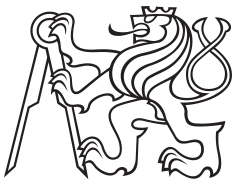
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Artificial Intelligence Center

Meta-prompts for LLM Prompt Optimization

abcd

Vojtěch Klouda

Supervisor: Ing. Jan Drchal PhD.
Field of study: Artificial Intelligence
Subfield: Natural Language Processing
May 2025

Acknowledgements

=)))

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 10. May 2025

Abstract

TODO

Keywords: language model, optimization

Supervisor: Ing. Jan Drchal PhD. Resslova 307/9 Praha, E-322

Abstrakt

TODO

Klíčová slova: jazykový model, optimalizace

Překlad názvu: abcd — abcd

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Background

The central focus of this work is an instance of an LLM, denoted $\mathcal{M}$. When appropriate, we can differentiate between instances with a lower index, specifying its purpose. For example, when using separate LLM instances for optimizing and task-solving, we will denote them $\mathcal{M}_{optim}$ and $\mathcal{M}_{solve}$ respectively. This way, we put emphasis on the fact we can choose a different LLM provider and hyperparameters for each instance.

In general, $\mathcal{M} : \mathbb{T} \times \mathbb{T}$ is a stochastic (for a positive sampling temperature) mapping on the space of text sequences $\mathbb{T}$. A prompt $p \in \mathbb{T}$ is a text sequence that, when inputted into an LLM, produces an output

$$y \sim \mathcal{M}(p). \tag{1.1}$$

We can use LLMs to solve a general task

$$t \in \mathcal{D} \,|\, \mathcal{D} = \{(q_1, g_1), (q_2, g_2), ..., (q_n, g_n), \}, \tag{1.2}$$

where $\mathbb{D}$ is a dataset consisting of n pairs of queries q and gold-labels g. For open-ended tasks, the gold-label does not exist.

We can further define a prompt as

$$p = \mathbf{i}(q), \tag{1.3}$$

where $\mathbf{i} \in \mathbf{T}$ is a set of text instructions into which a task query q is inserted.

Algorithm 1: General optimization loop

Input: Initialization Operator $\mathcal{O}_I$, Selection Operator $\mathcal{O}_S$, Expansion Operator $\mathcal{O}_E$, Termination Condition Φ_{stop}
Output: Optimized Population $\mathcal{P}$
Data: $\mathcal{P} \leftarrow \mathcal{O}_I$
`// Initialize the population`
1 **while** $\neg\Phi_{stop}(\mathcal{P})$ **do**
  `// Selection and Expansion Steps`
2 $\mathcal{P}_{\text{selected}} \leftarrow \mathcal{O}_S(\mathcal{P})$
3 $\mathcal{P}_{\text{expanded}} \leftarrow \mathcal{O}_E(\mathcal{P}_{\text{selected}})$
4 $\mathcal{P} \leftarrow \mathcal{P}_{\text{expanded}}$ `// Update the population`
5 **return** $\mathcal{P}$ `// Return the optimized population`

Next, we move onto the optimization notation. In Algorithm 1 we can see the general outline of a population-based optimization method. The

initialization operator $\mathcal{O}_I$ creates an initial population of individuals $\mathcal{P}$. Then, in each step, a selection operator $\mathcal{O}_S$ selects a portion of the population according to some criteria. These selected individuals are then used by the expansion operator $\mathcal{O}_E$ to create new individuals. This process continues until a termination condition Φ_{stop} is reached.

Chapter 2

Literature

2.1 Inference-time scaling

Inference-time scaling or test-time scaling is a paradigm that has gained traction in recent years with the advent of dedicated reasoning models[1][2]. As opposed to training-time scaling, where the performance of models scales with training times, model parameter counts and dataset sizes[3], inference-time scaling aims to improve performance by dedicating more resources to each inference call.

At their heart LLMs are probabilistic models over sequences and to generate a sequence they employ generation algorithms. Welleck et al.[4] provide an overview of these generation algorithms and then frame more advanced inference-time techniques as meta-generations, or strategies that employ sub-generators. Most generation algorithms attempt to find either highly probable sequences (MAP algorithms) or sample from the model's distribution. The simplest MAP algorithm is greedy decoding, which recursively finds the next token with the highest probability in the distribution. An example of algorithms that sample from the model's distribution is the ancestral sampling algorithm.

A generalization of greedy decoding is the beam search algorithm which maintains a structure of possible prefixes and each step expands them and scores them. An example[5] of a beam search algorithm can identify decoding branches where the model employs a reasoning chain to solve a given task. Authors of this algorithm found that answer tokens found in the decoding paths with a reasoning chain have greater token probabilities. This means that the model shows greater confidence in its answer having reasoned about it beforehand. In general beam search improves on simple greedy decoding but at a high computational cost[4].

Another class of generation algorithms are those which interpolate between more categories of sampling algorithms. Temperature sampling, which outperforms other adapters in input-output tasks like code generation and translation, is an interpolation between greedy sampling and uniform sampling. Interpolating between ancestral sampling and simple greedy sampling gave rise to decoding algorithms such as nucleus, top-k and η- and ϵ-sampling. When we require a structured output, for example a JSON data structure following a JSON schema, we can utilize parser-based decoding, which enforces a structural requirement. This can however come at worsened performance when using inflexible templates.

These low-level generator can be interconnected into more complex technique, which Welleck et al. call meta-generators[4]. We will stick to their terminology and discuss different sequence-level meta-generation algorithms. We will omit further discussion of token-level methods as they are irrelevant to the main topic of this thesis. These strategies can be divided into the categories of chained, parallel, step-level, and refinement-based meta-generators.

2.1.1 Chained meta-generation

Chained meta-generation is the composition of several subgenerators in sequence. These can be LLM calls or other functions that use previous inputs, such as a code execution function[6] or a tool for interaction with and arbitrary environment or a data source[7]. The subgenerators can be implemented as several LLM calls or with a single call given sufficient instructions in the prompt. [8] Some examples include Program of Thoughts[6], ReAct[7] and Chain-of-Thought[9][10] techniques.

In its essence, the model is a left-to-right text completion engine. We can make the analogy with human thinking modes, where it is said that humans have a fast automatic "System 1" mode and a slow and deliberate "System 2" mode[11]. In direct-QA mode, the LLM can underestimate the difficulty of the task[5] and stay in the "System 1" thinking mode. Simple greedy decoding paths mostly do not contain a reasoning chain[5], which means the model tends to make a guess, staying in "System 1". By crafting a good prompt that instructs the model to reason we can shift the model from "System 1" to "System 2" thinking. Another reason for the effectiveness of chained generation is that in LLM training some concepts and variables are observed more frequently than others[12]. This discrepancy hurts performance in direct-QA scenarios where the relevant variables are rarely seen together in training. With CoT, models can incrementally chain known dependencies and bridge conceptional gaps.

Chain-of-thought

Chain-of-Thought[10] (CoT) is a LLM prompting technique that works by inducing a coherent series of intermediate reasoning steps that lead to the final answer for a problem, thereby increasing computation time. Upon its discovery, it brought a dramatic performance increase on arithmetic tasks, where models previously struggled. This enhanced capability comes with the

cost of longer and more computationally expensive outputs[13] and is more noticeable for more complicated problems[10].

CoT can been elicited by prompting techniques - few-shot with steps demonstrations or zero-shot with specific instructions[5] First CoT methods[10] involved one/few-shot prompting, Although effective, this requires human engineering of multi-step reasoning prompts. This method is also highly sensitive to prompt design with performance deteriorating for mismatched prompt example and task question types[9]. For this method, authors found that CoT is an emergent capability of model scale and did not observe benefits for small models[10]. where the prompt included examples of CoT reasoning in the prompt in facilitate a reasoning chain response.

On the other hand, zero-shot prompting induces a reasoning chain with a simple prompt like "Let's think step-by-step", making it versatile and task-agnostic[9]. Similar prompts also improve reasoning performance and some research[14] has been done on finding the optimal CoT prefix prompt.

Apart from prompting, CoT can be elicited and improved by model training or tuning. This method, requiring a significant amount of reasoning data[5], has gained traction with the development of dedicated reasoning models like OpenAI's o1[1] or Deepseek-R1[2]. Using methods such as supervised fine-tuning (SFT) or reinforcement learning (RL), the model is trained to automatically produce longer reasoning chains, often bound in dedicated "thought" tags or tokens. These models have shown significant performance boosts on reasoning benchmarks[1][2].

Models similar to o1 all primarily extend solution length by self-revision[15]. After finishing a thought process, the model tries to self-revise, which is marked by words such as "Wait" or "Alternatively". The model then tries to spot mistakes or inconsistencies in its reasoning or propose an alternative solution. Self-revision ability is thus a key factor in the effectiveness of sequential scaling for reasoning models[15].

Longer reasoning chains mean more computing power spent at inference. How far can we take this sequential scaling? In their study, Zeng et al.[15] argue that longer CoTs do not consistently improve accuracy of reasoning models. Furthermore, they find that the average length of correct solutions is shorter than that of incorrect ones.

Because self-revision accounts for most of the CoT length, the effectiveness of the method relies on the model's ability to self-revise. Authors of this paper argue that the self-revision ability of models is insufficient as they demonstrate limited capacity to correct their answers during self-revision. Some models on some tasks are even more likely to change a correct answer to an incorrect one than vice-versa.

Further research by Liu et al.[16] suggests that for some tasks CoT can be detrimental. Their experiments proved their hypothesis that CoT hurts performance on tasks where humans do better without deliberation and where the nature of LLM, like the much greater context memory, does not provide an advantage over human thinking. This phenomenon was observed on tasks like facial recognition, implicit statistical learning or pattern recognition.

2.1.2 Parallel meta-generation

Parallel meta-generation involves multiple generations concurrently. The final answer can then be chosen - with a reward model or with voting - or constructed from the ensemble of generations[4].

Parallel meta-generation allows weaker models to outperform bigger and more expensive models[13]. This can sometimes reduce cost as multiple samples with a smaller model are cheaper than a single sample with a more capable model. This is helped by the fact that parallel sampling can make use of batching and other system throughput optimization available for parallel inference[13].

One of the simplest such techniques is self-consistency[17] (SC), a method which builds upon CoT to aggregate answers from diverse reasoning chains and selects the best one based on majority voting. It significantly improves accuracy in a range of arithmetic and commonsense reasoning tasks at the cost of increased computation expenditure[17]. The effectiveness of SC with majority-voting comes from the fact that, for tasks with objective answers, there are often more ways to be wrong than to be right.

For our next discussion of SC and related methods we will compare the terms *coverage* $\mathrm{C}_{\mathbb{D}}$ and *accuracy* $\mathrm{A}_{\mathbb{D}}$ for a dataset $\mathbb{D}$. Given a language model $\mathcal{M}$, a task query $q_k \in \mathbb{D}$ and a task instruction $\mathbf{i}$, we can define the generation collection of length n as

$$Y_k = \{y_{jk} \mid j \in 1, ..., n\}, \tag{2.1}$$

$$y_{jk} \sim \mathcal{M}(\mathbf{i}(q_k)). \tag{2.2}$$

For objective tasks we can check the correctness with a metric $\mathcal{G}$

$$\mathcal{G}_k(y_{jk}, q_k) = \begin{cases} 1.0 & y_{jk} \text{ is the correct answer for } q_k \\ 0.0 & y_{jk} \text{ is an incorrect answer for } q_k. \end{cases} \tag{2.3}$$

To choose the final answer, we will define an answer selection function $\mathcal{S}(Y)$. This can be a majority vote selection function or some reward-based method. We can now define *coverage* $\mathrm{C}_{\mathbb{D}}$ and *accuracy* $\mathrm{A}_{\mathbb{D}}$ as

$$\mathrm{C}_{\mathbb{D}} = \frac{1}{|\mathbb{D}|} \sum_{q_k \in \mathbb{D}} \max_{j=1,...,n} \mathcal{G}_k(y_{jk}, q_k) \tag{2.4}$$

$$\mathrm{A}_{\mathbb{D}} = \frac{1}{|\mathbb{D}|} \sum_{q_k \in \mathbb{D}} \mathcal{G}_k\left(\mathcal{S}(Y_k), q_k\right). \tag{2.5}$$

In plain language, coverage is the fraction of the tasks where at least one sample results in a correct answer, whereas accuracy is the fraction of the tasks where a correct answer is selected by the algorithm as a final answer.

It is easy to see why coverage might rise as we increase the amount of samples n in SC generation. One could imagine that as letting students answer with their top n guesses for each question on a test. Indeed research[13] has found that the relationship of coverage and the number of samples can be modeled by an exponentiated power law, suggesting a scaling law for inference similar to the training scaling laws[3].

However coverage alone is not enough to paint the complete picture. What good is it to have a large collection which contains a correct answer if we cannot verify which one is correct. Parallel scaling with large sample collections is only useful if the correct samples in a collection can be identified[13][15]. The accuracy gain of SC tends to saturate quickly as we increase the number of paths[17]. Although coverage rises, it diverges[13] from accuracy as the algorithm is unable to select the correct answer from the collection. This highlight the necessity to develop better answer selection mechanisms than simple majority voting and automatic answer verification methods.

Zeng et al.[15] make use of the fact that correct solutions have shorter CoT on average and develop a length-weighted majority vote that outperforms simple majority voting on the challenging math benchmarks. GLaPE[18] is a method for gold label-agnostic evaluation which makes use of the fact that incorrect answers tend to be inconsistent.

2.1.3 Step-level meta-generation

Maintaining the terminology of Welleck et al.[4], step-level meta-generation algorithms implement search on the generation state-space. This can be done on the token level or on the level of longer sequences, but in this section we will focus on the latter.

Previously discussed inference-time scaling techniques all relied on sequential or parallel linear thought processes. They do not explore different continuations within a thought process and do not make use of planning, lookahead, or backtracking[11]. These methods also do not allow combining the flow of reasoning upon discovering new insights, something humans utilize when solving problems[19].

By generalizing CoT[10][9] into a tree structure, Yao et al.[11] present Tree of Thoughts (ToT), a technique which maintains a tree of thought. In this tree, each node is a thought in a form of a coherent language sequence, serving as an intermediate step in the reasoning process. For traversing the tree, a general tree-search algorithm, such as breadth-first or depth-first search, can be employed.

An important parameter in ToT is the branching factor. Unlike the standard tasks typically tackled by tree search algorithms where the number of possible actions at each node is finite, each call to LLM can yield a new output even for the same input, making each node's branching factor theoretically infinite[20]. Misaki et al.[20] argue that fixed-width multi-turn methods exhibit diminishing gains and develop a tree search method with an adaptive branching factor, leading to a more balanced exploration and exploitation capability.

Although ToT allows for planning and backtracking from unpromising thought chains, its structure is still too rigid[19]. For example, it is not possible to combine thoughts from independent branches from the tree. Graph of Thoughts[19] (GoT) is a framework that models the reasoning process as a heterogenous directed graph where each vertex is a thought containing a (partial) solution and edges are dependencies between these thoughts[19].

In both chain- and tree-based inference-time scaling methods, a substantial amount of compute power is allocated to processing historical information that is not beneficial to the reasoning process. To alleviate this, Atom of Thoughts[21] (AoT) iteratively decomposes the current question into a directed acyclic graph. The graph consists of subquestions which, depending on whether they have dependencies, are dependent or independent. All the independent questions can be answered directly and their answers added combined as context with the remaining subquestions to be contracted into a new current question.

2.1.4 Refinement meta-generation

The last category of meta-generation algorithms are refinement algorithms. Refinement algorithms work by alternating between generation and refinement. The refiner generates a revised version of the output based on past versions and additional information such as intrinsic or extrinsic feedback or environment observations[4]. Intrinsic refinement comes from the model inspecting its own answers. As we discussed in 2.1.1, models struggle with self-revision and rarely modify their answers in long reasoning chains. Feedback from general models is ineffective compared to dedicated feedback models or other quality feedback sources[22].

For extrinsic refinement, the model can utilize external information which can lead to a potential gain with refinement[4]. One example of a refinement-based framework, Reflexion[23], converts binary or scalar feedback from the environment into verbal feedback in the form of a textual summary. This feedback is then added as additional context for the LLM agent, e.g. CoT or ReAct module, in the next episode. Reflexion improves performance over strong baselines on sequential decision making, reasoning and programming tasks[23].

2.2 Prompt engineering

Maintaining the notation outlined in 1.1, prompt $p = \mathbf{i}(q)$ is a combination of a set of instruction $\mathbf{i}$ and a query q.

By prompt engineering we mean crafting a instruction set which transforms the query into a result according to our task requirements. Our task requirements can for example be

- obtaining the correct answer for a mathematical problem
- fixing a bug in a code base
- explaining the contents of an image.

Each of these tasks needs a separate instruction set $\mathbf{i}$ which can then be used with multiple queries, representing specific task instances. This signifies a shift from the training and fine-tuning paradigm, where a base model is first trained on a large corpus of data and then adapted for a specific task with supervised fine-tuning. This process requires a substantial amount of training data and computation power, making specialized LLMs unsuitable for many users and for applications, where extensive data collection is infeasible.

Since the inception of modern LLMs, prompt engineering has evolved into a field of its own. Current LLM systems, often containing multiple chained and interlinked models, require robust and well thought-out prompts at each step. Indeed, in many modern LLM applications, prompts have become programs themselves[24], marking a huge leap from the basic text messages of the early LLM days.

In this section, we will briefly cover the most notable prompt engineering techniques, which we will then be able to utilize in our study of automatic prompt optimization.

2.2.1 Components of a prompt

We can dissect a prompt into several components[25].

- **Directive:** The main task of the prompt, e.g., *"Write an email to a coworker."*
- **Context:** Everything necessary or beneficial to completing the directive, e.g., *"I was supposed to send a report to my boss, but I forgot."*
- **Examples:** How you would have solved a similar task, e.g. a past email on a similar topic.
- **Output specifications:** Style and format instructions, e.g., *"Respond with three paragraphs in formal style with tasteful emojis."*

Although flexible and sometimes blended together, high-performing prompts often follow this structure and order. In more technical applications, it is beneficial to use tags or delimiters to explicitly separate components. Models often reward prompts with a more code-like structure[26]. We now discuss each component in detail.

Directive

The directive should be a clear and objective description of the task. Specific requirements that narrow the scope should be avoided and left for other components. A good rule of thumb is to treat the directive as a subject to an email.

In cases where the prompt serves as a prompt template, meaning it can be reused with various data points, the directive can include a placeholder. For example consider this directive

```
Write a limerick about {topic}.
```

This directive, and the prompt to which it belongs, could be reused for multiple values of *topic*.

Context

Context should provide all the background information relevant to the task at hand. The user can include more information about why they are using the

model for the task, define the target audience or attach relevant documents. For example, a prompt that asks the LLM to explain a code snippet:

In this case, the first part is the directive and the second part is the context, which specifies the user's situation. Without the context, the model might explain each line of the snippet in too much detail and not explain the dictionary unwrapping operator specifically.

The context can become the endpoint for retrieval pipelines, which search a data source for relevant documents, or memory mechanisms, which gather personal information about the user from other conversations.

````
Explain this Python code snippet.
```python
    user_info = {'name': 'Alice', 'age': 30, 'city': 'New York'}
    def greet_user(**kwargs):
        print(f"Hello, {kwargs['name']} from {kwargs['city']}!")
    greet_user(**user_info)
```

I am a beginner to Python programming.
I understand the function definition and that user_info is a dictionary,
but I don't know what the double asterisk does in the function call.
Can you explain how the double asterisk
works in this context and what it does step by step?
````

Another possible feature of the context component is role-assignment. We can instruct the model to adopt an identity or an expertise level. For example, we can tell the model something like "You are a experienced business analyst". This primes the model to use a more technical language in its response.

Examples

Sufficiently large models trained on massive datasets a Prompts are distinguished based on the number of included examples.

Research[27] has shown that with growing model size the knowledge-generalizing ability of the model increases. Instead of expensive fine-tuning

Prompting Type	Description
Zero-shot Prompting	Prompt has no examples. Model relies on its pre-trained knowledge.
One-shot Prompting	Prompt has one example to guide the model.
Few-shot Prompting	Prompt includes a few examples (typically 2 to 5).

Table 2.1: Comparison of Zero-shot, One-shot, and Few-shot Prompting

models can reuse knowledge from pre-training and solve many tasks when provided just by a few examples.

Few-shot prompting highlights that LLMs can be seen as powerful pattern-completion engines. [28]

Providing a prompt of examples from a distribution can condition the LLM to generate further high-probability examples from that distribution [28]

Output specifications

2.3 Prompt optimization

In a previous section on inference-time algorithms, we outlined several methods for improving the performance of LLM generation and contrasted it to the traditional training-time scaling paradigm. In this section, we will discuss prompt optimization. This technique fits in neither training- or inference-time scaling techniques, as the computation allocated on optimizing our prompts can then be amortized over multiple uses of the prompt. We will refer to this as compile-time[24] scaling.

Prompt engineering is a language generation task requiring complex reasoning to identify model error's and remedy them by modifying the prompt [29] Creating effective prompts often requires substantial trial-and-error experimentation and deep task-specific knowledge [30] Compile-time optimization is carried out only once before deployment thus amortizing the optimization cost over multiple uses of the prompt program. [24]

2.3.1 Soft prompt tuning

Prompts for models which allow access to gradients, which is not the case for proprietary models accessed via APIs, can be optimized in the high-dimensional embedding space.

This makes the optimization problem continous. Soft prompts however pose the problem of interpretability and are non-transferable across different LLMs [31].

Continuous prompt-optimization techniques, although effective, require parameters of LLMs inaccessible to black-box APIs and often fall short of interpretability. [32]

2.3.2 Discrete prompt tuning

The area of optimizing prompts discretely while utilizing language models as optimization operators has attracted significant research interest in recent years.

Natural language prompt engineering is particularly interesting because it is a natural interface for humans to communicate with machines, but plain language prompts do not always produce the desired result. [33]

Natural language program synthesis search space is infinitely large. [33]

Discrete tokens are not amenable to gradient-based optimization and brute-force search has an exponential complexity. [31]

Unlike perturbing e.g. network weights continuously which predictably generates small changes in functionality, perturbing code requires discrete changes which often dramatically change functionality. [34] **Reinforcement learning**

Heuristics based on "enumeration (e.g., paraphrasing)-then-selection" do not explore the prompt space systematically [31]

RLPrompt trains a policy network which is inserted as a MLP layer into a frozen compact model. [31]

Agent chooses the next token at each step using the previous tokens according to a learned policy. When the prompt is completed, the agent receives the task reward. [31]

The policy network can also take another inputs, leading to input-specific prompts. [31]

Meta-prompts are flexible but studies lack principled guidelines about their design. [35]

Reproduces key model parameter learning factors - update direction and update method - in LLMs to seek theoretical foundations. [35]

OPRO[14] and APO[36] introduced analogical "gradient" forms. [35]

Analogical momentum forms inspired by the momentum method involve including the optimization trajectory in the meta-prompt. To fit into the context limit and reduce noise, trajectory can be summarized or k most recent/relevant/important gradients can be retrieved. [35]

To mimic effects or learning rate, prompt variation can be limited by edit distance (maximum words to be changed). Warm-up and decay strategies can be applied to this constraint. [35]

New prompt can be created by editing a previous prompt or generate a new one by following a demonstration. [35]

In an experiment on BBH, authors found that optimization without reflection performs better and the best momentum method being relevance. For prompt variation control, the best combination was cosine decay and no warm-up. [35]

Summarization-based trajectory is less helpful because it tends to only capture common elements. [35]

Task input-output examples are beneficial in the meta-prompt to provide additional context to the LLM to understand the task. [35]

GPT-4 can consistently find better task prompts than GPT-3.5-turbo, which suggests the need for a capable model as the prompt optimizer [35]

Trajectory-based methods perform very well possible because trajectory helps the prompt optimizer pay more attention to the important information instead of the noise in the current step. [35]

APE LLMs are used to construct a good set of candidate solutions by inferring the most likely instructions from input/output demonstrations. [33]

Local search around the best candidates by resampling - asking the LLM to paraphrase the candidate prompt - this however only provides marginal improvements over just choosing the best-performing prompt from instruction induction. [33]

APE was used to improve on Zero-Shot-CoT [9] universal "Let's think by step" prompt"on GSM8k.[33]

Prompt to the LLM optimizer is called the meta-prompt and includes previous prompts with their training accuracies sorted in ascending order along with the task description and training set samples. [14]

The main advantage of LLMs for optimization is their ability of understanding natural language, which allows people to describe their optimization tasks without formal specifications. [14]

Motivated by linear regression and TSP and on small-scale traveling salesman problems, OPRO performs on par with some hand-crafted heuristic algorithms. [14]

Optimization stability can be improved by generating multiple solutions when relying on random ICL samples. [14]

To balance between exploration and exploitation, LLM sampling temperature can be tuned. Lower temperature encourages exploitation in the local solution space and higher temperature allows more aggressive exploration of different solutions. [14]

Only the top instructions are kept in the meta-prompt to fit in the LLM context limit. [14]

New outstanding solution is usually found only all the prompts are of similar quality: first all the worse prompts are purged and substituted by a prompt similar to the current best. [14]

Semantically similar instructions have vastly different performance on GSM8k: "Let's think step by step." achieves accuracy 71.8, "Let's solve the problem together." has accuracy 60.5, while the accuracy of "Let's work together to solve this problem step by step." is only 49.4. [14]

Symbolic search With increasing complexity of prompt structure, many prompt optimization techniques are no longer applicable. [24] Symbolic prompt programs (SPPs) can be represented as directed acyclic graphs where nodes are functions (subprograms) and edges indicate call dependencies. [24]

Search space can be defined in two ways, as an enumerative search, where a small number of options is known beforehand, and iterative search, where a large search space is explored with iterative search strategies. [24]

Textual gradients

Naturally there are no gradients in the text space but some researchers try to emulate them using reflection-based operators.

APO mirrors the steps of gradient descent within a text-based Socratic dialogue substituting differentiation with LLM feedback and backpropagation with LLM editing [36]

Beam search is an iterative optimization process where in current prompt is expanded into many more candidates in each iteration and a selection process decides which will be used in the next iteration. [36]

Expansion first uses gradients to edit the current prompt and then explores the local monte-carlo search space by paraphrasing the editions [36]

To limit the computation used on evaluating prompts, an approach inspired by best arm identification in bandit optimization is utilized. [36]

Applying previous iterative prompt optimization methods, based on prompt+score pairs, to text generation tasks is challenging due to the lack of effective optimization signals. [37]

Critiques and suggestions, written in natural language, are more helpful for prompt improvement than a single score.[37]

CriSPO uses prompt+score+critique triples for next candidate generation. [37] Unlike APE [36] prompt generation is decoupled from suggestions and a history of critiques and suggestions as packed into the optimizer for a more stable optimization. [37]

CoT is applied in optimization by first asking to compare high-score prompts to low-score ones and draft general ideas. [37]

Critique-based optimization explores a larger space, which is indicated by lower similarity of the prompts in lexicons and semantics.[37]

CriSPO outperforms OPRO [14] both on summarization and QA tasks and

metaprompt allows for creating ICL and RAG template prompts. [37]

DSPy optimizers

Most prompt optimizer approaches do not apply to multi-stage LLM programs where we lack gold labels or evaluation metrics for individual LLM calls. [38]

Proposing a few high-quality instructions is essential due to the intractably large search space. [38]

Uses a surrogate Bayesian optimization model, which is updated periodically by evaluating the program on batches, to sample instructions and demonstrations for each stage of the LLM program [38]

Optimizing demonstrations alone usually yields better performance than just optimizing instructions, but optimizing both yield the best performance. [38]

Optimizing instructions is most valuable for tasks with subtle conditional rules not expressible by a few examples. [38]

For LLM programs, it is beneficial to alternate between optimizing weights (fine-tuning) and optimizing prompts. [39]

Evolutionary optimization

There is considerable synergy potential between the fields of evolutionary computation and deep learning. [34]

LLM-based variation operator

LLMs trained on code can suggest intelligent mutations and thus sidestepping many of the challenges in evolving programs. [34]

Genetic programming still offers an advantage when the programming task is far from the training distribution of the LLM. [34]

Genetic programming can in principle evolve in any space. [34] this can be connected with the idea that promp

Topic of mutation is guided by previously chosen "commit message" this is like a pseudogradient. [34]

Evolution with language models can be used as a way of generating domain data for downstream deep learning where it did not previously exist. [34]

The pattern-completion ability of few-shot prompting can be leveraged to create a form of intelligent evolutionary crossover. [28]

Performance at in-context learning improves with model scale, implying that methods relying upon this capability will benefit from continuing progress in LLM training. [28]

Advent of large, pretrained foundation models marked a significant step in the paradigm of evolutionary recombination with deep generative models, where these new models can be directly leveraged without additional training and allow for searching more abstract spaces. [28]

Next-token prediction naturally lends itself to creating an evolutionary variation operator. [28]

LLM variation operator should be capable of generating meaningful variation for any text representation that has moderate support in the training set, meaning it is basically domain-independent. [28]

LLM crossover acts like an EDA in that it builds a probabilistic model of parents from which the children are sampled. [28]

LLM crossover becomes more similar to an EDA as the number of parents increases. [28]

LLM crossover can express any genetic operator even with small parent sets by fine-tuning or through effective prompting schemes. [28]

Simultaneously evolving the solution x along with the LLM and the prompting mechanism could be a powerful paradigm for more open-ended systems. [28]

Input arranged in ascending fitness order prompts the model to generate output that follows an ascending fitness trend. [28]

Evolutionary algorithms have been shown to be effective in search spaces with millions and billions of variables, which are inaccessible to LLM crossover due to the size of the LLM context window. [28]

The level of stochasticity in LMX can be controlled by the softmax temperature parameter, which can be seen as analogous to a mutation rate parameter in a traditional EA [28]

Frameworks Building upon the inherent ability of LLMs to paraphrase (mutation) and combine (crossover) text, an interesting intersection of traditional evolutionary algorithms and modern LLMs has formed.

Sequences of phrases can be regarded as gene sequences in typical Evolutionary algorithms. [32]

Considers two widely used EAs: Genetic Algorithm and Differential Evolution with DE outperforming GA on most tasks [32]

ELM uses a MAP-elites Quality Diversity algorithm. [34]

Initial population consists of manually-written prompts to leverage human knowledge as well as some prompts generated by LLMs to reflect the fact that EAs start from random solutions to avoid local optima. [32]

DE-inspired approached builds on the idea that the common elements of the current best prompts need to be preserved [32]

Evoprompt performs best with roulette selection when compared with tournament and random selection. [32]

Similar results are achieved when population is initialized with the best and with random prompts, hinting that the crafted design of initial prompts is not essential. [32]

Previous research optimized zero-shot instructions and examples separately, overlooking their interplay and resulting in sub-optimal performance. [40]

There is a prevailing notion that prompt engineering sacrifices efficiency for performance due to the lengthening of prompts, but PhaseEvo actively shortens the prompts [40]

Current EA applications to prompt optimization suffer from extremely high computational cost and slow convergence speed due to the complexity of the high-dimensional search space. [40]

PhaseEvo alternates between two phases: exploration with evolution operators and exploitation using a feedback "gradient". [40]

TABLE 1 recreate compares all 5 operators. [40]

4 phases: initialization - lamarck or manual, local feedback mutation, global evolution with EDA and CR operators, local semantic mutation (paraphrasing) [40]

Candidates for evolution operators are selected based on a "performance vector", combining prompts that do not make the same mistakes. [40]

When the performance improvement with an operator stagnates up to some operator-specific tolerance, the current phase is terminated. [40]

Evolution in phases outperforms random operator selection. [40]

PhaseEvo is the most cost-effective but still needs around 12 iterations and 4000 API calls. [40]

APE [33] ran into problems with diminishing returns and abandoning the iterative approach entirely, Promptbreeder aims to solve this with a diversity-maintaining evolutionary algorithm for self-referential self-improvement of prompts [41]

Prompt optimization techniques utilize the fact that LLMs are effective at generating mutations from examples and can encode human notions of interestingness and can be used to quantify novelty. [41]

Self-referential system should improve the way it is improving, thus Promptbreeder used a "hyper-prompt" to optimize its meta-prompt [41]

Uses a binary tournament genetic algorithm. [41]

Uses a random uniformly sampled mutation operators out of 9 total from 5 broad categories for each replication event. [41]

Zero-order mutation (creating a prompt from task description) generates new task prompts more aligned with the task description in the event the evolution diverges. [41]

LLMs tend to be biased to examples found later in EDA mutation lists. Lying to the LLM and telling it that the prompts are sorted by performance in a descending order improves diversity. [41]

Removing any self-referential operator in ablation is harmful under nearly all circumstances [41]

Prior works do not provide sufficient guidance in the meta-prompt. [29]

PE2 improves on APE and APO with a back-tracking search procedure and a more complete metaprompt with a two-step task description, prompt layout specification and a step-by-step reflection template. [29]

Experiments with step size and momentum and also a prompt engineering tutorial in the metaprompt to mixed results. [29]

Optimized prompts do not seem to be generalizable across different models. [29]

Gold-agnostic methods

Current prompt optimization methods often depend heavily on external references for evaluation which are often unavailable or unpractical to define in many applications, especially for open-ended tasks [30] Gold-agnostic evaluation is important because we ultimately expect LLMs to solve problems for which answers are not already known [18]

Two primary sources can be used for evaluation: LLM-generated outputs and task-specific truths. These can then be evaluated using either a predefined

metric, LLM-as-a-judge or by a human judge to produce an optimization signal based on a numeric score or a textual feedback. [30]

In each iteration, SPO generates new prompts, executes them, and performs pair-wise evaluations of outputs to assess their adherence [30]

SPO achieves high efficiency, requiring only 8 LLM calls per iteration with three samples, significantly lower than existing methods [30]

Outputs of LLMs inherently contain rich quality information that directly reflects prompt effectiveness [30]

LLMs exhibit human-like task comprehension [30]

With score-based feedback a large sample size is needed to ensure scoring stability, which can be avoided by pairwise comparison of outputs [30]

LLM-as-a-judge biases do not affect the overall optimization trend because eval's feedback merely serves as a reference for the next round of optimization [30]

While maintaining comparable performance with other ground truth-dependent prompt optimization methods, SPO requires only 1.1% to 5.6% of their optimization costs [30]

Self-consistency can by used as a metric instead of accuracy as correct answers generally exhibit greater self-consistency than incorrect ones. However it can overestimate prompts that produce consistent incorrect answers. [18]

Mutual-consistency refinement refines self-consistency scores based on the self-consistency scores of other prompts [18]

Gold-agnostic evaluation methods like GLaPE are robust metrics akin to accuracy and are able to produce effective prompts similarly to gold-label-based methods like OPRO[14]. [18]

If all prompts produce consistent but incorrect answers it is challenging to discern the error without external resources. This happens in some datasets, leading to diminished correlation between GLaPE and accuracy. [18]

Metaprompting

Metaprompting or "prompting to create prompts"

Meta-prompts are task-agnostic, meaning they will return the relevant outputs for an arbitrary task, provided a task description is provided as an input. [42].

It is possible to construct a general-purpose meta-prompt. [42].

Meta-prompts will perform better than standard prompts at executing a wide range of tasks. [42].

In tests meta-generated prompts were ranked as more useful than the baselines as well as producing content that was rated more suitable. [42].

Chapter 3

Experiments

3.1 Datasets

Datasets were chosen according to the following requirements:

1. The task is challenging for modern LLMs using a standard CoT prompt but has non-zero accuracy
2. Complex output (no multiple-choice answers)
3. Easy to check programmatically to avoid human/AI judges

3.1.1 External dataset

We found the Livebench datasets to meet our requirements.

3.1.2 Custom dataset design

Sequences dataset challenges the pattern recognition and algebraic capabilities of the model

3.2 Optimization methods

3.2.1 Optimization operators

Metaprompts that define the transition between optimizer generations.

3.3 Experimental setup

Language model used for solving is gpt-4o-mini. Prompts for the solver model are optimized by the optimized model, for which we use the gpt-4o. To encourage diversity and exploration in the optimization process, a temperature of 0.7 is used for the optimizer model. The solver model uses temperature 0.0 to keep the outputs deterministic.

3.4 Comparative analysis of optimization operators

Chapter 4

Conclusion

Appendix A

Bibliography

[1] OpenAI, :, A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, A. Iftimie, A. Karpenko, A. T. Passos, A. Neitz, A. Prokofiev, A. Wei, A. Tam, A. Bennett, A. Kumar, A. Saraiva, A. Vallone, A. Duberstein, A. Kondrich, A. Mishchenko, A. Applebaum, A. Jiang, A. Nair, B. Zoph, B. Ghorbani, B. Rossen, B. Sokolowsky, B. Barak, B. McGrew, B. Minaiev, B. Hao, B. Baker, B. Houghton, B. McKinzie, B. Eastman, C. Lugaresi, C. Bassin, C. Hudson, C. M. Li, C. de Bourcy, C. Voss, C. Shen, C. Zhang, C. Koch, C. Orsinger, C. Hesse, C. Fischer, C. Chan, D. Roberts, D. Kappler, D. Levy, D. Selsam, D. Dohan, D. Farhi, D. Mely, D. Robinson, D. Tsipras, D. Li, D. Oprica, E. Freeman, E. Zhang, E. Wong, E. Proehl, E. Cheung, E. Mitchell, E. Wallace, E. Ritter, E. Mays, F. Wang, F. P. Such, F. Raso, F. Leoni, F. Tsimpourlas, F. Song, F. von Lohmann, F. Sulit, G. Salmon, G. Parascandolo, G. Chabot, G. Zhao, G. Brockman, G. Leclerc, H. Salman, H. Bao, H. Sheng, H. Andrin, H. Bagherinezhad, H. Ren, H. Lightman, H. W. Chung, I. Kivlichan, I. O'Connell, I. Osband, I. C. Gilaberte, I. Akkaya, I. Kostrikov, I. Sutskever, I. Kofman, J. Pachocki, J. Lennon, J. Wei, J. Harb, J. Twore, J. Feng, J. Yu, J. Weng, J. Tang, J. Yu, J. Q. Candela, J. Palermo, J. Parish, J. Heidecke, J. Hallman, J. Rizzo, J. Gordon, J. Uesato, J. Ward, J. Huizinga, J. Wang, K. Chen, K. Xiao, K. Singhal, K. Nguyen, K. Cobbe, K. Shi, K. Wood, K. Rimbach, K. Gu-Lemberg, K. Liu, K. Lu, K. Stone, K. Yu, L. Ahmad, L. Yang, L. Liu, L. Maksin, L. Ho, L. Fedus, L. Weng, L. Li, L. McCallum, L. Held, L. Kuhn, L. Kondraciuk, L. Kaiser, L. Metz, M. Boyd, M. Trebacz, M. Joglekar, M. Chen, M. Tintor, M. Meyer, M. Jones, M. Kaufer, M. Schwarzer, M. Shah, M. Yatbaz, M. Y. Guan, M. Xu,

M. Yan, M. Glaese, M. Chen, M. Lampe, M. Malek, M. Wang, M. Fradin, M. McClay, M. Pavlov, M. Wang, M. Wang, M. Murati, M. Bavarian, M. Rohaninejad, N. McAleese, N. Chowdhury, N. Chowdhury, N. Ryder, N. Tezak, N. Brown, O. Nachum, O. Boiko, O. Murk, O. Watkins, P. Chao, P. Ashbourne, P. Izmailov, P. Zhokhov, R. Dias, R. Arora, R. Lin, R. G. Lopes, R. Gaon, R. Miyara, R. Leike, R. Hwang, R. Garg, R. Brown, R. James, R. Shu, R. Cheu, R. Greene, S. Jain, S. Altman, S. Toizer, S. Toyer, S. Miserendino, S. Agarwal, S. Hernandez, S. Baker, S. McKinney, S. Yan, S. Zhao, S. Hu, S. Santurkar, S. R. Chaudhuri, S. Zhang, S. Fu, S. Papay, S. Lin, S. Balaji, S. Sanjeev, S. Sidor, T. Broda, A. Clark, T. Wang, T. Gordon, T. Sanders, T. Patwardhan, T. Sottiaux, T. Degry, T. Dimson, T. Zheng, T. Garipov, T. Stasi, T. Bansal, T. Creech, T. Peterson, T. Eloundou, V. Qi, V. Kosaraju, V. Monaco, V. Pong, V. Fomenko, W. Zheng, W. Zhou, W. McCabe, W. Zaremba, Y. Dubois, Y. Lu, Y. Chen, Y. Cha, Y. Bai, Y. He, Y. Zhang, Y. Wang, Z. Shao, and Z. Li, "Openai o1 system card," 2024.

[2] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025.

[3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[4] S. Welleck, A. Bertsch, M. Finlayson, H. Schoelkopf, A. Xie, G. Neubig, I. Kulikov, and Z. Harchaoui, "From decoding to meta-generation: Inference-time algorithms for large language models," 2024.

[5] X. Wang and D. Zhou, "Chain-of-thought reasoning without prompting," 2024.

[6] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," 2023.

[7] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," 2023.

[8] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, "Dspy: Compiling declarative language model calls into self-improving pipelines," 2023.

[9] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), vol. 35, pp. 22199–22213, Curran Associates, Inc., 2022.

[10] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[11] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023.

[12] B. Prystawski, M. Y. Li, and N. D. Goodman, "Why think step by step? reasoning emerges from the locality of experience," 2023.

[13] B. Brown, J. Juravsky, R. Ehrlich, R. Clark, Q. V. Le, C. Ré, and A. Mirhoseini, "Large language monkeys: Scaling inference compute with repeated sampling," 2024.

[14] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," 2024.

[15] Z. Zeng, Q. Cheng, Z. Yin, Y. Zhou, and X. Qiu, "Revisiting the test-time scaling of o1-like models: Do they truly possess test-time scaling capabilities?," 2025.

[16] R. Liu, J. Geng, A. J. Wu, I. Sucholutsky, T. Lombrozo, and T. L. Griffiths, "Mind your step (by step): Chain-of-thought can reduce performance on tasks where thinking makes humans worse," 2024.

[17] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2023.

[18] X. Zhang, Z. Zhang, and H. Zhao, "Glape: Gold label-agnostic prompt evaluation and optimization for large language model," 2024.

[19] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, and T. Hoefler, "Graph of thoughts: Solving elaborate problems with large language models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, p. 17682–17690, Mar. 2024.

[20] K. Misaki, Y. Inoue, Y. Imajuku, S. Kuroki, T. Nakamura, and T. Akiba, "Wider or deeper? scaling llm inference-time compute with adaptive branching tree search," 2025.

[21] F. Teng, Z. Yu, Q. Shi, J. Zhang, C. Wu, and Y. Luo, "Atom of thoughts for markov llm test-time scaling," 2025.

[22] Z. Wang, J. Zeng, O. Delalleau, D. Egert, E. Evans, H.-C. Shin, F. Soares, Y. Dong, and O. Kuchaiev, "Dedicated feedback and edit models empower inference-time scaling for open-ended general-domain tasks," 2025.

[23] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," 2023.

[24] T. Schnabel and J. Neville, "Symbolic prompt program search: A structure-aware approach to efficient compile-time prompt optimization," 2024.

[25] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff, P. S. Dulepet, S. Vidyadhara, D. Ki, S. Agrawal, C. Pham, G. Kroiz, F. Li, H. Tao, A. Srivastava, H. D. Costa, S. Gupta, M. L. Rogers, I. Goncearenco, G. Sarli, I. Galynker, D. Peskoff, M. Carpuat, J. White, S. Anadkat, A. Hoyle, and P. Resnik, "The prompt report: A systematic survey of prompting techniques," 2024.

[26] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[27] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray,

B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020.

[28] E. Meyerson, M. J. Nelson, H. Bradley, A. Gaier, A. Moradi, A. K. Hoover, and J. Lehman, “Language model crossover: Variation through few-shot prompting,” 2024.

[29] Q. Ye, M. Axmed, R. Pryzant, and F. Khani, “Prompt engineering a prompt engineer,” 2024.

[30] J. Xiang, J. Zhang, Z. Yu, F. Teng, J. Tu, X. Liang, S. Hong, C. Wu, and Y. Luo, “Self-supervised prompt optimization,” 2025.

[31] M. Deng, J. Wang, C.-P. Hsieh, Y. Wang, H. Guo, T. Shu, M. Song, E. P. Xing, and Z. Hu, “Rlprompt: Optimizing discrete text prompts with reinforcement learning,” 2022.

[32] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, “Connecting large language models with evolutionary algorithms yields powerful prompt optimizers,” 2024.

[33] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, “Large language models are human-level prompt engineers,” 2023.

[34] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, “Evolution through large models,” 2022.

[35] X. Tang, X. Wang, W. X. Zhao, S. Lu, Y. Li, and J.-R. Wen, “Unleashing the potential of large language models as prompt optimizers: An analogical analysis with gradient-based model optimizers,” 2024.

[36] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng, “Automatic prompt optimization with "gradient descent" and beam search,” 2023.

[37] H. He, Q. Liu, L. Xu, C. Shivade, Y. Zhang, S. Srinivasan, and K. Kirchhoff, “Crispo: Multi-aspect critique-suggestion-guided automatic prompt optimization for text generation,” 2024.

[38] K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab, “Optimizing instructions and demonstrations for multi-stage language model programs,” 2024.

[39] D. Soylu, C. Potts, and O. Khattab, “Fine-tuning and prompt optimization: Two great steps that work better together,” 2024.

[40] W. Cui, J. Zhang, Z. Li, H. Sun, D. Lopez, K. Das, B. Malin, and S. Kumar, “Phaseevo: Towards unified in-context prompt optimization for large language models,” 2024.

[41] C. Fernando, D. Banarse, H. Michalewski, S. Osindero, and T. Rocktäschel, “Promptbreeder: Self-referential self-improvement via prompt evolution,” 2023.

[42] A. de Wynter, X. Wang, Q. Gu, and S.-Q. Chen, "On meta-prompting," 2024.