__(self):
        graph = nx.DiGraph()
        # add nodes
        step0 = "decided to create a video game"
        step1 = "Learn the basics of programming"
        step2 = "Learn to use a language that is used in games"
        step3 = "Learn to use an existing game engine"
        step4 = "Program the game"
        step5 = "Test the game"
        step6 = "create a video game"
        graph.add_nodes_from([step0, step1, step2, step3, step4, step5, step6])

        # add edges
        graph.add_edge(step0, step1)
        graph.add_edge(step1, step2)
        graph.add_edge(step1, step3)
        graph.add_edge(step2, step4)
        graph.add_edge(step3, step4)
        graph.add_edge(step4, step5)
        graph.add_edge(step5, step6)
```

Figure 8: Proscript as a Networkx class.

```
class CreateAVideoGame:

   title = "create a video game"
   steps = 7

   def step0(self):
       return "decided to create a video game"
   def step1(self):
       return "Learn the basics of programming"
   def step2(self):
       return "Learn to use a language that is used in games"
   def step3(self):
       return "Learn to use an existing game engine"
   def step4(self):
       return "Program the game"
   def step5(self):
       return "Test the game"
   def step6(self):
       return "create a video game"
   def get_relations(self):
       return [
           "step0 -> step1",
           "step1 -> step2",
           "step1 -> step3",
           "step2 -> step4",
           "step3 -> step4",
           "step4 -> step5",
           "step5 -> step6",
       ]
```

Figure 9: Representing PROSCRIPT graph literally.

	StCA (↑)	SeCA (↑)	G-BS (↑)	GED (↓)	EA (↑)
DAVINCI	25.4 ± 2.7	13.7 ± 2.8	20 ± 2.3	82.5 ± 1.9	13.6 ± 1.8
COCOGEN	**44.0** ± 1.2	**25.1** ± 2.5	**34.1** ± 0.7	**69.5** ± 0.7	**22.0** ± 1.3

Table 20: EXPLAGRAPHS: mean and standard deviation across three different random seeds.

```
class Tree:

    goal = "serve the potpies on a plate"

    def __init__(self):
        # nodes
        begin = Node()
        take_pies_out_to_cool = Node()
        take_out_several_plates = Node()
        open_cabinet_drawer = Node()
        fill_pies_onto_plates_evenly = Node()
        begin_putting_pies_on_plates = Node()
        serve_potpies_on_plate = Node()

        # edges
        begin.children = [take_pies_out_to_cool, open_cabinet_drawer]
        take_pies_out_to_cool.children = [take_out_several_plates]
        open_cabinet_drawer.children = [take_out_several_plates]
        take_out_several_plates.children = [begin_putting_pies_on_plates,
                                            fill_pies_onto_plates_evenly]
        begin_putting_pies_on_plates.children = [serve_potpies_on_plate]
        fill_pies_onto_plates_evenly.children = [serve_potpies_on_plate]
        serve_potpies_on_plate.children = [end]
```

Figure 10: Proscript with a tree-encoding.