Report

Work Done

All tasks have been attempted. Task 1: page 1, Task 2: page 11, Task 3: page 6, Task 4: page 12.
Supplementary material has been attached. refer branch Report on github repository

I apologise for the poor formatting of this report. I was very sick while making the report. (ref mail to Debangan, CCed to Professor PK)

Disclosures

1. I thank Prof. Kshitij Gajjar (IIIT-H), who helped me understand the intuition behind the proof of **Theorem 2**.
2. I would like to thank my friend and fellow student Manit Roy, who graciously provided me with an OpenAI API key for experimentation.
3. Used GitHub Copilot to assist in implementing functions (but not in defining the architecture).

Assumptions

- For the creation of the dataset, publicly available problems (even with simple heuristics) must be used only to a very limited extent.

Defining the Problem

Given a string s and a set of transitions $w \to w'$ with exactly one transition of the form $w \to$ empty, when a transition is applied, the first occurrence of w as a substring of s is replaced with w'. The goal is to determine the order in which these transitions should be applied (with repetition allowed) so that the final string is empty.

Task 1: Data Generation

Concept of Non-Isomorphic Strings

We do not want to sample structurally similar strings. For example, consider "abca" and "xyzx". Although they are distinct, they are very similar structurally to each other. The alphabet used is not relevant to the problem. Thus, we sample only non-isomorphic strings.

Isomorphic Strings: Two strings are said to be isomorphic if the characters in one string can be mapped to characters in another string while preserving the order of the characters.

Methodology to create database of Non-Isomorphic strings

- S(n,k) represents set of non-isomorphic strings of length n and k distinct characters
- Create $S(1,1) = \{"a"\}$.
- **Theorem 1:**Recursion: $|S(n,k)| = |S(n-1,k)| \times k + |S(n-1,k-1)|$ (where k is the number of unique characters in the string) holds
- **Proof 1:**i.e To obtain a string of length n, and k distinct characters, Case 1: take a string, s of length n-1, and k-1 distinct characters and add a new distinct letter. All strings obtained by this, will be in the same isomorphic class. Hence, we can take only one string from this case Case 2: take a string, s of length n-1, and k distinct characters. Take any of the k distinct characters and concatenate to end of string. All such string would be non-isomorphic with respect to each other. (proof below)

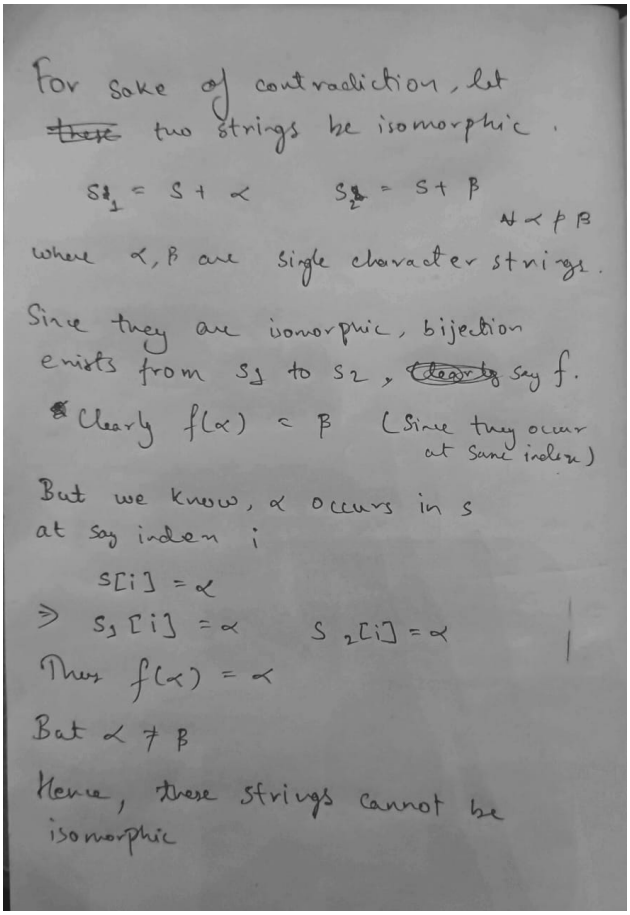


Figure 1: Part of proof 1

- We use the concept used in proof 1 to generate our dataset
- Interestingly, the same recursion holds for number of partitions of a set Theorem 2. By the virtue of this, we know the storage complexity of this dataset of NonIsomorphic Strings, which is $\mathrm{O}\left(\frac{n}{\ln(n)}\right)^n$

Methodology of Data Generation

Since puzzle solutions are of the form initial string $\rightarrow \cdots \rightarrow$ penultimate string $\rightarrow$ empty string, we break it into two parts:

1. Initial string $\rightarrow \cdots \rightarrow$ penultimate string.
2. (Trivial) Penultimate string $\rightarrow$ empty string.

Part 1

- Select n (length) from a Gaussian distribution.
- Select t (number of unique transitions) from a Gaussian distribution.
- Select d (number of transitions in the puzzle, where the total number of transitions is $d + 1$) from a Gaussian distribution.
- Sample p and q as well as the length of the string in transition (exponential with scale = n).
- For source (src) and target (tgt):
  - Allow up to 2 new characters from the original string. (e.g., if the root string is "abc", transitions may contain only "a, b, c, d, e").
- Apply all available transitions till depth d.
- Using the graph generated, sample individual paths, which will act as distinct puzzles.
- see figure 1 for example

Applying Final Transition

Now, we add the final transition: penultimate string $\rightarrow$ empty string. Hence, we have created a graph of the form initial string $\rightarrow \cdots \rightarrow$ penultimate string $\rightarrow$ empty string. The conversion of this graph to a puzzle with the provided schema is rather trivial (refer to `src/puzzle_generator.py`).

This has 3 problems:

1. The complexity of the graph scales up very quickly, and it is infeasible to use such graphs. (See figure 3)
2. The number of paths we receive in such a graph is uncertain. Hence, it is logistically difficult to use such a method for sampling paths.
3. All these problems use the same set of transitions. This is not representative of the actual problem.

Hence, I came up with the improvement that, rather than applying all possible transitions, we randomly choose an applicable transition and apply it. We repeat this for d steps. (ref src/puzzle_generator.py) (see figure 4)

Key point: To get more patterns in sequence, keep d/t ratio high

0.1 Bias Mitigation

1. Since in all the puzzles, the last transition would always be applied at the end and not anywhere else, which is almost always the case with puzzles on sed-puzzle.com. However, there are certain puzzles where transitions of the form **"non-empty-string" $\rightarrow$ "empty string" are also applied multiple times**, not necessarily only at the end of the puzzle. This problem was not addressed due to time constraints.
2. Since our implementation for sampling root strings is dependent on the database of non-isomorphic strings, a string of length n only features up to the first n letters of the alphabet. This is not completely representative of the problem, and thus, we must **shuffle the alphabet** while creating a puzzle. However, due to the lack of time, we ignore this issue for now. Since the structure of the puzzle is essentially the same, it should not be a significant problem.

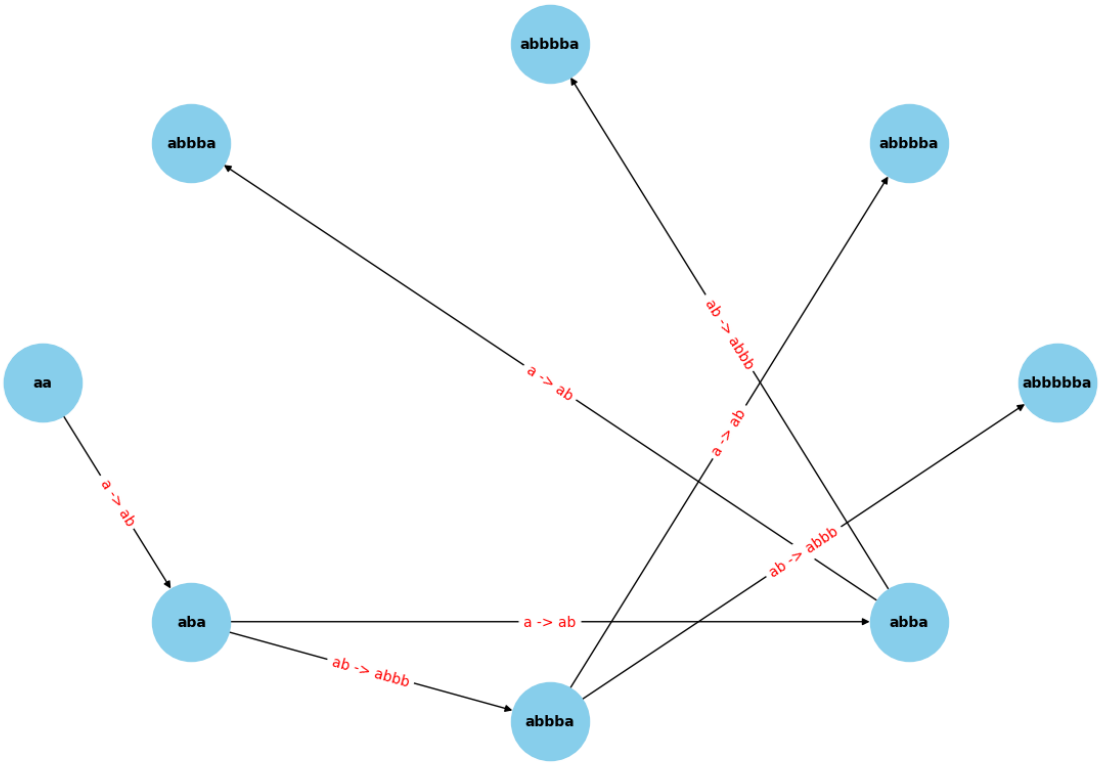


Figure 2: sample graph (n=2, t= 3, d= 3)

Dataset Creation

The dataset is made up using a combination of various subsets to ensure a good gradient in accuracy (or other evaluation metrics).

tabularx

Quantifying Patterns

Throughout data generation and evaluation, the key concept we revolve around is patterns. They make puzzles on sed-puzzle.com distinct from random samples. To evaluate how "patterny" or "predictable" a sequence is, we use:

Markov Entropy

Markov Entropy: The Markov entropy of a sequence is a measure of the uncertainty or randomness in the sequence, given the previous state. It is defined as:

$$H(X) = -\sum_{i} P(x_i) \log_2 P(x_i)$$

where $P(x_i)$ is the probability of the i-th state in the sequence.

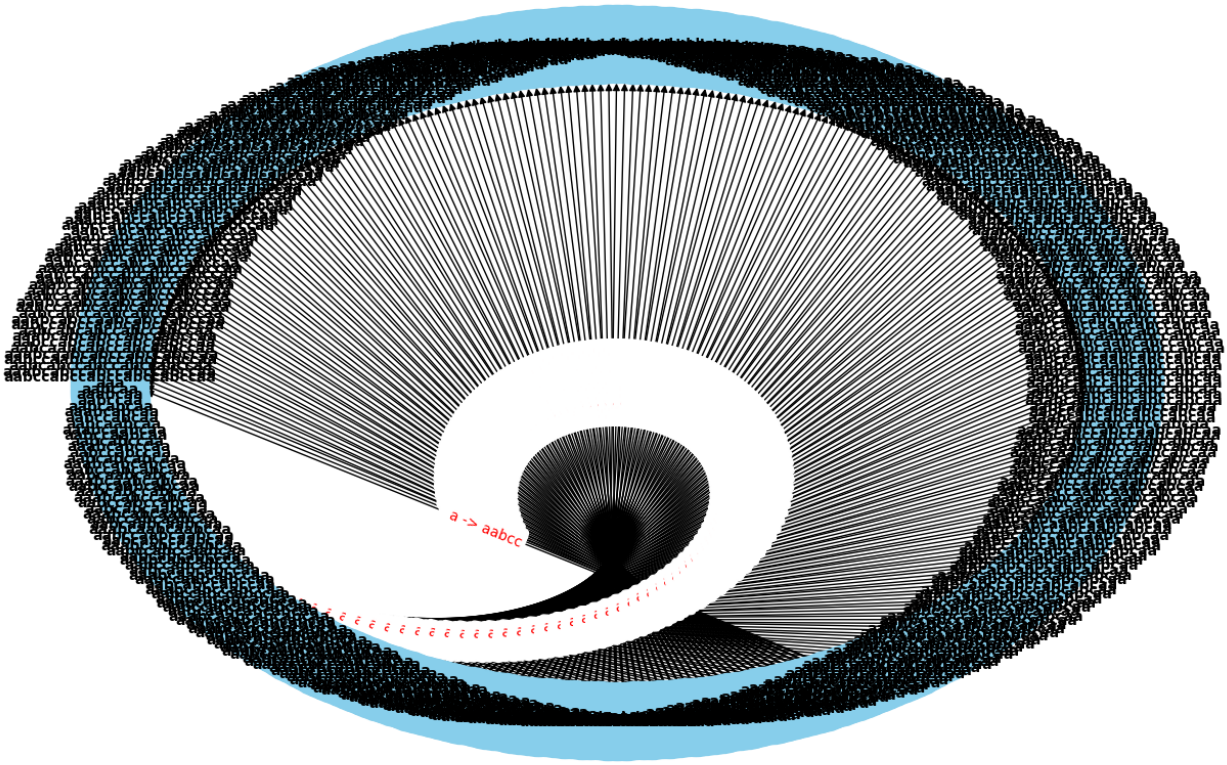


Figure 3: graph n =3, t = 3, d = 5

Pattern Score

Pattern Score: The pattern score is defined as:

$$\text{Pattern Score} = 1 - \frac{\text{Markov Entropy of Solution Sequence}}{\log_2(\text{Length of Solution Sequence})}$$

This changes the range to $(0, 1]$ and makes it easier to analyze.

0.2 Note

Markov entropy is a known metric. Pattern score, however, was defined by me for the purpose of this project. I could not find instances where such a definition of pattern score was used.

Task 3: Evaluation Metric

We want to measure the accuracy. However, problems with less pattern score are not as important. Thus, we define pattern-weighted accuracy.

Pattern-Weighted Accuracy:

$$\text{Pattern-Weighted Accuracy} = \frac{\sum(\text{Pattern Score} \times \text{Accuracy})}{\sum(\text{Pattern Score})}$$

This measures performance with respect to the pattern in the solution sequence, which is my major consideration for being representative of sed-puzzle.com.

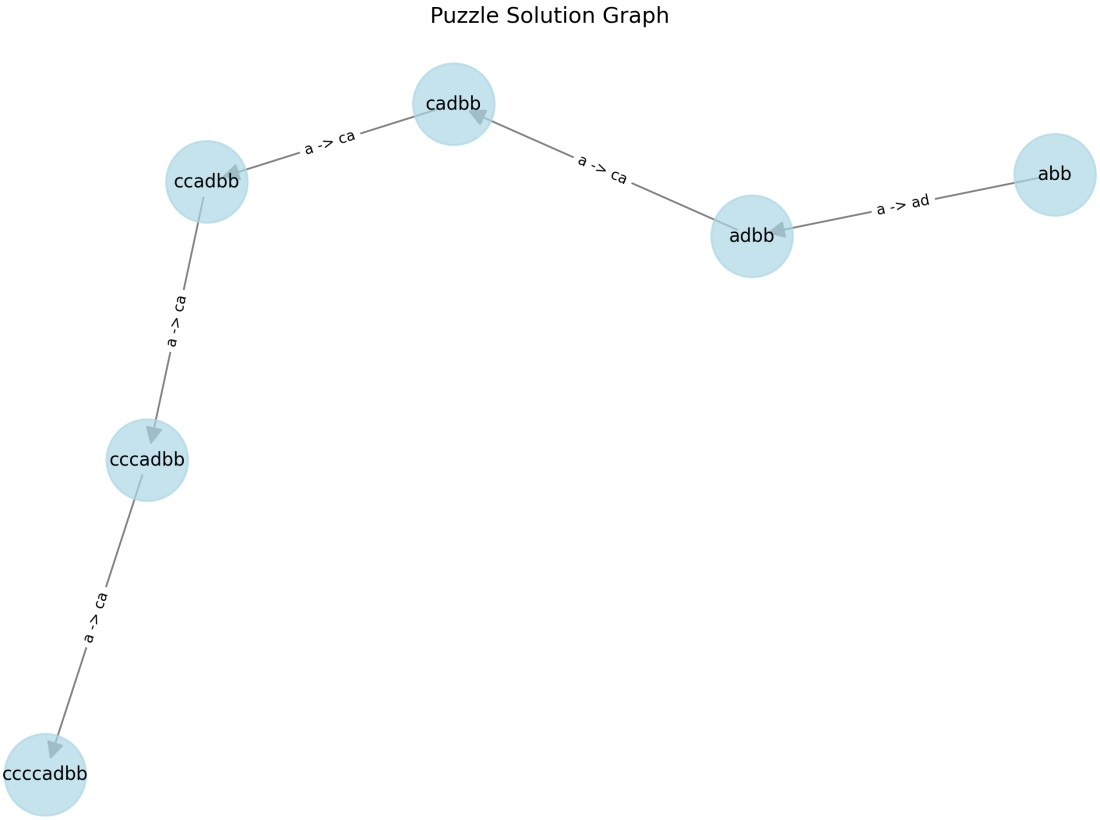


Figure 4: Single path sample

Task 2: LLM Evaluation

Exploratory Analysis

Evaluation on very simple datasets (OneShot prompting) showed that certain problems performed very well as training problems, while some performed poorly. Interestingly, for problems that were good for training, the accuracy was very low when the model was tested upon these. Similarly, problems that were bad training examples were easy to solve.

Methodology for Further Testing

I implement a pipeline to systematically test LLMs on datasets(src/pipeline.py).

The data creation part was later removed for testing on pre-made datasets(src/SED$_{p}ipeline.py$)($SeeFigure11 : Pipelineimplementationbelow$)

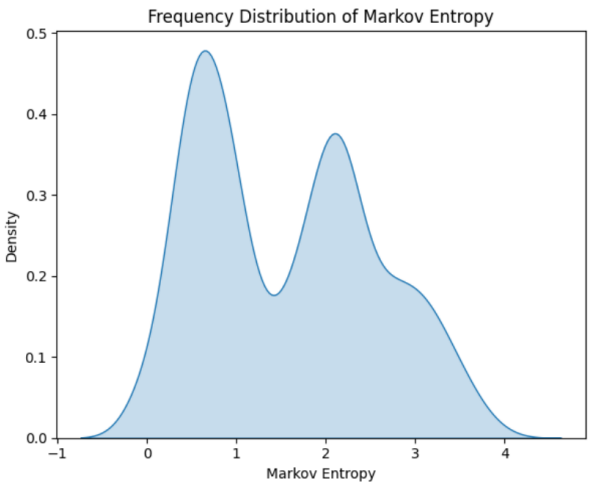
Note: Additionally, experiments were also divided into runs. All parameters for multiple runs of a single parameter were identical, but runs were different by the virtue of random sampling. Self-consistency of experiments was evaluated by comparing runs.

$$\text{Self-Consistency Score} = 1 - \frac{\sigma}{\mu}$$

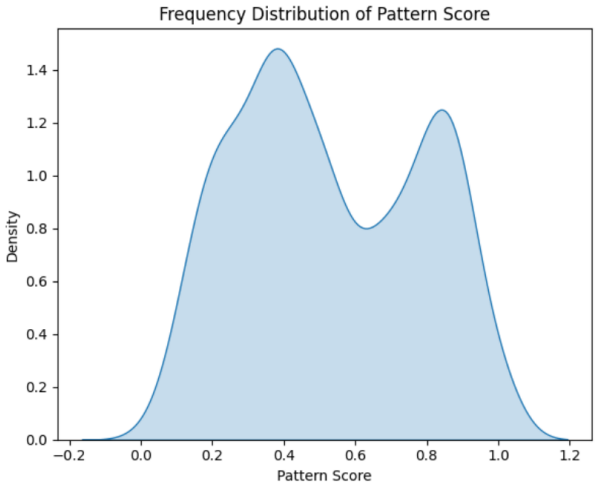
where σ is the standard deviation of accuracy of multiple runs in an experiment, and μ is the mean of accuracy of multiple runs in an experiment.

Dataset Name	# Samples	Length of Initial String (n)	Number of Possible Transitions (t)	Length of Solution Sequence (d)
SED_10	10	1-10	1-7	t to 4t
MIX_2_3_5_SED_20	20	1-3	1-3	1-5
MIX_3_4_5_SED_10	10	1-3	1-4	1-5
MIX_3_2_3_SED_20	20	1-3	1-2	1-3
MIX_3_3_4_SED_20	20	1-3	1-3	1-4
MIX_2_2_3_SED_20	20	1-2	1-2	1-3

Figure 5: Subsets and their parameters



(a) markov Entropy Distribution



(b) Pattern score distribution

Figure 6: refer SED_1000/plots

However, due to time constraints, I could not afford to do multiple runs for most experiments.

This pipeline is robust and can be used for various kinds of prompting:

- Zero Shot: set `train_count` to 0.
- Few Shot: set `train_count` to a finite number.
- Chain of Thought: set `give_explanation` flag to 1.

Results (See table 1: results on standard prompting below)

2 puzzles were sampled from each of the subsets in view of resource constraints for testing

Since datasets gave 0 accuracy on Few shot chain of thought, I did not test most of them on other methods, to save api toens and time

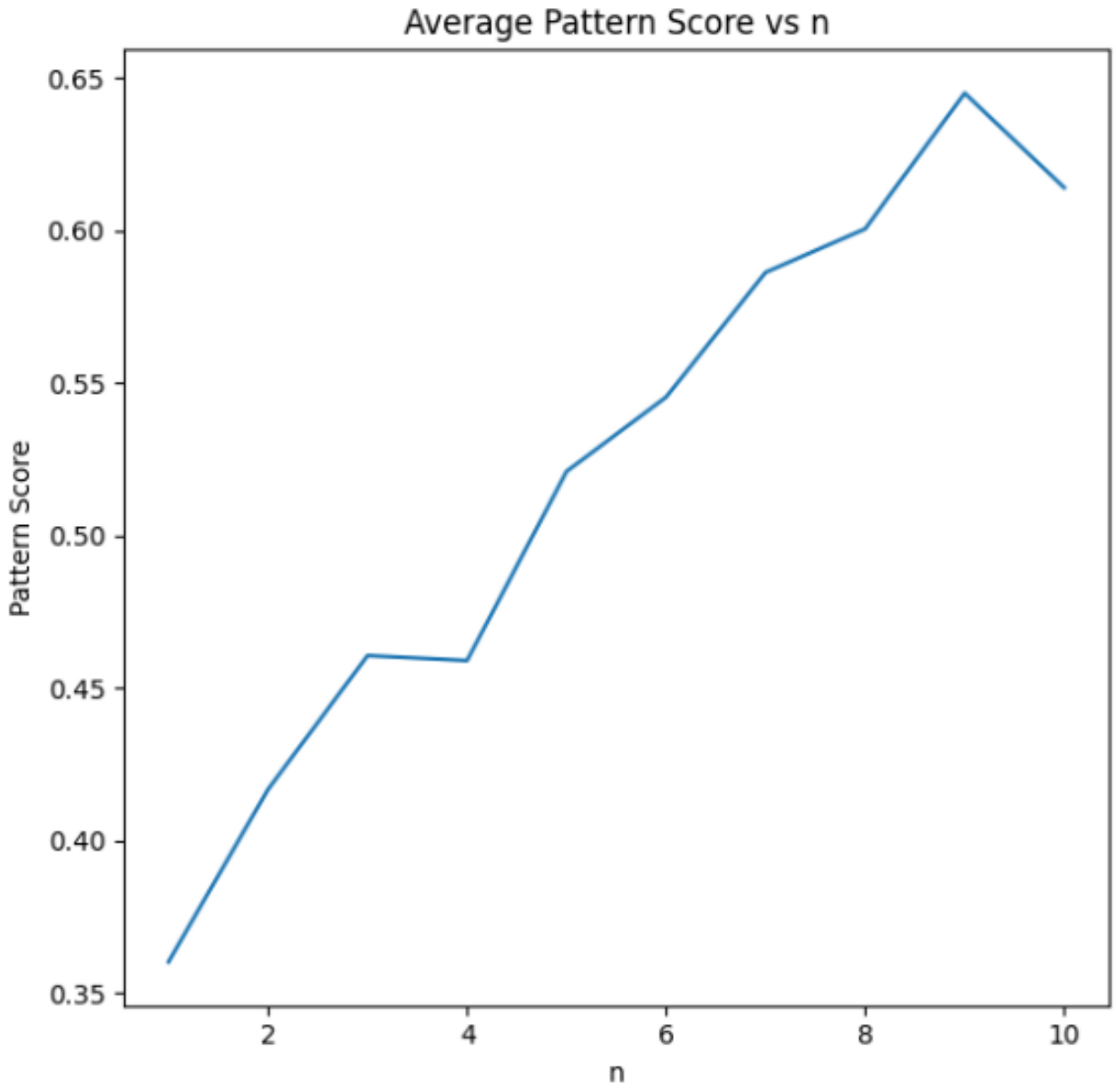

Average Pattern Score vs n
Pattern Score
0.65
0.60
0.55
0.50
0.45
0.40
0.35
2
4
6
8
10
n

Figure 7

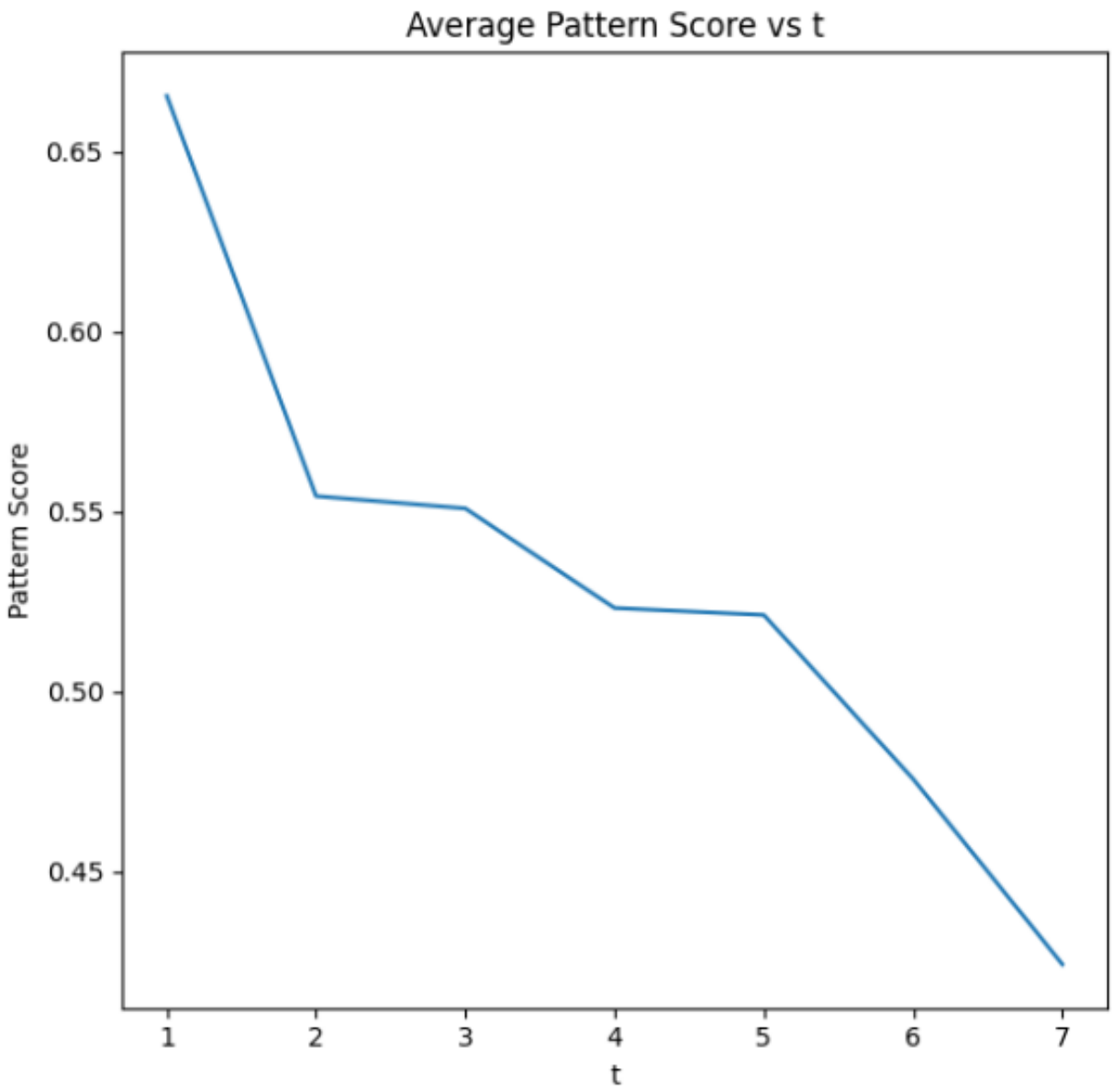

Average Pattern Score vs t
Pattern Score
0.65
0.60
0.55
0.50
0.45
1
2
3
4
5
6
7
t

Figure 8

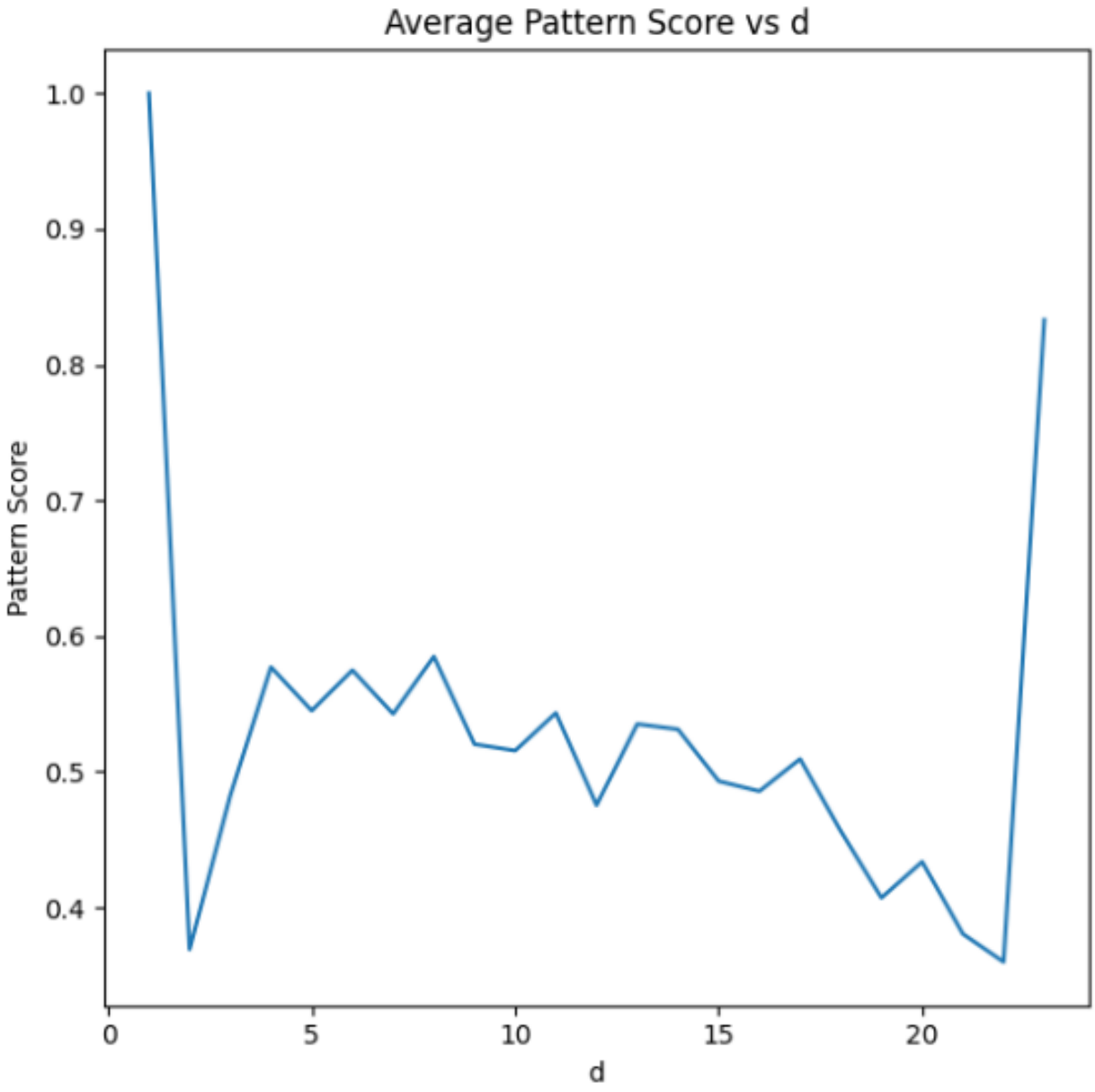

Average Pattern Score vs d
Pattern Score
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
5
10
15
20
d

Figure 9

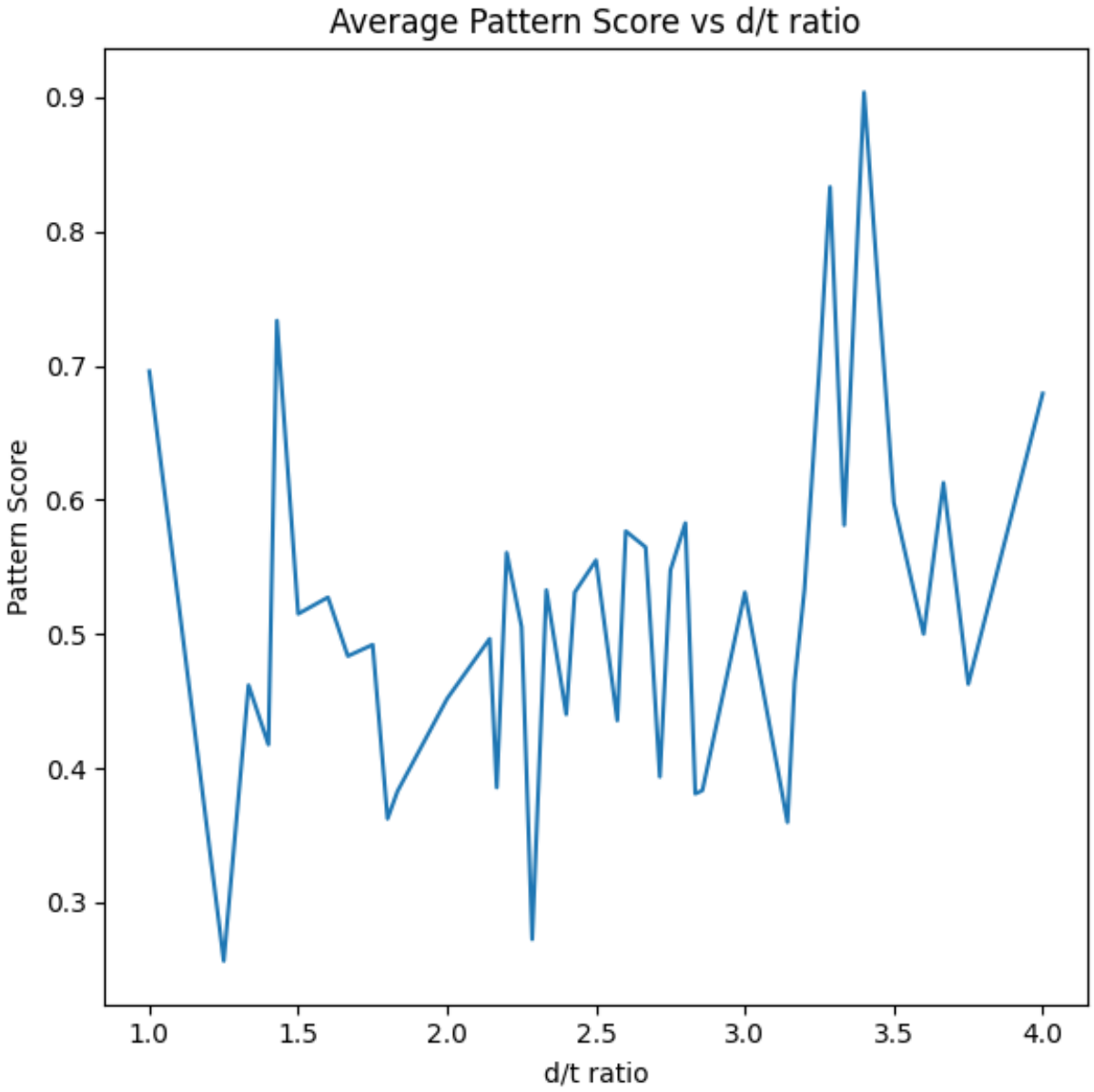

Average Pattern Score vs d/t ratio
Pattern Score
d/t ratio
0.9
0.8
0.7
0.6
0.5
0.4
0.3
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Figure 10

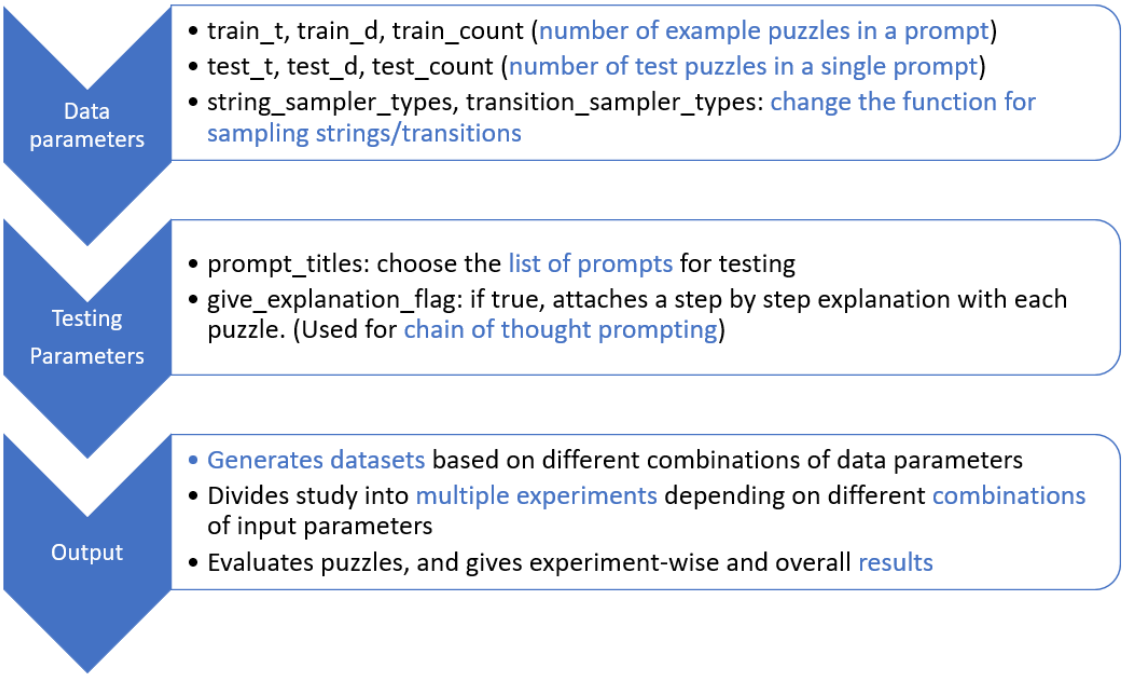


Figure 11: Pipeline Implementation.

Dataset	Zero Shot	Zero Shot Chain of Thought	Few Shot	Few Shot Chain of Thought
MIX_2_3_5_SED_20	0%	0%	0%	0%
MIX_3_2_3_SED_20	-	-	-	0%
MIX_3_3_4_SED_20	-	-	-	0%
MIX_3_4_5_SED_10	-	-	-	0%
SED_10	-	-	-	0%

Table 1: Results Table

- **What if the model can't read training problems?**
  - When it is given the same problems for training and testing, the accuracy is 100%.
- **What if the model can't read testing problems?**
  - The solutions it gives have problem IDs of the testing problems.
  - It is visible from the logs that it can see the correct root string and transitions.
- **What if the model can't see the prompt?**
  - When asked to give a specific phrase in the explanation (e.g., "Bla Bla Bla") using the prompt, it includes the phrase in the explanation.

Interactive Prompting

If the model gives an invalid solution for a problem, its solution is evaluated, and a string is generated that explains the outcomes of this incorrect solution. Then, the model is prompted again with the conversation history and this string. (ref src/symbolic$_p$*ipeline.py*)

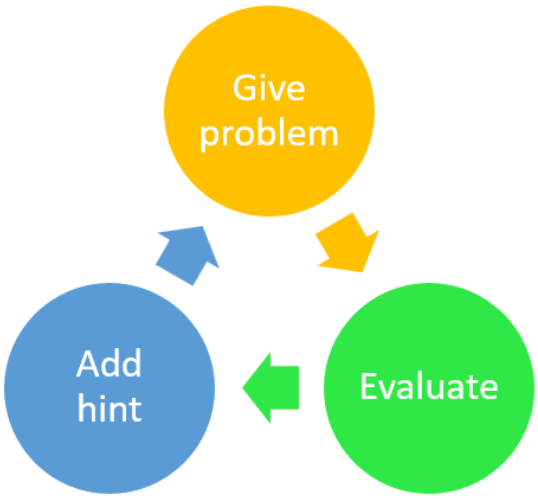


Figure 12: Interactive prompting

Results

12 problems were sampled from each subset (except MIX_3_4_5_SED_10 as it had only 10 problems).

Dataset	Accuracy	Pattern-Weighted Accuracy
SED_10	0%	0%
MIX_2_3_5	40%	28%
MIX_3_4_5	30%	30%
MIX_3_2_3	40%	27%
MIX_3_3_4	70%	41%
MIX_2_2_3	40%	36%

Table 2: Results with Pattern-Weighted Accuracy

Comparison with Humans

While I could not do a study over humans due to the lack of time, here is what I got from a few minutes of testing over my friends:

Humans perform poorly on problems with a high number of letters and complex-looking transitions but beat LLMs on problems with a high d/t ratio and high pattern score.

Since this was informal, I do not have proper written records of the same. These are just general insights I could gain by seeing them trying to answer.

Potential

- One may use the concept of pattern_score to sample better root strings. The present dataset does not account for the fact that initial strings in sed-puzzle.com are also "patterny".
- One may scrape some puzzles from sed-puzzle.com and ask an LLM to generate more data. This is a practice that has recently started to show up in various studies.

- I personally feel I could have worked better on the prompts that I gave. I feel the reason is I could not give enough time to understanding the puzzles and, more importantly, a general approach to optimally answer them.