Prompt Structuring in Gemini AI Using Set Theory, Logic, and Relations

Reynard Nathanael - 13524103
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: reynardnathanael11@gmail.com , 13524103@std.stei.itb.ac.id

***Abstract*—Effective prompt engineering is crucial for leveraging Large Language Models (LLMs) like Gemini AI, yet conventional methods often lack precision for complex generative tasks. This paper introduces a systematic prompt construction methodology founded on set theory, logic, and relations. Using a framework that decomposes intricate task requirements into distinct, universal categories, formally defining their elements and interrelationships. Through an empirical evaluation with Gemini AI, comparing intuitive prompts against those designed using this structured approach, to demonstrate its efficacy. The results consistently show that structured prompts yield significantly more precise, consistent, contextually relevant, and role-aligned outputs. By refining control over the AI's output generation, this method enhances response consistency and contributes to the development of prompt engineering as a more methodical and evidence-driven practice.**

Keywords— Prompt engineering; Set theory; Gemini AI; Prompt design.

I. INTRODUCTION

Generative Artificial Intelligence (Generative AI) such as *Gemini AI*, offers remarkable capabilities in understanding, processing, and automatically generating relevant output. This form of AI demonstrates vast potential, ranging from content creation and information retrieval to task automation. However, to fully harness its capabilities, prompt engineering has become essential. It is the technique of crafting effective instructions to elicit relevant and high-quality responses. The quality and relevance of AI-generated outputs heavily depend on the clarity, precision, and structure of the prompt provided.

Unfortunately, as the complexity of tasks assigned to AI increases, conventional prompting approaches often fail to identify the internal and external relationships between instructional elements, leading to ambiguous, inefficient prompts that result in suboptimal responses. This issue highlights the need for more advanced frameworks for prompt construction, aimed at formalizing the relationships among prompt components and minimizing ambiguities that hinder effective AI responses.

Set theory, logic, and relations in discrete mathematics offer essential tools for modeling the structural complexity of prompts, making it a key asset in generative AI analysis. Set theory enables us to define a prompt as a collection of discrete elements; logic provides rules to bind instruc tions and conditions; while relations allow the user to model the interconnections between different parts of a prompt. In the context of generative AI, a prompt may include various instructions, constraints, and contextual elements. The application of these mathematical theories opens up systematic methods for decomposing complexity, identifying redundancies or conflicts, and offering actionable insights for prompt optimization.

The research in this paper aims to demonstrate the application of set theory, logic, and relations in optimizing prompt structure and improving interaction analysis with generative AI, particularly *Gemini AI*. This study seeks to simplify complex networks of instructions within a prompt, uncover logical, and hierarchical relationships between its components, and identify opportunities for crafting more precise and efficient prompts through the construction of mathematical models. This model-based approach offers actionable insights for prompt engineering practitioners, supporting better decision-making and more effective interaction management with generative AI systems.

II. THEORETICAL FRAMEWORK

2.1 Set

A set is an unordered collection of distinct objects, known as elements or members. The order of elements within a set is not important, and repeated elements are counted only once, unless it is specifically referred to as a multiset.

In this context, a prompt can be viewed as a set of keywords, instructions, or constraints. Set notation typically employs capital letters, such as A={e1,e2,...,en }, where ei are the elements of the set. An object belonging to a set is referred to as an element (∈).

The following are the fundamental set operations relevant to prompt structuring:

- Subset (⊆): A set A is a subset of set B if every element in A is also an element in B.
- Union (∪): This operation yields a set containing all elements that are in at least one of the given sets.

- Intersection (∩): This operation yields a set containing all elements common to all given sets.
- Difference (− or \): This operation yields a set containing the elements of the first set that are not in the second set.
- Complement (A^c or A'): This is the set of all elements in the universal set U that are not in A.

2.2 Logic

Logic provides a formal system for reasoning and deriving conclusions from premises. In prompt structuring, it allows for the precise definition of instructions, conditions, and constraints, ensuring that the AI's response adheres to specified rules.

- Proposition: A declarative statement that is either true or false, but not both. Each instruction or condition within a prompt can often be framed as a proposition.
    - *Examples:* "The output must be in English," "Summarize the text."

- Logical Operators:
    - Conjunction (AND, ∧): True only if both propositions are true. Represents multiple simultaneous requirements.
    - Disjunction (OR, ∨): True if at least one proposition is true. Represents alternative options or requirements.
    - Negation (NOT, ¬): Reverses the truth value of a proposition. Represents exclusions or negative constraints.
    - Implication (IF...THEN, →): If the first proposition is true, then the second must also be true. Represents conditional instructions.
    - Biconditional (IF AND ONLY IF, ↔): True if both propositions have the same truth value. Represents equivalence between instructions or conditions.

- Quantifiers:
    - Universal Quantifier (∀): "For all" or "for every." Used to state that a property applies to every element in a domain.
    - Existential Quantifier (∃): "There exists" or "for some." Used to state that at least one element in a domain has a certain property.

2.3 Relations

Relations describe connections or associations between elements of sets. During prompt formulation, they help illustrate the connections between elements such as instructions, context, limitations, and the intended format of the output.

- Binary Relation: A set of ordered pairs (a,b) where a∈A and b∈B. It indicates that element a is related to element b.
- Domain: The set of all first elements in the ordered pairs of a relation.
- Codomain: The set of all possible second elements in the ordered pairs.
- Range: The set of all actual second elements in the ordered pairs of a relation.

- Properties of Relations:
    - Reflexive: If (a,a)∈R for every element a in the set.
    - Symmetric: If (a,b)∈R implies (b,a)∈R.
    - Antisymmetric: If (a,b)∈R and (b,a)∈R, then a=b.
    - Transitive: If (a,b)∈R and (b,c)∈R, then (a,c)∈R.

III. IMPLEMENTATION

3.1 Experimental Design

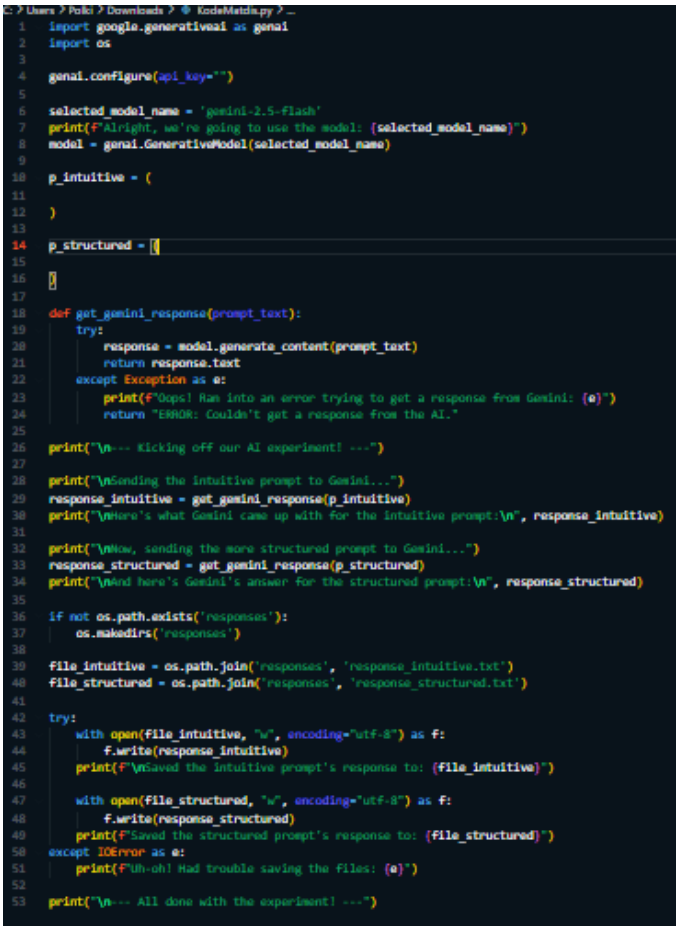
A comparative approach will be employed for a specific task scenario requiring the AI to explain a technical concept. Two distinct prompts will be formulated and submitted to Gemini AI:

1. Intuitive Prompt ($P_{intuitive}$) :
A prompt crafted using conventional, unstructured language, mimicking how a user might naturally phrase the request without explicit consideration of discrete mathematical principles. It serves as a baseline for comparison.

2. Mathematically Structured Prompt ($P_{structured}$) :
A prompt crafted by explicitly applying the principles of set theory, logic, and relations.

Task Scenario:
The chosen task scenario is to explain the concept of **Linked List in C programming language** to **informatics students in their second semester**. The explanation must include **two simple analogies**, be presented **in English**, and be limited to a **maximum of 200 words.**

These prompts will then be submitted to Gemini AI using model 2.5-flash for further analysis. While direct submission via the Gemini AI Web API is an option, the implementation process will use a Python program integrated with a Google API Key. This approach is chosen to ensure consistency, reproducibility, and a systematic execution of the prompt.



```python
import google.generativeai as genai
import os

genai.configure(api_key="")

selected_model_name = 'gemini-2.5-flash'
print(f"Alright, we're going to use the model: {selected_model_name}")
model = genai.GenerativeModel(selected_model_name)

p_intuitive = (

)

p_structured = (

)

def get_gemini_response(prompt_text):
    try:
        response = model.generate_content(prompt_text)
        return response.text
    except Exception as e:
        print(f"Oops! Ran into an error trying to get a response from Gemini: {e}")
        return "ERROR: Couldn't get a response from the AI."

print("\n--- Kicking off our AI experiment! ---")

print("\nSending the intuitive prompt to Gemini...")
response_intuitive = get_gemini_response(p_intuitive)
print("\nHere's what Gemini came up with for the intuitive prompt:\n", response_intuitive)

print("\nNow, sending the more structured prompt to Gemini...")
response_structured = get_gemini_response(p_structured)
print("\nAnd here's Gemini's answer for the structured prompt:\n", response_structured)

if not os.path.exists('responses'):
    os.makedirs('responses')

file_intuitive = os.path.join('responses', 'response_intuitive.txt')
file_structured = os.path.join('responses', 'response_structured.txt')

try:
    with open(file_intuitive, "w", encoding="utf-8") as f:
        f.write(response_intuitive)
    print(f"\nSaved the intuitive prompt's response to: {file_intuitive}")

    with open(file_structured, "w", encoding="utf-8") as f:
        f.write(response_structured)
    print(f"Saved the structured prompt's response to: {file_structured}")
except IOError as e:
    print(f"Uh-oh! Had trouble saving the files: {e}")

print("\n--- All done with the experiment! ---")
```

Figure 3.1 Python code base for sending prompt to Gemini AI

3.2 Prompt Construction and Implementation

For the chosen task scenario, both $P_{intuitive}$ and $P_{structured}$ will be developed, and their generation process will be detailed.

A. Construction of the Intuitive Prompt ($P_{intuitive}$)

This prompt is formulated to represent a typical, natural language request, without a structural guidance from discrete mathematics.

- **$P_{intuitive}$ Text:** Explain Linked List in C for 2nd-semester informatics students. Provide 2 simple analogies, ensure the output is in English, and limit it to a maximum of 200 words.

B. Construction of the Mathematically Structured Prompt ($P_{structured}$)

The creation of this prompt is guided by the formal application of set theory, logic, and relations, aiming for maximal clarity and precision.

By formally representing each component of a prompt, such as the intended role of the AI, the main subject, the expected output elements, and any explicit constraints, as a distinct group of attributes, this framework helps reduce ambiguity. This decomposition using set theory serves as a fundamental step toward building prompts that are more reliable, reusable, and adaptable, which in turn improves the consistency and quality of AI-generated content across various tasks.

1. Set-Theoretic Decomposition with Universal Categories

   The task's requirements are broken down into distinct sets of elements, categorized by their universal function within a prompt. This approach allows for reusability across different tasks by defining "slots" for specific types of information.

   - Role = The persona or role the AI should adopt.
   - CoreTopic = The central subject matter and its key components.
   - Audience = The intended recipients and their understanding level.
   - RequiredElements = Mandatory components to be included in the output.
   - OutputConstraints = Restrictions on the output's form or content.
   - Exclusions = Elements or styles that must be explicitly omitted.

   For the "Linked List" scenario:

   - Role = {tutor_data_structure}
   - CoreTopic = {linked_list_concept, C_programming_language}
   - Audience = {informatics_students, semester_2_level}
   - RequiredElements = {comprehensive _explanation, two_simple_analogies}
   - OutputConstraints = {English_language, max_200_words}
   - Exclusions = {} (no specific exclusions for this prompt)

   The complete prompt, $P_{scenario}$, is conceptually represented as the union of these instantiated sets: $P_{scenario}$ = Role ∪ CoreTopic ∪ Audience ∪ RequiredElements ∪ OutputConstraints ∪ Exclusions.

2. Logical Formalization with Universal Operators

   The relationships between these instantiated elements are translated into logical propositions and operators. This step emphasizes that the logical structure (AND, OR, NOT, IF-THEN) is

universal, while the propositions themselves are specific to the current task.

- Let R be the proposition representing the Role assignment.
- Let T be the proposition representing the CoreTopic explanation.
- Let $A_{demographic}$ be the proposition ensuring Audience suitability.
- Let RE_1 and RE_2 be propositions for each specific RequiredElement
- Let OC_1 and OC_2 be propositions for each OutputConstraint
- Let EXi be propositions for each Exclusion.

The logical structure of the prompt for the "Linked List" scenario can be formalized as:
$(R \land T \land A_{demographic}) \land (RE_1 \land RE_2 \land RE_3) \land (OC_1 \land OC_2) \land (\text{no EXi})$.

This pervasive use of conjunction (∧) explicitly demands that *all* core explanation components, both analogies, the language, and the word count, must *all* be simultaneously satisfied. This formalization ensures unambiguous instructions and clearly enforces all constraints, minimizing ambiguity in how the AI interprets the prompt.

3. Relational Mapping with Universal Relationships
The interconnections and dependencies between prompt components are identified and mapped using universal types of relations, which apply across various prompts.

- applies_role_to: (Role, CoreTopic)
  The AI's persona applies to the explanation of the core topic.
- targets_to: (CoreTopic, Audience)
  The explanation of the core topic is aimed at a specific audience.
- contains_elements : (CoreTopic, Required Elements)
  The core explanation must contain the specified required elements.
- governed_by: (CoreTopic ∪ Required Elements, OutputConstraints)
  The entire explanatory content, including its components, is bound by output constraints.
- must_avoid: (CoreTopic ∪ Required Elements, Exclusions)
  Certain content must be excluded from the generated output.

For the "Linked List" scenario:

- applies_role_to: (tutor_data_structure, linked _list_concept)
- targets_to: (linked_list_concept, informatics _students_semester_2_level)
- contains_elements: (comprehensive_ explanation, two_simple_analogies)
- governed_by: (explanation, Indonesian _language), (explanation, max_200_words)

This mapping clarifies that the analogies are *part of* the explanation, and the language/word count constraints *apply to* the entire explanation, under the specified role and for the target audience. For example, if a Role applies to a CoreTopic, and the CoreTopic needs to be targeted_to an Audience, then the Role implicitly influences how the Audience is addressed. This ensures internal consistency across the entire prompt. This formal mapping helps confirm that all constraints are properly linked to the relevant parts of the output within a generalized framework.

Therefore, the $P_{structured}$ Text become:

- **$P_{structured}$ Text:**
  Act as a data structure tutor for 2nd-semester informatics students. Comprehensively explain the concept of Linked List in C programming language. The explanation must include exactly two simple analogies to aid understanding. The entire output must be in English and have a maximum length of 200 words.

IV. TESTING AND ANALYSIS

4.1 Qualitative Analysis of Results

After both $P_{intuitive}$ and $P_{structured}$ has been constructed, the prompts can be sent to Gemini AI using the python program.

```
C:\Users\Pelbi\Downloads\KodeMatdis.py
import google.generativeai as genai
import os

genai.configure(api_key="AIzaSyAQoGoBKBs4V3dLCJOKRRgnlbPSAckJaQg")

selected_model_name = 'gemini-2.5-flash'
print(f"Alright, we're going to use the model: {selected_model_name}")
model = genai.GenerativeModel(selected_model_name)

p_intuitive = (
    "Explain Linked List in C for 2nd-semester informatics students. "
    "Provide 2 simple analogies, ensure the output is in English, and limit it to a maximum of 200 words."
)

p_structured = (
    "Act as a data structure tutor for 2nd-semester informatics students. "
    "Comprehensively explain the concept of Linked List in C programming language. "
    "The explanation must include exactly two simple analogies to aid understanding. "
    "The entire output must be in English and have a maximum length of 200 words."
)

def get_gemini_response(prompt_text):
    try:
        response = model.generate_content(prompt_text)
        return response.text
    except Exception as e:
        print(f"Oops! Ran into an error trying to get a response from Gemini: {e}")
        return "ERROR: Couldn't get a response from the AI."

print("\n--- Kicking off our AI experiment! ---")

print("\nSending the intuitive prompt to Gemini...")
response_intuitive = get_gemini_response(p_intuitive)
print("\nHere's what Gemini came up with for the intuitive prompt:\n", response_intuitive)

print("\nNow, sending the more structured prompt to Gemini...")
response_structured = get_gemini_response(p_structured)
print("\nAnd here's Gemini's answer for the structured prompt:\n", response_structured)

if not os.path.exists('responses'):
    os.makedirs('responses')

file_intuitive = os.path.join('responses', 'response_intuitive.txt')
file_structured = os.path.join('responses', 'response_structured.txt')

try:
    with open(file_intuitive, "w", encoding="utf-8") as f:
        f.write(response_intuitive)
    print(f"\nSaved the intuitive prompt's response to: {file_intuitive}")

    with open(file_structured, "w", encoding="utf-8") as f:
        f.write(response_structured)
    print(f"Saved the structured prompt's response to: {file_structured}")
except IOError as e:
    print(f"\nUh-oh! Had trouble saving the files: {e}")

print("\n--- All done with the experiment! ---")
```

Figure 4.1 Python code for getting Gemini AI response

The responses generated by Gemini AI for both the intuitive ($P_{intuitive}$) and mathematically structured ($P_{structured}$) prompts were subjected to a rigorous qualitative comparison based on the predefined criteria. The experiment was conducted three times, and the results are as follows:

- **Response 1**

  Intuitive

  A **Linked List** is a dynamic data structure where elements are *not* stored contiguously in memory, unlike arrays. Instead, each element, called a **node**, contains two parts: its actual data and a "link" or **pointer** to the *next* node in the sequence.

  The list begins with a special pointer, often called `head`, which points to the very first node. The last node in the list points to `NULL`, signifying its end. In C, you typically define a `struct` for the node, containing the data and a pointer to a `struct Node` itself (e.g., `struct Node *next;`). This allows flexible memory allocation and efficient insertions/deletions.

  Analogies:

  1. **Treasure Hunt:** Each clue card (node) tells you where to find the *next* clue (pointer), until the last card reveals the treasure (NULL/end). You only need the first clue (head) to start.
  2. **Chain of Paper Clips:** Each paper clip (node) holds a piece of data and is linked to the *next* paper clip. You hold the first clip (head) to hold the entire chain.

  Structured

  A Linked List is a dynamic, non-contiguous data structure where elements, called nodes, are linked together using pointers. Unlike arrays, nodes aren't stored sequentially in memory, offering great flexibility.

  In C, a node is typically a `struct` containing data and a pointer to the *next* node:

  ```c
  struct Node {
     int data;        // The actual data
     struct Node *next; // Pointer to the next node
  };
  ```

  The `next` pointer holds the memory address of the subsequent node, forming a chain. The list's beginning is accessed via a 'head' pointer.

  Analogy 1: Imagine a treasure hunt. Each clue (node) contains a piece of information (data) and precise directions to the *next* clue. You follow this chain of directions until the treasure is found.

  Analogy 2: Or, consider a train. Instead of fixed carriages, each carriage (node) carries its cargo (data) and has a flexible coupling (pointer) that can attach to *any* other available carriage, forming a custom, extendable sequence.

  This structure allows efficient insertions and deletions anywhere, as you only update a few pointers, unlike arrays which require shifting elements.

- **Response 2**

  Intuitive

  A Linked List in C is a dynamic data structure where elements are not stored contiguously like arrays. Instead, each element, called a 'node', contains two parts: the actual data and a 'pointer' (or reference) to the next node in the sequence. The first node is called the 'head', and the last node's pointer points to `NULL`, signifying the list's end. This structure is excellent for efficient insertions and deletions, as you only update pointers, not shift elements.

  Analogy 1: A Train. Each train car is a node, carrying passengers (data). Each car is physically connected to the next, forming a chain.

  Analogy 2: A Scavenger Hunt. Each clue card is a node, holding a riddle (data) and directing you to the location of the *next* clue. You follow the chain of clues until you find the treasure (NULL).

  Structured

  Welcome, students! A **Linked List** is a fundamental linear data structure in C, storing elements, called **nodes**, non-contiguously in memory. Unlike arrays, nodes don't need adjacent storage.

  Think of it like a **scavenger hunt**: each clue (node) gives you the exact location of the *next* clue. You follow these clues one by one until you find the final prize (the end of the list). The list begins with a `head` pointer, pointing to the very first node.

  In C, a node is commonly defined using a self-referential `struct`:

  ```c
  struct Node {
     int data;          // The data stored in this node
     struct Node *next;  // Pointer to the next node
  };
  ```

To access elements, you start at the `head` and follow each `next` pointer sequentially until you reach a `NULL` pointer, signifying the list's end.

Another analogy is a **paper chain**: each paper loop (node) is separate but directly linked to the *next* one. You can easily add or remove loops anywhere without breaking the entire chain, showcasing linked lists' efficiency for insertions and deletions compared to fixed-size arrays. They offer dynamic memory allocation, adapting size as needed.

- **Response 3**

Intuitive

A **Linked List** is a dynamic data structure where elements, called **nodes**, are not stored contiguously in memory like arrays. Instead, each node holds two parts: your actual data and a pointer to the *next* node in the sequence. This allows flexible memory allocation, growing or shrinking as needed. The last node's pointer is typically `NULL`, marking the end of the list.

Analogy 1: Train Cars. Imagine a train where each car is a node. It carries cargo (data) and is physically coupled to the next car. To move through the train, you follow the connections from car to car, starting from the engine.

Analogy 2: Scavenger Hunt. Each clue card is a node. It contains a puzzle (data) and, crucially, directs you to the exact location of the *next* clue. You don't know all locations beforehand; you discover them step by step.

Structured

Alright class, let's demystify Linked Lists. Unlike arrays, which store data contiguously, a **Linked List** is a linear data structure where elements are not stored at adjacent memory locations. Instead, each element, called a **node**, contains two parts: the data itself and a pointer (or reference) to the next node in the sequence. The first node is the 'head'.

In C, we typically define a node using a `struct`, for example:

```c
struct Node {
   int data;
   struct Node *next;
};
```

This `next` pointer is crucial. It's like a **treasure hunt map**: each clue (node) tells you where to find the *next* clue, leading you step-by-step to the final treasure.

Another way to think about it is a **train**: each carriage (node) holds passengers (data) and has a coupling mechanism (pointer) that connects it directly to the *next* carriage. You can easily add or remove carriages anywhere without rebuilding the entire train. This dynamic nature and efficient insertions/deletions are key advantages over static arrays.

4.2 Assessment Framework and Result Comparison

A. Evaluation Criteria Recap

For clarity, the responses were assessed based on:

- **Precision and Clarity:** How accurately and unambiguously does the response address the core concept (Linked List in C) and associated details?
- **Completeness:** Are all specified components present, particularly the *two* simple analogies?
- **Adherence to Constraints:** Strict compliance with the 200-word limit and appropriateness for the target audience (second-semester informatics students).
- **Target Audience / Persona:** Appropriateness of content and tone for 2nd-semester informatics students and adoption of a "tutor" persona.
- **Coherence and Consistency:** Does the explanation flow logically, and are the analogies well-integrated?

B. Comparative Discussion

1. Precision and Clarity of Explanation
   - $P_{intuitive}$ (Responses 1, 2, 3): Consistently provided accurate and clear definitions of Linked Lists, nodes, data, pointers, head, and NULL. All included a C struct example, maintaining a good foundational explanation.
   - $P_{structured}$ (Responses 1, 2, 3): Demonstrated equally high precision and clarity in explaining the core concept. In Response 2, it notably included "self-referential struct" and mentioned malloc for dynamic allocation, details commonly taught in C programming courses for informatics students. This indicates a more specific and comprehensive explanation, directly targeting the technical context of the audience.*-

2. Completeness
   - $P_{intuitive}$ (Responses 1, 2, 3): Successfully provided exactly two simple analogies in all three responses (e.g., "Treasure Hunt" & "Chain of Paper Clips" in Response 1;

"Train" & "Scavenger Hunt" in Response 2 and 3).

- $P_{structured}$ (Responses 1, 2, 3): Consistently delivered exactly two simple analogies in every response. This strict adherence directly stems from the explicit logical formulation within the structured prompt, which specified "must include exactly two simple analogies," leaving no room for ambiguity in quantity.

3. Adherence to Constraints
   - Word Count (Max 200 words): Both prompt types showed excellent adherence. All six responses were well within the 200-word limit, demonstrating effective constraint management by the AI.
   - Language (English Output): All six responses were consistently generated in English, successfully fulfilling the explicit language instruction in both prompt types. This confirms the model's ability to adhere to output language specifications when clearly stated in the prompt.

4. Target Audience / Persona
   This aspect most clearly highlights the superiority of the structured prompt:
   - $P_{intuitive}$ (Responses 1, 2, 3): While informative, the tone remained generally descriptive. The responses provided factual explanations of Linked Lists, but did not explicitly adopt a teaching persona or directly engage the target audience ("2nd-semester informatics students") beyond simply presenting the information.

   - $P_{structured}$ (Responses 1, 2, 3): Consistently demonstrated a stronger and more deliberate effort towards fulfilling the specified "data structure tutor" role and addressing "2nd-semester informatics students" directly. This was evident in openings like "Welcome, students!" (Response 2) and "Alright class, let's demystify Linked Lists" (Response 3), which were absent in intuitive responses, showcasing the successful guidance from the explicit role. Furthermore, the instruction to "Comprehensively explain... in C programming language" (from CoreTopic) led to more nuanced C-specific details, such as the mention of malloc and self-referential struct in Response 2, making the content highly relevant and directly applicable for the target students.

5. Coherence and Consistency
   - $P_{intuitive}$ (Responses 1, 2, 3): The explanations were logically sound, but analogies were often presented as a separate, distinct section under a "Analogies" heading, sometimes feeling appended rather than fully integrated into the narrative flow.

   - $P_{structured}$ (Responses 1, 2, 3): Demonstrated superior coherence and consistency. Analogies were often woven more seamlessly into the explanation, contributing to a better logical flow. For example, in Response 3, the explanation transitioned directly from discussing the next pointer to "It's like a treasure hunt map," followed by a smooth transition to the next analogy and then linking back to the advantages.

C. Discussion of Observed Benefits

The response comparative analysis clearly substantiates the significant advantages of the mathematically structured prompt over its intuitive counterpart. By explicitly defining the AI's Role (e.g., "Act as a data structure tutor") and specifying the Audience (e.g., "2nd-semester informatics students"), by utilizing **Relational Mapping with Universal Relationships** to dictate how these elements interact, the structured prompt consistently guided the AI to produce outputs more tailored in tone and direct engagement, resulting in responses that felt genuinely more instructional and well-suited for the target learners. This precise definition also enhanced contextual depth and specificity, as seen in the inclusion of highly relevant details like malloc and self-referential struct directly linked to CoreTopic for C programming. Furthermore, the **Logical Formalization with Universal Operators** ensured the unambiguous fulfillment of specific requirements, such as providing "exactly two simple analogies."

These consistent superior performance across multiple responses highlights the robustness and predictability of the structured approach, transforming prompt engineering into a more systematic and reliable process compared to less formalized, intuitive methods.

V. CONCLUSION

The empirical assessment carried out using Gemini AI clearly confirmed the effectiveness of the structured approach presented in this study. Through comparative analysis, it was consistently shown that prompts constructed

using set-theoretic decomposition produced markedly better results in terms of accuracy, coherence, and contextual relevance. Importantly, these structured prompts gave more precise control over how the AI follows the assigned role, generated content that was well-targeted for the intended audience (such as informatics students studying C), and incorporated analogies in a fluid and meaningful manner, thereby improving the educational quality of the responses. This structured technique also reduced ambiguity, helping generate more consistent and reliable outputs from the language model.

In summary, this research redefines prompt engineering as a structured and scientifically informed process, rather than relying solely on intuition. The set-theoretic model proposed here can be a useful and flexible tool for both developers and researchers in creating optimized prompts for various use cases. Future research may explore how this method can be applied to more complex tasks, including those involving multiple types of input such as text and images, therefore evaluating its scalability across different LLM architectures, and building automated systems that utilize this formal structure to streamline prompt creation and refinement.

VI. APPENDIX

The complete source code used by the program to send prompt to Gemini AI is available on GitHub. Access code repository here: https://github.com/Hagon47/Gemini-AI-API/blob/8e7f910f90e810a27d72e630198321048c829e5e/KodeMatdis.py.

VIDEO LINK AT YOUTUBE

Further explanations of the research are available in the video link: https://youtu.be/_8ogG93CK6s

VII. ACKNOWLEDGMENT

The author would first like to express deep gratitude to God Almighty for the strength, perseverance, and clarity of mind that enabled the completion of this paper. Furthermore, sincere appreciation is extended to Mr. Dr. Ir. Rinaldi Munir, M.T., the lecturer of IF1220 Discrete Mathematics, for his insightful guidance and unwavering dedication to sharing knowledge, which has played a significant role in the development and completion of this work.

VIII. REFERENCES

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/01-Logika-2024.pdf (Accessed 16 June 2025)

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025-2/02-Himpunan(2025)-1.pdf (Accessed 16 June 2025)

[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025-2/03-Himpunan(2025)-2.pdf (Accessed 17 June 2025)

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/11-Deretan,%20rekursi-dan-relasi-rekurens-(Bagian2)-2024.pdf (Accessed 17 June 2025)

[5] White, Jules, et al. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. Vanderbilt University, 2023. https://file.mixpaper.cn/paper_store/2023/681177f8-cd15-4e0f-a23b-997c6b9f9dd2.pdf. (Accessed 18 June 2025)

IX. STATEMENT

I hereby declare that this paper is my own work, not a paraphrase or a translation of someone else's paper, and definitely not plagiarism.

Bandung, 19 Juni 2025

Reynard Nathanael 13524103