tnGPS: Discovering Unknown Tensor Network Structure Search Algorithms via Large Language Models (LLMs)

Junhua Zeng [1*] **Chao Li** [2*] **Zhun Sun** [3] **Qibin Zhao** [2] **Guoxu Zhou** [1 4]

Abstract

Tensor networks are efficient for extremely high-dimensional representation, but their model selection, known as tensor network structure search (TN-SS), is a challenging problem. Although several works have targeted TN-SS, most existing algorithms are manually crafted heuristics with poor performance, suffering from the curse of dimensionality and local convergence. In this work, we jump out of the box, studying how to *harness large language models (LLMs) to automatically discover new TN-SS algorithms, replacing the involvement of human experts.* By observing how human experts innovate in research, we model their common workflow and propose an automatic algorithm discovery framework called tnGPS. The proposed framework is an elaborate prompting pipeline that instruct LLMs to generate new TN-SS algorithms through iterative refinement and enhancement. The experimental results demonstrate that the algorithms discovered by tnGPS exhibit superior performance in benchmarks compared to the current state-of-the-art methods. Our code is available at `https://github.com/ChaoLiAtRIKEN/tngps`.

1. Introduction

Tensor networks (TNs) are powerful methods that have proven to be highly beneficial in machine learning and various other fields (Markov & Shi, 2008; Anandkumar et al., 2014; Orús, 2014; Novikov et al., 2015; Cichocki et al., 2016; Stoudenmire & Schwab, 2016; Cichocki et al., 2017; Orús, 2019; Glasser et al., 2019; Kossaifi et al., 2020; Miller et al., 2021; Richter et al., 2021; Miller et al., 2021; Haghshenas et al., 2022). Their effectiveness lies in the ability to decompose extremely high-dimensional problems into low-dimensional factors. However, *tensor network structure search (TN-SS)*—the task of selecting the optimal model-related hyperparameters such as TN ranks (Ye & Lim, 2019) and topology (Li & Sun, 2020)—can be a challenging problem due to its high-dimensional discrete nature and computational difficulty (Hillar & Lim, 2013).

Until now, various approaches have been developed to solve TN-SS, employing different methods (Hashemizadeh et al., 2020; Li & Sun, 2020; Nie et al., 2021; Chen et al., 2022; Li et al., 2022; 2023; Zheng et al., 2023; Zeng et al., 2024). Among these, sampling-based algorithms (Hashemizadeh et al., 2020; Li & Sun, 2020; Li et al., 2022; 2023) have demonstrated superior performance in addressing TN-SS for machine learning tasks. The core idea of these approaches is to sequentially draw samples in the search space according to their heuristic strategies, such as TNGA (*evolutionary algorithm*, Li & Sun, 2020), GREEDY (*greedy algorithm*, Hashemizadeh et al., 2020), and TNLS and TnALE (*local-search algorithms*, Li et al., 2022; 2023), until optimal TN structures are reached. Additionally, these sampling-based approaches are more compatible with a variety of loss functions, model architectures, and optimizers in machine/deep learning.

However, achieving a good balance between exploration and exploitation in sampling-based heuristic approaches is notoriously difficult and seemingly endless. This issue arises because different downstream tasks require different balance points, and various heuristic strategies have distinct preferences between exploration and exploitation. For example, TNGA excels in exploration but performs poorly in exploitation, suffering from the curse of dimensionality. In contrast, TNLS and TnALE are more efficient than TNGA as they leverage the local smoothness prior of the search space. However, as noted by Li et al. (2023), they are more prone to getting stuck in local minima, particularly with real-world data. This issue not only leads to poor performance of approaches but also results in a labor-intensive *"research cycle"*, where human experts need to repeatedly develop

[*]Equal contribution [1]School of Automation, Guangdong University of Technology, Guangzhou, China [2]RIKEN Center for Advanced Intelligence Project (RIKEN-AIP), Tokyo, Japan [3]Tencent Inc., Shenzhen, China [4]Key Laboratory of Intelligent Detection and the Internet of Things in Manufacturing, Ministry of Education, Guangzhou, China. Correspondence to: Guoxu Zhou <gx.zhou@gdut.edu.cn>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

new TN-SS algorithms to deal with different downstream tasks.

Is there a (semi-)automatic way to advance this cycle, reducing the need for intensive labor efforts? If such a method exists, it would free human experts to tackle more challenging problems. More importantly, it would help us address the ultimate question: "How far can this cycle take us?"

To answer the questions, we propose a large language model (LLM)-driven automation framework, designed to automatically discover unknown TN-SS algorithms. Leveraging the remarkable full-domain knowledge of LLMs and their emergent understanding and reasoning capabilities, the proposed framework, termed tensor-network-purposed GPT-driven structure search (tnGPS), takes the existing TN-SS algorithms (coded in Python) as inputs, and then mimic human experts' workflow for innovative research, to generate a batch of novel TN-SS algorithms expected to surpass the state-of-the-art (SOTA) algorithms. It is worth noting that in tnGPS LLMs play a crucial role in "innovation": we construct a pipeline of prompts through which LLMs explore novel TN-SS algorithms from various perspectives, such as knowledge categorization (`KC`), knowledge recombination (`KR`), incremental innovation (`II`) and diversity injection (`DI`). The generated new TN-SS algorithms are then evaluated in local computers with extensive downstream-task-specific numerical experiments. The experimental results are subsequently used to update the prompts, guiding the LLMs to improve the algorithms in the next iteration.

We numerically evaluate the effectiveness of the discovered TN-SS algorithms by tnGPS on benchmarks, comparing them to the existing TN-SS algorithms. The experiment results demonstrate that tnGPS can discover novel TN-SS algorithms that not only outperform existing algorithms on in-domain data but also exhibit superior performance on out-of-domain data. In addition, we also implement ablation experiments to analyze the contributions of various components of the tnGPS framework. The main contributions of this paper are summarized as two-fold:

- We propose tnGPS, a large language model (LLM)-driven automation framework designed to automatically generate novel and effective TN-SS algorithms tailored to specific downstream tasks;
- Experimental results demonstrate that the algorithms discovered by tnGPS outperform existing TN-SS algorithms on benchmark data.

1.1. Related Works

Tensor network structure search (TN-SS). The problem of TN-SS can be viewed as an extension of rank selection for tensor decomposition (Rai et al., 2014; Zhao et al., 2015; Yokota et al., 2016), which can be further traced back to studies of matrix factorization (Babacan et al., 2012). Unlike rank selection, TN-SS aim to search for a richer set of model-related hyperparameters of a tensor network, including not only TN-ranks (Cheng et al., 2020; Mickelin & Karaman, 2020; Li et al., 2021; Kodryan et al., 2023) but also network topology (Hayashi et al., 2019; Hashemizadeh et al., 2020; Li & Sun, 2020; Haberstich et al., 2023) and permutation (Chen et al., 2022). From the technique perspective, various methods have been employed including Bayesian inference (Zeng et al., 2024), spectrum methods (Chen et al., 2022), continuous optimization (Zheng et al., 2023) and discrete optimization (Li et al., 2023), etc.. In this work, we follow the technical route of discrete optimization, particularly the sampling-based technique, due to its high precision and flexibility with a variety of loss functions. Unlike the existing methods that solve TN-SS, the goal of this work is to develop a "*meta-method*" that can create novel and effective TN-SS algorithms through the automatic discovery of algorithms.

Automatic algorithm discovery (AAD). The studies on AAD can be traced back to the early 1900s (GRATCH, 1992; Minton, 1993). Building on this foundational works, more recent studies (KhudaBukhsh et al., 2016; Meng & Qu, 2021; Yi et al., 2022) developed various frameworks based on evolutionary programming and machine learning to discover efficient algorithms for solving computationally hard problems such as complex combinatorial optimization. Parallel to these efforts, similar problems have been addressed in the field of AutoML (He et al., 2021). The difference is that the works (Bello et al., 2017; Real et al., 2020; Wang et al., 2022; Chen et al., 2023) in AutoML dedicate efforts to neural architecture search and optimizer design.

Large language models revolutionize AAD. LLMs have demonstrated remarkable performance in code programming (Haluptzok et al., 2023; Liventsev et al., 2023; Min et al., 2024; Hong et al., 2024; Hemberg et al., 2024; Gur et al., 2024). Since numerical algorithms are typically implemented on computers through programming languages like Python or C++, the capability of LLMs for code programming can be naturally extended to AAD (Zelikman et al., 2023; Liu et al., 2023; Pluhacek et al., 2023; Liu et al., 2024; Ye et al., 2024; Romera-Paredes et al., 2024). Note that conventional AAD methods discover new algorithms from a handcrafted, finite-dimensional algorithm space, while LLMs can explore new algorithms from an infinite-dimensional code space, which encompasses knowledge drawn from extensive training data across multiple fields. The most closely related methods to ours are the works (Liu et al., 2023; 2024) that utilize LLMs to evolve algorithms for solving the well-known traveling salesman problem. Unlike their works, we target the TN-SS problem in this paper, establishing an AAD framework that leverages LLMs to mimic human experts.

2. Preliminaries

In this section, we first review the problem of tensor network structure search for self-constrained purposes. Following this, we introduce a workflow that models how human experts typically innovate in research. This workflow will then be used in the next section to guide the design of the proposed framework.

2.1. Tensor Network Structure Search (TN-SS)

To provide intuitive simplicity, we review TN-SS through its application to tensor decomposition. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ be a non-zero tensor of order N. In this work, we consider solving TN-SS by minimizing the following objective function (Li & Sun, 2020):

$$F(A) := \phi(A) + \lambda \cdot \min_{\{\mathcal{Z}_n\}_{n=1}^N} \underbrace{\|\mathcal{X} - tns(\{\mathcal{Z}_n\}; A)\|_F^2 / \|\mathcal{X}\|_F^2}_{\text{relative squared error (RSE)}}. \quad (1)$$

Here, $A \in \mathbb{A} \subseteq \mathbb{R}^{N \times N}$ represents the adjacency matrix that formulates the graphical diagram of a tensor network, *i.e.*, the tensor network (TN) structure. The feasible set $\mathbb{A}$ is determined by specific searching domains, such as TN-ranks, permutations (Li et al., 2022) and topology (Li & Sun, 2020). The set $\{\mathcal{Z}_n\}_{n=1}^N$ represents the collection of core tensors (Cichocki et al., 2016) of a tensor network $tns(\{\mathcal{Z}_n\}; A)$ associated to the TN-structure A (Ye & Lim, 2019). The objective function (1) describes a linear combination of two terms with a tuning parameter $\lambda > 0$, where the first term $\phi(A)$ models a TN's complexity (such as compression ratio and graph sparsity), and the second term calculates the minimization of relative squared error (RSE) in tensor decomposition, modeling the expressibility of a TN. In summary, within the context of tensor decomposition, TN-SS aims to find the optimal TN structures, modeled with A, that minimizes both the TN's complexity and the RSE.

2.2. A Unified Paradigm for TN-SS Algorithms

In this work, we consider minimizing the objective function (1) as a discrete optimization problem. Most existing algorithms that solve (1) follow a "*sampling-evaluation*" paradigm (Li & Sun, 2020; Li et al., 2022; 2023). Algo. 1 provides its basic description, in which the key ingredient is the sampling operation as follows:

$$p \leftarrow \mathit{GenerateSamples}(C, P, F, i, m, \#Iter, L), \quad (2)$$

where p denotes a collection of adjacency matrices defined in (1), the arrow $\leftarrow$ represents the value assignment, and $C, P, F, i, m, \#Iter, L$ denotes arguments required in the function. They contain important information such as the current optimal TN structures C, historical sampled structures and their corresponding evaluation scores (P, F), as described in Algo. 1. The operation (2) implies that in

Algorithm 1 The "Sampling-Evaluation" Paradigm

```
1:  Initialize:
2:  $m$                     ▷ Number of samples
3:  $L$                     ▷ Set of hyperparameters
4:  $\#Iter$                ▷ Maximum number of iterations
5:  $P \leftarrow []$       ▷ Historical TN structures, initially empty
6:  $F \leftarrow []$       ▷ Historical evaluation scores, initially empty
7:  $C \leftarrow []$       ▷ The best TN structures, initially empty
8:
9:  for $i = 1$ to $\#Iter$ do
10:     $p \leftarrow GenerateSamples(C, P, F, i, m, \#Iter, L)$
11:     $F \leftarrow F \cup Eval(p)$     ▷ Evaluate new samples
12:     $P \leftarrow P \cup p$           ▷ Update historical samples
13:     Update $C$ if necessary           ▷ Update the best structure
14:
15: Output: Set $C$ containing the best TN structures.
```

each iteration a batch of new adjacency matrices are generated, conditioned on the historical adjacency matrices and their corresponding evaluation scores in the searching process. Given adjacency matrices, the evaluation scores are calculated in the operation $Eval(p)$, by minimizing the objective (1) with respect to $\{\mathcal{Z}_n\}_{n=1}^N$.

Under the paradigm illustrated in Algo. 1, the existing TN-SS algorithms differ mainly from the customization on the *GenerateSamples*$(\cdot)$ in (2). For example, TNGA (Li & Sun, 2020) specifies (2) with evolutionary operators; GREEDY (Hashemizadeh et al., 2020), TNLS (Li et al., 2022) and TnALE (Li et al., 2023) leverage incremental, random or alternating sampling in neighborhood to generate new samples. In this work, we aim to leverage LLMs to discover new ideas for designing the function *GenerateSamples*$(\cdot)$, with the goal of improving the performance of solving TN-SS.

2.3. How Human Experts do Innovation?

The intuition of this work is to harness LLMs to mimic human experts to realize the goal of discovering novel TN-SS algorithms. In doing so, we formalize below a basic but complete workflow, shown in Figure 1, to describe how human experts do innovative research.

In academic research, human experts typically begin with preliminary studies, such as gathering information through literature reviews and paper retrieval. The gathered information is then compiled into an *idea pool*, which includes numerous existing ideas related to the targeted problem, such as solving TN-SS. Each idea is accompanied by various descriptors, including an evaluation score and contextual information that describes the idea's background, intuition, and other relevant details. Subsequently, in the *knowledge categorization* (KC) phase, the ideas are refined into *knowledge* clusters. Each cluster consists of different ideas that

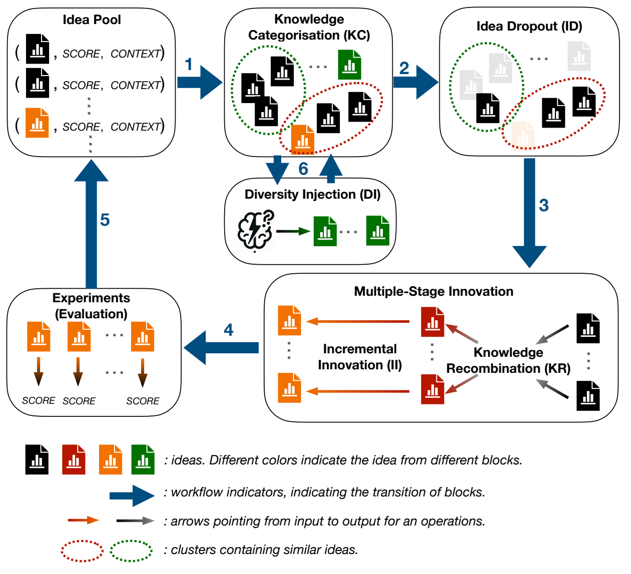

Idea Pool
(, SCORE, CONTEXT)
(, SCORE, CONTEXT)
(, SCORE, CONTEXT)
1
Knowledge Categorisation (KC)
2
Idea Dropout (ID)
3
6
Diversity Injection (DI)
5
Multiple-Stage Innovation
Incremental Innovation (II)
Knowledge Recombination (KR)
4
Experiments (Evaluation)
SCORE SCORE SCORE
: ideas. Different colors indicate the idea from different blocks.
: workflow indicators, indicating the transition of blocks.
: arrows pointing from input to output for an operations.
: clusters containing similar ideas.

Figure 1. A basic workflow to illustrate how human experts do innovation in research.

follow a similar hypothesis or principle. In *Idea Dropout* (ID), this phase models the behavior of human experts by filtering out certain ideas from the pool, allowing them to focus only on the most interesting ones for deeper study. The metrics used in ID are typically multivariate, encompassing factors such as personal interests, performance, trends, or even randomness due to the large scale of the idea pool.

In this work, innovation is considered as an operation that generates new ideas from existing ones. As shown in Figure 1, we model innovation as a two-stage process consisting of *knowledge recombination* (KR) and *incremental innovatio* (II) These phases are key ingredients to create values in not only research but also in strategy and entrepreneurship (Rubin & Abramson, 2018; Xiao et al., 2022). More specifically speaking, KR refers to the process of generating new ideas by merging, integrating, or reconfiguring existing ideas. II involves gradual improvements or minor modifications to existing ideas. Additionally, Figure 1 formalizes *diversity injection* (DI), which is forced to generate new ideas that are "orthogonal" to the existing ones. DI is commonly observed in real-world scenarios, such as brainstorming sessions in team meetings or critical comments from non-experts. Significant innovation is ultimately expected to emerge by recursively invoking KR, II, and DI within the workflow.

3. tnGPS: a LLM-Driven Framework for TN-SS Algorithm discovery

In this section, we introduce tnGPS, an LLM-driven framework for discovering TN-SS algorithms. The introduction primarily focuses on the technical aspects, illustrating how tnGPS is designed with a pipeline of prompts to harness LLMs in order to discover novel TN-SS algorithms, mimicking the innovation process of human experts.

Global architecture of tnGPS. We conceptualize tnGPS as a system where the inputs are known TN-SS algorithms encoded in Python, and the outputs are newly discovered TN-SS algorithms. Inspired by the way human experts conduct innovative research, tnGPS is designed with a global pipeline similar to the one depicted in Figure 1. In this design, the LLM acts as an agent to perform the functions of each phase shown in Figure 1, replacing human experts. Below, we introduce the specific implementation of each block in detail.

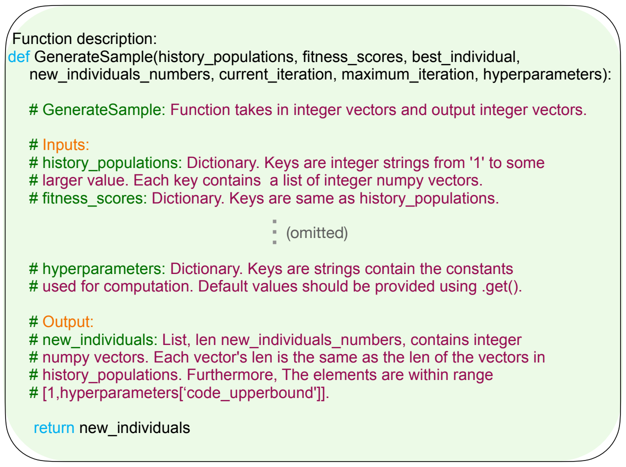
Idea Pool, also referred to as *"the pool"* for brevity throughout the paper, is defined as a set of algorithms described as (algorithm, score). Here, algorithm contains various implementations of the function *GenerateSamples*$(\cdot)$ as defined in Eq. (2) of a TN-SS algorithm using Python, and score is a scalar value indicating the algorithm's performance in the TN-SS problem. To help LLMs understand TN-SS algorithms effectively, all algorithms in the pool are standardized with a unified interface, as depicted in Figure 2. Additionally, the text in Figure 2 will serve as the "pilot" for constructing the prompt, as discussed at the end of this section.

Knowledge categorisation (KC). In this phase, we create clusters of algorithms using LLMs. Each cluster contains similar algorithms, representing a distinct piece of knowledge on how to solve TN-SS. In doing so, we first simplify and initialize the clusters with each algorithm in the pool. Once a new algorithm (e.g. one generated by tnGPS) is coming, we prompt the LLM to evaluate the methodological similarity between the new algorithm and the cluster centroids, which are the algorithms with the best scores in each cluster. The new algorithm is then assigned an index to declare its cluster membership. The key prompt used in this phase is illustrated in Figure 3.

Idea dropout (ID). The dropout is conducted by randomly selecting algorithms from the pool using a roulette selection mechanism. Consider N algorithms, indexed from 1 to N based on their scores, ranked from highest to lowest. Inspired by the previous work (Li & Sun, 2020), the roulette selection performs a sequential sampling without replacement The probability of selecting the kth algorithm is given by

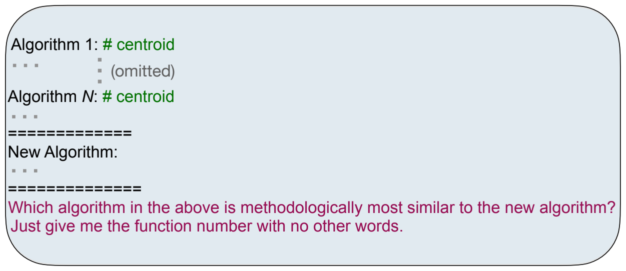
$$Pr(k) = \max\left\{0.01, \ln\left(\frac{\alpha}{eps + k}\right)\right\}, \tag{3}$$

where eps is a very small positive number to ensure numerical stability, $\ln(\cdot)$ denotes the natural logarithm function, and $\alpha > 0$ is a hyperparameter that controls the selection preference. A smaller value of α increases the likelihood of selecting algorithms with higher scores.



```
Function description:
def GenerateSample(history_populations, fitness_scores, best_individual,
    new_individuals_numbers, current_iteration, maximum_iteration, hyperparameters):

    # GenerateSample: Function takes in integer vectors and output integer vectors.

    # Inputs:
    # history_populations: Dictionary. Keys are integer strings from '1' to some
    # larger value. Each key contains  a list of integer numpy vectors.
    # fitness_scores: Dictionary. Keys are same as history_populations.

                                  ⋮ (omitted)

    # hyperparameters: Dictionary. Keys are strings contain the constants
    # used for computation. Default values should be provided using .get().

    # Output:
    # new_individuals: List, len new_individuals_numbers, contains integer
    # numpy vectors. Each vector's len is the same as the len of the vectors in
    # history_populations. Furthermore, The elements are within range
    # [1,hyperparameters['code_upperbound']].

     return new_individuals
```

Figure 2. The prompt used for interface description.



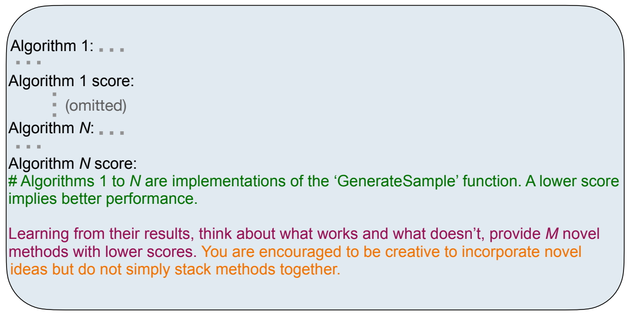
```
Algorithm 1: # centroid
⋯          ⋮ (omitted)
Algorithm N: # centroid
⋯
=============
New Algorithm:
⋯
==============
Which algorithm in the above is methodologically most similar to the new algorithm?
Just give me the function number with no other words.
```

Figure 3. The key prompt used in knowledge categorisation (KC). The *purple* sentence specifies the goal of the prompt.

To select a preferred algorithm in ID, we need to implement the roulette selection twice. First, we perform selection at the knowledge level. This involves selecting the centroids of clusters to determine which clusters will be considered. Second, within the selected clusters, we perform selection at the algorithm level to choose the preferred algorithm from each cluster. This "bi-level" process is repeated until the desired number of algorithms has been selected.

Knowledge recombination (KR). Guided by the workflow shown in Figure 1, KR is a fundamental phase to generate novel TN-SS algorithms in tnGPS. In this phase, the input consists of $N \geq 0$ pairs of (algorithm, score). The output is M new algorithms generated by the LLM. The key prompt used in KR is illustrated in Figure 4. The design of this prompt aims to enable the LLM to understand the factors contributing to the superior performance of certain algorithms while avoiding the meaningless stacking of Python code.

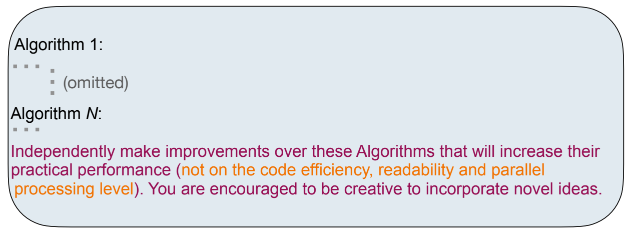
Incremental innovation (II) follows KR as another block for creating innovation in tnGPS. Unlike KR which recombines algorithms, II aims at mortifying algorithms individually and slightly. Figure 5 gives the key prompt used in this phase. In our experience with GPT-3.5/4, we found



```
Algorithm 1: ⋯
⋯
Algorithm 1 score:
        ⋮ (omitted)
Algorithm N: ⋯
⋯
Algorithm N score:
# Algorithms 1 to N are implementations of the 'GenerateSample' function. A lower score
implies better performance.

Learning from their results, think about what works and what doesn't, provide M novel
methods with lower scores. You are encouraged to be creative to incorporate novel
ideas but do not simply stack methods together.
```

Figure 4. The key prompt used in knowledge recombination (KR). The *purple* sentence specifies the goal and the *orange* sentence clarifies the restriction.

that the LLM tends to improve code aspects such as efficiency, readability, and parallelism, which are not the focus of this phase. To prevent this, we include specific instructions (highlighted in orange in Figure 5) to guide the LLM away from these unintended improvements and focus on the targeted modifications.

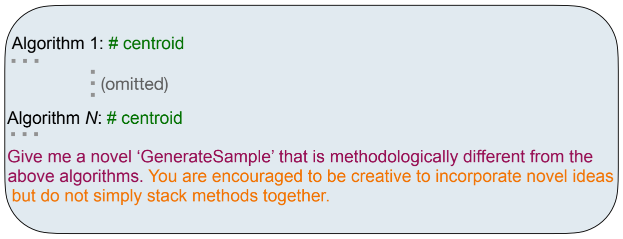


```
Algorithm 1:
⋯   ⋮ (omitted)
Algorithm N:
⋯
Independently make improvements over these Algorithms that will increase their
practical performance (not on the code efficiency, readability and parallel
processing level). You are encouraged to be creative to incorporate novel ideas.
```

Figure 5. The key prompt used in incremental innovation (II). The *purple* sentence specifies the goal and the *orange* sentence clarifies the restriction.

Diversity injection (DI). We implement DI by leveraging the LLM to create new clusters in the algorithm pool. In doing so, we design the prompt as Figure 6, instructing the LLM to generate TN-SS algorithms that are methodologically distinct from the centroids of existing clusters. Following this, the newly created algorithms from DI serve as centroids for new clusters. The newly created algorithms from DI serve as centroids for new clusters. Since relying solely on existing knowledge can lead to "path dependence"—where new ideas are heavily influenced by past knowledge—we encourage the LLM to explore new ideas in DI without considering performance metrics (e.g., scores). This approach enriches the diversity of the algorithm pool.

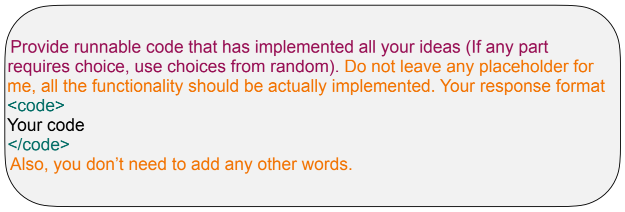
Format restriction and prompt architecture. In addition to the goal-oriented prompts mentioned earlier, we need to enforce specific format restrictions on the LLM's output. For instance, this includes omitting unnecessary analyses and specifying the desired format for code outputs. Furthermore, instructions related to code writing must be included to ensure that there are no compilation failures during evalu-



Algorithm 1: # centroid
...
(omitted)
Algorithm *N*: # centroid
...
Give me a novel 'GenerateSample' that is methodologically different from the above algorithms. You are encouraged to be creative to incorporate novel ideas but do not simply stack methods together.

Figure 6. The key prompt used in diversity injection (DI). The *purple* sentence specifies the goal and the *orange* sentence clarifies the restriction.

ation. To meet these requirements, we designed the prompt shown in Figure 7. It's important to note that the prompt in Figure 7 was developed through a trial-and-error process, making it dependent on the specific LLM used and our code-writing conventions. As a result, manual adjustments may be necessary when using different LLM models.



Provide runnable code that has implemented all your ideas (If any part requires choice, use choices from random). Do not leave any placeholder for me, all the functionality should be actually implemented. Your response format
<code>
Your code
</code>
Also, you don't need to add any other words.

Figure 7. The prompt used for format restriction of the output. The *purple* sentence specifies the goal and the *orange* sentence clarifies the restriction.

Finally, the complete prompts used in KC, KR, II and DI are constructed following the same architecture as shown in Figure 8. It consists of three parts: interface description (Figure 2), those goal-oriented prompts following a list of algorithms and scores (e.g., Figures 3, 4, 5, 6), and format restriction (Figure 7). These three parts are concatenated together to provide comprehensive instructions to the LLM.

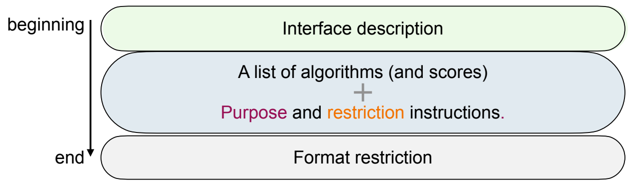

beginning
Interface description
A list of algorithms (and scores)
+
Purpose and restriction instructions.
end
Format restriction

Figure 8. Illustration to prompt architecture.

Experiments (evaluation) with sandbox. All newly generated TN-SS algorithms are evaluated on local (super)computers using task-specific training data. However, due to the probabilistic nature of LLMs, there is a possibility of receiving unexpected outputs, such as unrunable code, unnecessary comments, or error messages from the LLM platforms. To address this issue, we construct a sandbox environment to run the code in a relatively isolated setting using small-scale training data before conducting high-cost formal numerical experiments. If program errors, abnormal resource consumption, or unexpected output formats occur, the sandbox will immediately terminate the process and remove the problematic code from the job queue awaiting implementation.

4. Experimental Results

In this section, we use several benchmarks to demonstrate that tnGPS can discover new algorithms that outperform SOTA methods for TN-SS. Additionally, we conduct ablation experiments to evaluate the impact of each component within tnGPS on the discovery performance.

4.1. Natural Images Compression

In this experiment, we aim to use TN-SS algorithms to search for optimal topology and ranks for a tensor network (TN) to represent natural images with fewer parameters.

Data preparation. We randomly select 14 images from the BSD500 dataset (Arbelaez et al., 2010). These images are converted to grayscale and resized to 256×256 pixels. Each image is then reshaped into an order-8 tensor by the default Python reshaping function. The 14 pre-processed images are split into two sets: 4 images for training and 10 images for testing.

Settings of tnGPS. We use three existing TN-SS algorithms as inputs: TNGA (Li & Sun, 2020), GREEDY (Hashemizadeh et al., 2020) and TNLS (Li et al., 2022). The evaluation phase in tnGPS calculates Eq. (1) for each generated algorithm, averaging the results over the images in the training set. In Eq. (1), we set $\lambda = 5$ and use the same compression ratio function ϕ as in previous work (Li & Sun, 2020). The hyperparameters required in tnGPS are listed in Table 1. For this experiment, we set $m = 2$, $n = 1$, $\alpha_1 = \alpha_2 = 100$, $c = 5$ and $\#Iter = 30$. Additionally, we select gpt-4-1106-preview as the LLM model, applying a temperature of 0.7 uniformly across all prompts. After implementation, we select the top-three algorithms from the pool (excluding the input algorithms), termed Ho-1[1], Ho-2, and Ho-3, as the outputs of tnGPS. The three algorithms will be evaluated and compared with the existing TN-SS algorithms.

Implementation details. In the experiment, we implement four additional sampling-based TN-SS algorithms including TNGA (Li & Sun, 2020), TNLS (Li et al., 2022), GREEDY (Hashemizadeh et al., 2020) and TnALE (Li et al., 2023). Since the vanilla TNGA and TNLS are designed to

[1]The name "Ho" is shorthand for *"homunculus"*, representing that these algorithms are created through some "unusual" means.

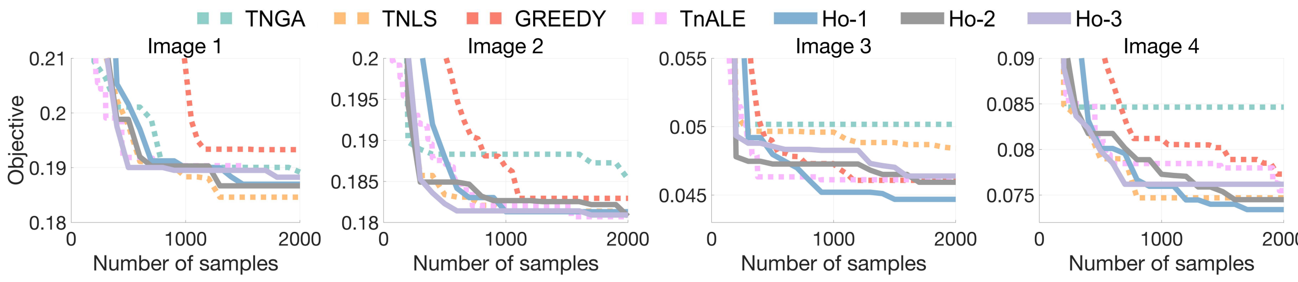


Figure 9. Objective $vs.$ number of sample curves of different algorithms on four training images.

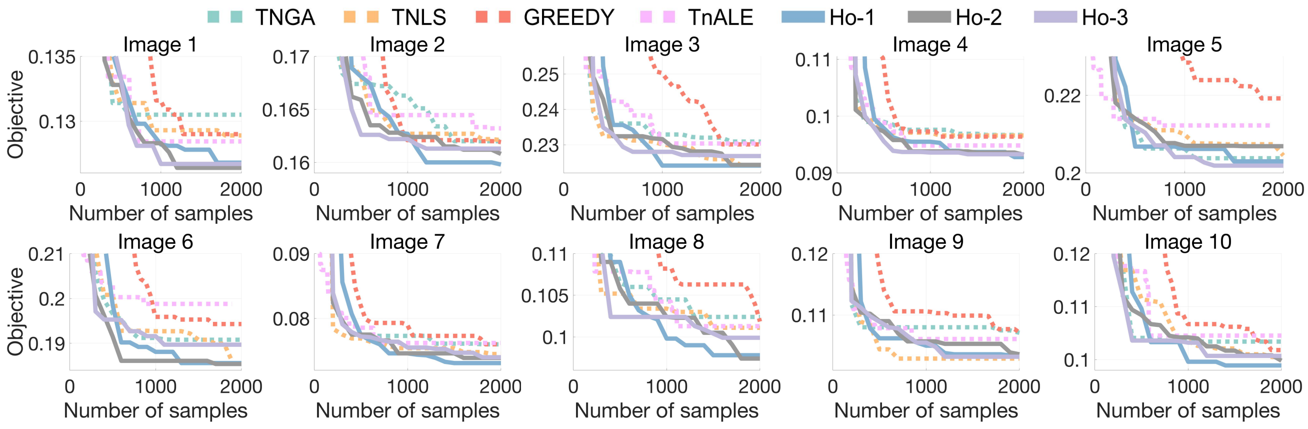


Figure 10. Objective $vs.$ number of sample curves of different algorithms on ten testing images.

Table 1. Parameters involved in tnGPS.

Parameter	**Description**
$\#Iter$	Maximum iteration
α_1	Parameter in Eq. 3 for cluster selection
α_2	Parameter in Eq. 3 for algorithm selection
m	Number of selected algorithms in `ID`
n	Number of generated algorithms
c	Maximum number of clusters

search only for the topology or permutation of a TN, we extend them to fit the settings of this experiment. Specifically, we extend TNGA by relaxing its binary constraint for encoding the topology with integers as done in the previous works (Li et al., 2022; 2023). For TNLS, we fix the permutation and set the *template* used in the algorithm to be a complete graph, enabling simultaneous search for TN topology and ranks. In GREEDY, we modify its objective function from RSE to the function in (1), and further allow the algorithm to both increase and decrease the ranks during the search.

The parameter settings of the algorithms are as follows: in TNGA, we set $\alpha = 100$, $\beta = 5$, the elimination rate to 10%, and the mutation probability to 25%; in TNLS, we set $c_1 = 0.99$; and in TnALE, we set $L_0 = 0, L = 15, r_2 = 1$, and $D = 1$. For all methods (including those generated by tnGPS), we set the upper limit for rank search to 4, the number of iterations in searching to 20, and the number of samples[2] in each iteration to 100. For TNGA and the algorithms generated by tnGPS, we initialize them with same TN structures, which have TN ranks close to one but with a 15% probability of changing each rank from 1 to 2. We then select the best TN structure from the TNGA initialization to initialize GREEDY, TNLS, and TnALE.

Results. Figures 9 and 10 show how the value of the objective function (3) changes with increasing the number of samples for different TN-SS algorithms. Table 2 shows the averaged performance metrics, including compression ratio and RSE, of different algorithms on both the training and test sets. As shown in Table 2, the three algorithms generated by tnGPS achieve comparable performance on the training set, while 'Ho-2" outperforms other algorithms on average in the test set. Figure 10 visually demonstrates that the curves associated with the three algorithms generated by tnGPS tend to reach smaller values of the objective function compared to other methods.

New insights gained from the generated algorithms. The codes of Ho-1,2,3 are provided in Appendix B. These codes reveal several new insights on how to improve the effectiveness and efficiency of solving TN-SS problems. First, the

[2]In TnALE, the number of samples in each iteration is determined by other hyperparameters.

Table 2. The averaged compression ratio (in log form, calculated by dividing the number of parameters of the image by the number of parameters of the TN) and the RSE for the structures obtained by the algorithms.

Images	**Log compression ratio↑ + RSE↓ – *CR(RSE)***						
	TNGA	**TNLS**	**GREEDY**	**TnALE**	**Ho-1**	**Ho-2**	**Ho-3**
Train	1.478 (0.128)	1.419 (0.119)	1.451 (0.123)	1.439 (0.122)	1.436 (0.120)	1.496 (0.124)	1.446 (0.121)
Test	1.332 (0.132)	1.335 (0.131)	1.331 (0.132)	1.328 (0.132)	1.329 (0.129)	**1.352 (0.130)**	1.322 (0.129)

Table 3. Number of parameters ($\times$1000) for TGP model compression. The values in [square brackets] give the number of samples used to find the structure for the first time. The symbol "-" means the algorithm fails to achieve the experiment's configuration.

	Initialization	**Baseline**	**TNGA**	**TNLS**	**GREEDY**	**TnALE**	**Ho-1**	**Ho-2**	**Ho-3**
CCPP	`random`	2.64	2.36 [1900]	2.50 [1900]	2.60 [850]	2.36 [588]	2.60 [1900]	**2.24** [1600]	2.74 [1300]
MG	`random`	3.36	12.69 [8400]	17.25 [7600]	-	-	6.81 [9200]	**3.01** [10000]	27.74 [8400]
CCPP	`TT`	2.64	2.24 [500]	2.24 [200]	2.24 [21]	2.24 [**18**]	2.24 [400]	2.24 [500]	2.24 [300]
MG	`TT`	3.36	3.36 [100]	3.01 [500]	3.01 [64]	3.01 [**42**]	3.01 [500]	3.01 [1200]	3.01 [2400]

Table 4. Value of objective function (1) of the best algorithm discovered by tnGPS and its variants. In the table, "baseline" refers to the best result obtained from TNGA, TNLS, GREEDY, and TnALE, "tnGPS" refers to the proposed model, and "`KR, II, DI`" are tnGPS's variants, whose corresponding components are ablated.

	baseline	**tnGPS**	`KR`	`II`	`DI`
Objective	0.1558	**0.1102**	0.1308	0.1273	0.1239

Table 5. Value of objective function (1) of the best algorithm discovered by tnGPS and its variants. In the table, "baseline" refers to the best result obtained from TNGA, TNLS, GREEDY, and TnALE, "GPT-4, GPT-3.5, Claude-1, Claude-2" refers to tnGPS using various LLMs, and "Incomplete descriptions" is the variant of tnGPS, whose interface description (Figure 2) is partially removed.

baseline	**GPT-4**	**GPT-3.5**	**Claude-1**	**Claude-2**	**Incomplete descriptions**
0.1847	**0.1813**	0.1842	0.1840	0.1834	0.1819

new algorithms are no longer "Markovian". Unlike previous methods that considered only the samples from the last iteration, the new algorithms incorporate information from all historical samples. Second, the annealing trick is used inversely. For instance, in Ho-1, the algorithm employs a mutation operation similar to TNGA, with the mutation rate updated as in TNLS. However, contrary to the traditional annealing trick used in TNLS, the mutation rate in Ho-1 increases progressively to enhance exploration. Third, the new algorithms adopt various novel exploitation strategies inspired by TNLS. Notably, Ho-2 introduces an innovative Gaussian perturbation mutation strategy, which, to the best of our knowledge, is unprecedented in the existing genetic algorithms literature.

4.2. Model Compression for Gaussian Process: an Out-of-Domain Experiment

In this experiment, we evaluate whether the algorithms generated by tnGPS maintain their effectiveness for the out-of-domain tasks, *i.e.*, tasks not considered in the algorithm discovery process. In doing so, we follow previous TN-SS works (Li et al., 2022; 2023) and consider the compression of a regression model using tensorial Gaussian process (TGP) (Izmailov et al., 2018).

Experiment Setup. We apply TN-SS to the high-order variational mean tensor of TGP to compress the model while preserving the prediction accuracy. Two datasets are used: CCPP (Tüfekci, 2014) and MG (Flake & Lawrence, 2002). In these datasets, the corresponding variational means are tensors of order 4 with a mode dimension of 12, and order 6 with a mode dimension of 8, respectively. For the experiment, the upper bound for rank search is set to 10. The Adam (Kingma & Ba, 2014) learning rate is set to 0.001, and the core tensors of TN are initialized with a Gaussian distribution with a variance of 0.01. Due to the distinct scales of the two datasets, the hyperparameter λ in Eq. (1) is set to 10^5 for CCPP and 10^7 for MG, respectively.

We consider two types of initialization in the experiment: `random` and `TT`. In `random`, the initial TN structures are selected randomly following the same method used in the preceding experiment. In `TT`, we first initialize TN structures using the method in `random`, but one of the initialized structures is then replaced by tensor train (Oseledets, 2011), which is the model used in the baseline (Izmailov et al., 2018). If only one structure requires initialization, such as in TNLS and TnALE, the baseline tensor train is used directly. In the settings of MG + `random`, we set the number of iterations in searching to 50, and the number of samples in each iteration to 200. In other settings, we set the number of iterations in searching to 30, and the number of samples in each iteration to 100.

Results. The experimental results are presented in Table 3, which shows the number of parameters (in thousands) required by tensor networks to represent the model. Additionally, the number of samples used by the algorithms to find the structure for the first time is indicated in square brackets. We can see that, in the `TT` initialization, all algorithms find better structures than the baseline, with GREEDY and TnALE requiring fewer samples than the other algorithms. However, in the `random` initialization, only Ho-2 finds structures as good as those in `TT`. Although GREEDY and TnALE still converge quickly, the solutions they find are worse than Ho-2's. Notably, in the setting of MG+`random`, only Ho-2 finds a structure better than the baseline. These results suggest that tnGPS can generate new TN-SS algorithms that consistently outperform the existing SOTA methods in both in-domain and out-of-domain tasks.

4.3. Ablations

Next, we conduct ablation studies to evaluate the impact of various components of tnGPS on its performance. These components include those LLM-driven phases in the pipeline, the interface description, and the selection of LLM models. By systematically removing or modifying these components, we aim to understand their individual contributions to the overall effectiveness of tnGPS.

LLM-driven phases in tnGPS. Here, we individually ablate the phases including knowledge recombination (`KR`), incremental innovation (`II`), and diversity injection (`DI`). We use the variational mean tensor from the MG dataset in the model compression experiment as the training data for tnGPS to assess the impact of each phase on the overall performance.

In this part, the hyper-parameter settings for the tnGPS are the same as in the image compression experiment. The other TN-SS algorithms maintain the same settings as in the model compression experiment, except that we set the number of samples per iteration to 50 and run the algorithms for 10 iterations for simplicity.

We present the experimental results for the tnGPS components in Table 4. As observed, the complete tnGPS model achieves the best results, surpassing existing methods. However, the model's performance declines when components like `KR`, `II` and `DI` are removed, underscoring the importance of integrating, enhancing, and injecting ideas. Additionally, the ablation results for `KR`, `II` and `DI` still surpass existing methods, suggesting the potential benefits of these individual components in discovering better algorithms.

Selection of LLM models and interface description. We next examine how the selection of LLM models affects tnGPS's performance. We compare GPT-4 (`gpt-4-1106-preview`), GPT-3.5 (`gpt-3.5-turbo-16k-0613`), Claude-1 (`claude-instant-1`), and Claude-2 (`claude-2`). Additionally, we analyze tnGPS's sensitivity to interface description (shown in Figure 2) by randomly removing 8 from the total 12 comments (see Figure 11 in Appendix). For simplicity, we select image 2 from the training set in the image compression experiment as the training data and use the same hyper-parameter settings as in the image compression experiment, except for adjusting the iteration and sample settings for the TN-SS algorithm as done in the LLM-driven phases ablation experiment.

As concluded from the results in Table 5, more powerful LLMs like GPT-4 lead to better tnGPS performance. Moreover, by further analyzing the algorithms generated by GPT-3.5, we found that it tends to make trivial modifications of algorithms, such as the naive stacking of different methods. Additionally, the incomplete interface description does not significantly affect the performance of tnGPS. We conjecture that the algorithms themselves provide sufficient information to let LLMs understand the function to be generated. However, during the experiment, we observed that the LLM sometimes misunderstood the meaning of the objective function values, for example, mistakenly thinking that larger values implied better performance.

Concluding Remarks

Our experiential results confirm the positive impact of LLMs on solving TN-SS. Specifically, the proposed framework, tnGPS, can leverage insights gained from the existing algorithms and the embedded knowledge in LLMs to automatically discover novel TN-SS algorithms that achieve a better balance between exploration and exploitation. The benchmarks of image compression and model compression consistently demonstrate the superiority of the discovered algorithms in finding better TN structures.

Limitation. A primary limitation of our method is that the final discovered algorithms' performance is subject to variation due to changes in LLMs. Nonetheless, we anticipate a positive correlation between the performance of designed algorithms by our method and the capabilities of LLMs.

Impact Statement

This paper explores the potential of using LLMs to automatically discover improved algorithms without human intervention, a step that could significantly boost the efficiency of algorithm development and augment production processes. However, it also raises concerns about LLMs generating unreliable results due to the hallucination issue, which affects their reliability and practicality. To address this, our study emphasizes the significance of not only evaluating the outputs produced by LLMs but also constructing precise prompts to guide them effectively. These approaches help minimize the risk of unreliable results, ensuring the practicality and reliability of LLMs for practitioners. Moreover, the increasing capabilities of LLMs may also lead to the question of whether machines may eventually replace human expertise entirely.

Acknowledgements

We appreciate the anonymous (meta-)reviewers for their helpful comments and are grateful to Yumeng Ma for her invaluable advice in formulating the method. Junhua completed this work during an internship at RIKEN-AIP under the supervision of Chao and Qibin. The work was partially supported by National Natural Science Foundation of China (Grant No. 62073087 and 62071132), and JSPS Kakenhi (Grant No. 24K03005 and JP23K28109). Additionally, Chao was also supported by RIKEN Incentive Research Project.

References

Anandkumar, A., Ge, R., Hsu, D., Kakade, S. M., and Telgarsky, M. Tensor decompositions for learning latent variable models. *The Journal of Machine Learning Research*, 15(1):2773–2832, 2014.

Arbelaez, P., Maire, M., Fowlkes, C., and Malik, J. Contour detection and hierarchical image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 33(5):898–916, 2010.

Babacan, S. D., Luessi, M., Molina, R., and Katsaggelos, A. K. Sparse bayesian methods for low-rank matrix estimation. *IEEE Transactions on Signal Processing*, 60(8): 3964–3977, 2012. doi: 10.1109/TSP.2012.2197748.

Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pp. 459–468. PMLR, 2017.

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C.-J., Lu, Y., et al. Symbolic discovery of optimization algorithms. *Advances in Neural Information Processing Systems*, 36, 2023.

Chen, Z., Lu, J., and Zhang, A. R. One-dimensional tensor network recovery. *arXiv preprint arXiv:2207.10665*, 2022.

Cheng, Z., Li, B., Fan, Y., and Bao, Y. A novel rank selection scheme in tensor ring decomposition based on reinforcement learning for deep neural networks. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3292–3296. IEEE, 2020.

Cichocki, A., Lee, N., Oseledets, I., Phan, A.-H., Zhao, Q., Mandic, D. P., et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9(4-5):249–429, 2016.

Cichocki, A., Phan, A.-H., Zhao, Q., Lee, N., Oseledets, I., Sugiyama, M., Mandic, D. P., et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. *Foundations and Trends® in Machine Learning*, 9(6):431–673, 2017.

Flake, G. W. and Lawrence, S. Efficient SVM regression training with SMO. *Machine Learning*, 46(1):271–290, 2002.

Glasser, I., Sweke, R., Pancotti, N., Eisert, J., and Cirac, I. Expressive power of tensor-network factorizations for probabilistic modeling. *Advances in neural information processing systems*, 32, 2019.

GRATCH, J. Composer: A probabilistic solution to the utility problem in speed-up learning. In *AAAI-92*, pp. 235–240, 1992.

Gur, I., Furuta, H., Huang, A. V., Safdari, M., Matsuo, Y., Eck, D., and Faust, A. A real-world webagent with planning, long context understanding, and program synthesis. In *The Twelfth International Conference on Learning Representations*, 2024.

Haberstich, C., Nouy, A., and Perrin, G. Active learning of tree tensor networks using optimal least squares. *SIAM/ASA Journal on Uncertainty Quantification*, 11(3): 848–876, 2023.

Haghshenas, R., Gray, J., Potter, A. C., and Chan, G. K.-L. Variational power of quantum circuit tensor networks. *Physical Review X*, 12(1):011047, 2022.

Haluptzok, P., Bowers, M., and Kalai, A. T. Language models can teach themselves to program better. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=SaRj2ka1XZ3.

Hashemizadeh, M., Liu, M., Miller, J., and Rabusseau, G. Adaptive learning of tensor network structures. *arXiv preprint arXiv:2008.05437*, 2020.

Hayashi, K., Yamaguchi, T., Sugawara, Y., and Maeda, S.-i. Exploring unexplored tensor network decompositions for convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 5553–5563, 2019.

He, X., Zhao, K., and Chu, X. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622, 2021.

Hemberg, E., Moskal, S., and O'Reilly, U.-M. Evolving code with a large language model. *arXiv preprint arXiv:2401.07102*, 2024.

Hillar, C. J. and Lim, L.-H. Most tensor problems are NP-hard. *Journal of the ACM (JACM)*, 60(6):45, 2013.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Izmailov, P., Novikov, A., and Kropotov, D. Scalable Gaussian processes with billions of inducing inputs via tensor train decomposition. In *International Conference on Artificial Intelligence and Statistics*, pp. 726–735. PMLR, 2018.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., and Leyton-Brown, K. Satenstein: Automatically building local search sat solvers from components. *Artificial Intelligence*, 232:20–42, 2016.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kodryan, M., Kropotov, D., and Vetrov, D. Mars: Masked automatic ranks selection in tensor decompositions. In *International Conference on Artificial Intelligence and Statistics*, pp. 3718–3732. PMLR, 2023.

Kossaifi, J., Lipton, Z. C., Kolbeinsson, A., Khanna, A., Furlanello, T., and Anandkumar, A. Tensor regression networks. *Journal of Machine Learning Research*, 21: 1–21, 2020.

Li, C. and Sun, Z. Evolutionary topology search for tensor network decomposition. In *International Conference on Machine Learning*, pp. 5947–5957. PMLR, 2020.

Li, C., Zeng, J., Tao, Z., and Zhao, Q. Permutation search of tensor network structures via local sampling. In *International Conference on Machine Learning*, pp. 13106–13124. PMLR, 2022.

Li, C., Zeng, J., Li, C., Caiafa, C. F., and Zhao, Q. Alternating local enumeration (tnale): Solving tensor network structure search with fewer evaluations. In *International Conference on Machine Learning*, pp. 20384–20411. PMLR, 2023.

Li, N., Pan, Y., Chen, Y., Ding, Z., Zhao, D., and Xu, Z. Heuristic rank selection with progressively searching tensor ring network. *Complex & Intelligent Systems*, pp. 1–15, 2021.

Liu, F., Tong, X., Yuan, M., and Zhang, Q. Algorithm evolution using large language model. *arXiv preprint arXiv:2311.15249*, 2023.

Liu, F., Tong, X., Yuan, M., Lin, X., Luo, F., Wang, Z., Lu, Z., and Zhang, Q. An example of evolutionary computation+ large language model beating human: Design of efficient guided local search. *arXiv preprint arXiv:2401.02051*, 2024.

Liventsev, V., Grishina, A., Härmä, A., and Moonen, L. Fully autonomous programming with large language models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1146–1155, 2023.

Markov, I. L. and Shi, Y. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

Meng, W. and Qu, R. Automated design of search algorithms: Learning on algorithmic components. *Expert Systems with Applications*, 185:115493, 2021.

Mickelin, O. and Karaman, S. On algorithms for and computing with the tensor ring decomposition. *Numerical Linear Algebra with Applications*, 27(3):e2289, 2020.

Miller, J., Rabusseau, G., and Terilla, J. Tensor networks for probabilistic sequence modeling. In *International Conference on Artificial Intelligence and Statistics*, pp. 3079–3087. PMLR, 2021.

Min, M. J., Ding, Y., Buratti, L., Pujar, S., Kaiser, G., Jana, S., and Ray, B. Beyond accuracy: Evaluating self-consistency of code large language models with identitychain. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=caW7LdAALh.

Minton, S. An analytic learning system for specializing heuristics. In *IJCAI*, volume 93, pp. 922–929. Citeseer, 1993.

Nie, C., Wang, H., and Tian, L. Adaptive tensor networks decomposition. In *BMVC*, 2021.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pp. 442–450, 2015.

Orús, R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

Orús, R. Tensor networks for complex quantum systems. *Nature Reviews Physics*, 1(9):538–550, 2019.

Oseledets, I. V. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

Pluhacek, M., Kazikova, A., Kadavy, T., Viktorin, A., and Senkerik, R. Leveraging large language models for the generation of novel metaheuristic optimization algorithms. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, pp. 1812–1820, 2023.

Rai, P., Wang, Y., Guo, S., Chen, G., Dunson, D., and Carin, L. Scalable bayesian low-rank decomposition of incomplete multiway tensors. In *International Conference on Machine Learning*, pp. 1800–1808. PMLR, 2014.

Real, E., Liang, C., So, D., and Le, Q. Automl-zero: Evolving machine learning algorithms from scratch. In *International conference on machine learning*, pp. 8007–8019. PMLR, 2020.

Richter, L., Sallandt, L., and Nüsken, N. Solving high-dimensional parabolic pdes using the tensor train format. In *International Conference on Machine Learning*, pp. 8998–9009. PMLR, 2021.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Rubin, G. D. and Abramson, R. G. Creating value through incremental innovation: Managing culture, structure, and process. *Radiology*, 288(2):330–340, 2018.

Stoudenmire, E. and Schwab, D. J. Supervised learning with tensor networks. In *Advances in Neural Information Processing Systems*, pp. 4799–4807, 2016.

Tüfekci, P. Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods. *International Journal of Electrical Power & Energy Systems*, 60:126–140, 2014.

Wang, R., Xiong, Y., Cheng, M., and Hsieh, C.-J. Efficient non-parametric optimizer search for diverse tasks. *Advances in Neural Information Processing Systems*, 35: 30554–30568, 2022.

Xiao, T., Makhija, M., and Karim, S. A knowledge recombination perspective of innovation: review and new research directions. *Journal of Management*, 48(6):1724–1777, 2022.

Ye, H., Wang, J., Cao, Z., and Song, G. Reevo: Large language models as hyper-heuristics with reflective evolution. *arXiv preprint arXiv:2402.01145*, 2024.

Ye, K. and Lim, L.-H. Tensor network ranks. *arXiv preprint arXiv:1801.02662*, 2019.

Yi, W., Qu, R., Jiao, L., and Niu, B. Automated design of metaheuristics using reinforcement learning within a novel general search framework. *IEEE Transactions on Evolutionary Computation*, 2022.

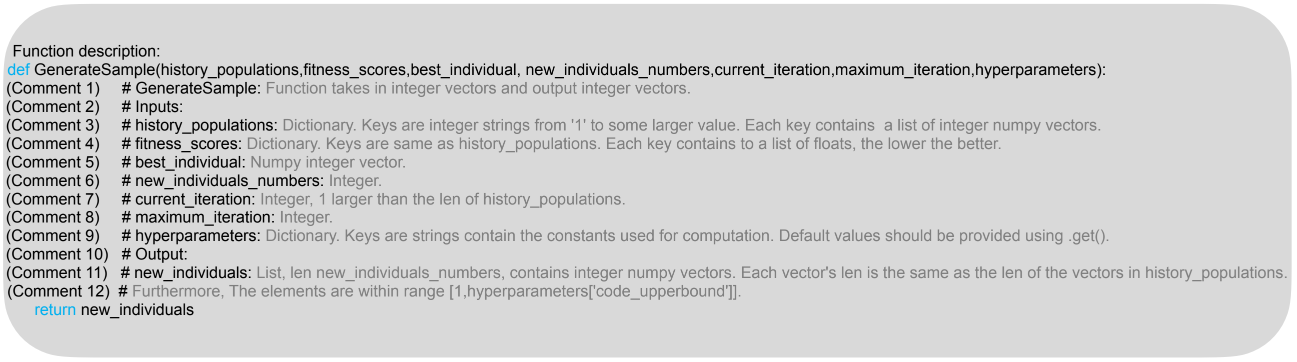
Yokota, T., Zhao, Q., and Cichocki, A. Smooth PARAFAC decomposition for tensor completion. *IEEE Transactions on Signal Processing*, 64(20):5423–5436, 2016.

Zelikman, E., Lorch, E., Mackey, L., and Kalai, A. T. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023.

Zeng, J., Zhou, G., Qiu, Y., Li, C., and Zhao, Q. Bayesian tensor network structure search and its application to tensor completion. *Neural Networks*, pp. 106290, 2024.

Zhao, Q., Zhang, L., and Cichocki, A. Bayesian CP factorization of incomplete tensors with automatic rank determination. *IEEE transactions on pattern analysis and machine intelligence*, 37(9):1751–1763, 2015.

Zheng, Y.-B., Zhao, X.-L., Zeng, J., Li, C., Zhao, Q., Li, H.-C., and Huang, T.-Z. Svdinstn: An integrated method for tensor network representation with efficient structure search. *arXiv preprint arXiv:2305.14912*, 2023.



```
Function description:
def GenerateSample(history_populations,fitness_scores,best_individual, new_individuals_numbers,current_iteration,maximum_iteration,hyperparameters):
(Comment 1)    # GenerateSample: Function takes in integer vectors and output integer vectors.
(Comment 2)    # Inputs:
(Comment 3)    # history_populations: Dictionary. Keys are integer strings from '1' to some larger value. Each key contains  a list of integer numpy vectors.
(Comment 4)    # fitness_scores: Dictionary. Keys are same as history_populations. Each key contains to a list of floats, the lower the better.
(Comment 5)    # best_individual: Numpy integer vector.
(Comment 6)    # new_individuals_numbers: Integer.
(Comment 7)    # current_iteration: Integer, 1 larger than the len of history_populations.
(Comment 8)    # maximum_iteration: Integer.
(Comment 9)    # hyperparameters: Dictionary. Keys are strings contain the constants used for computation. Default values should be provided using .get().
(Comment 10)  # Output:
(Comment 11)  # new_individuals: List, len new_individuals_numbers, contains integer numpy vectors. Each vector's len is the same as the len of the vectors in history_populations.
(Comment 12)  # Furthermore, The elements are within range [1,hyperparameters['code_upperbound']].
     return new_individuals
```

Figure 11. Illustration of the interface description. In the interface description ablation experiment, comments $\{1, 2, 3, 4, 6, 7, 8, 11\}$ are removed.

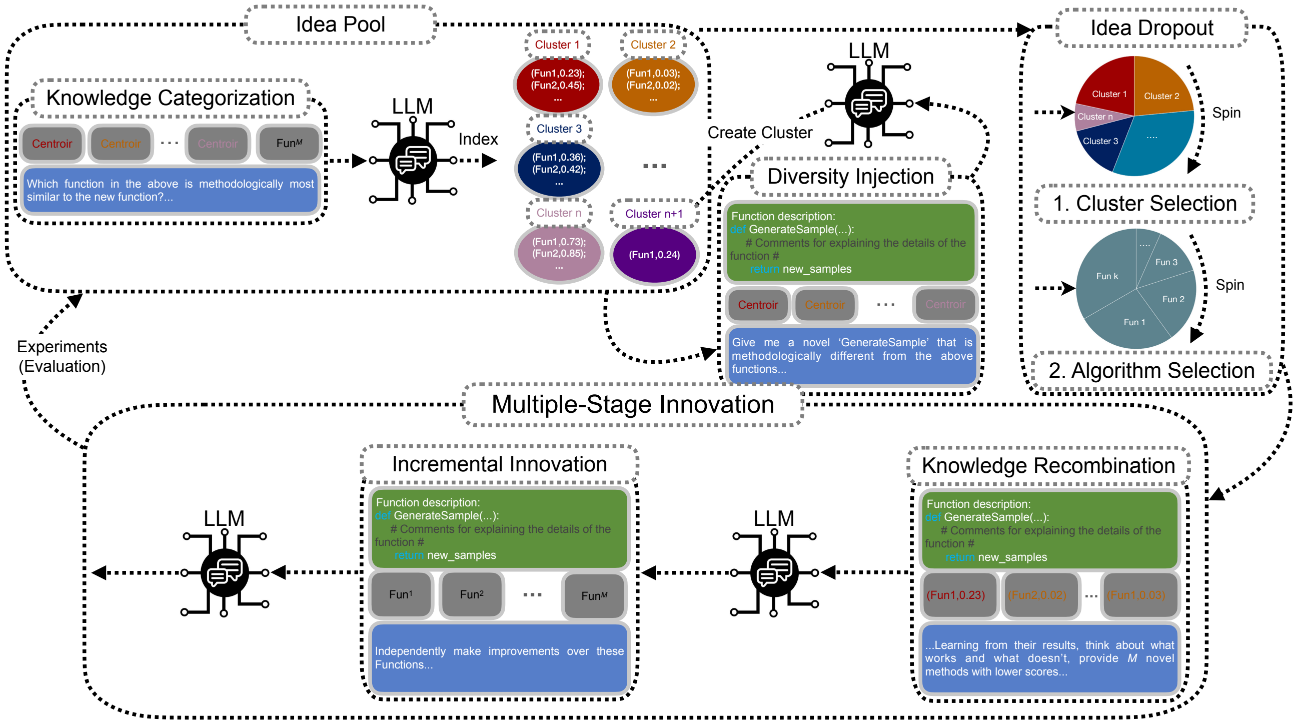

Idea Pool
Knowledge Categorization
Centroir
Centroir
Centroir
Fun^M
Which function in the above is methodologically most similar to the new function?...
LLM
Index
Cluster 1
(Fun1,0.23); (Fun2,0.45); ...
Cluster 2
(Fun1,0.03); (Fun2,0.02); ...
Cluster 3
(Fun1,0.36); (Fun2,0.42); ...
Cluster n
(Fun1,0.73); (Fun2,0.85); ...
Cluster n+1
(Fun1,0.24)
LLM
Create Cluster
Diversity Injection
Function description:
def GenerateSample(...):
# Comments for explaining the details of the function #
return new_samples
Centroir
Centroir
Centroir
Give me a novel 'GenerateSample' that is methodologically different from the above functions...
Idea Dropout
Cluster 1
Cluster 2
Cluster n
Cluster 3
Spin
1. Cluster Selection
Fun k
Fun 3
Fun 2
Fun 1
Spin
2. Algorithm Selection
Experiments (Evaluation)
Multiple-Stage Innovation
Incremental Innovation
Function description:
def GenerateSample(...):
# Comments for explaining the details of the function #
return new_samples
Fun^1
Fun^2
Fun^M
Independently make improvements over these Functions...
LLM
LLM
Knowledge Recombination
Function description:
def GenerateSample(...):
# Comments for explaining the details of the function #
return new_samples
(Fun1,0.23)
(Fun2,0.02)
(Fun1,0.03)
...Learning from their results, think about what works and what doesn't, provide M novel methods with lower scores...

Figure 12. The workflow of tnGPS. Each prompt can be constructed using up to three components: the interface description, the in-context algorithms, and the meta-prompt. The interface description is used to provide a precise objective, ensuring that the codes generated by the LLM are correct. The in-context algorithms provide detailed information on the existing algorithms for LLM. Finally, the meta-prompt is designed to guide the LLM. The other colors in the figure are utilized to indicate different clusters of algorithms.

A. The workflow of tnGPS

In Figure 12, we demonstrate the detailed workflow of tnGPS.

B. Codes for different TN-SS algorithms

In the following, we present the codes for Ho-1, Ho-2, and Ho-3, as well as the codes of the algorithms discovered by tnGPS in the ablation experiments, and the codes for TNGA, TNLS, and GREEDY.

B.1. Code for Ho-1

```
def GenerateSample(history_populations,fitness_scores,best_individual,new_individuals_numbers,current_iteration,
↪   maximum_iteration,hyperparameters):
    # Define default hyperparameters using .get()
    hyperparams = {
        'code_upperbound': hyperparameters.get('code_upperbound', 10),
        'mutation_rate': hyperparameters.get('mutation_rate', 0.15),
        'crossover_rate': hyperparameters.get('crossover_rate', 0.7),
        'selection_pressure': hyperparameters.get('selection_pressure', 1.8),
        'elitism_count': hyperparameters.get('elitism_count', 1),
        'mutation_scaling_factor': hyperparameters.get('mutation_scaling_factor', 0.9),
        'max_mutation': hyperparameters.get('max_mutation', 2),
        'tournament_size_factor': hyperparameters.get('tournament_size_factor', 0.15)
    }

    # Nested functions under GenerateSample
    def mutate(individual, scaling_factor):
        mutation_count = max(1, int(len(individual) * scaling_factor * hyperparams['mutation_rate']))
        mutation_indices = random.sample(range(len(individual)), mutation_count)
        for i in mutation_indices:
            individual[i] = random.randint(1, hyperparams['code_upperbound'])
        return individual

    def crossover(parent1, parent2):
        crossover_indices = random.sample(range(len(parent1)), int(len(parent1) * hyperparams['crossover_rate']))
        child = np.array([parent2[i] if i in crossover_indices else parent1[i] for i in range(len(parent1))])
        return child

    def tournament_selection(population, scores):
        tournament_size = max(int(len(population) * hyperparams['tournament_size_factor']), 2)
        selected_indices = random.sample(range(len(population)), tournament_size)
        selected_scores = [(i, scores[i]) for i in selected_indices]
        selected_scores.sort(key=lambda x: x[1])
        return population[selected_scores[0][0]]

    def create_new_individual(population, scores):
        parent1 = tournament_selection(population, scores)
        parent2 = tournament_selection(population, scores)
        child = crossover(parent1, parent2)
        mutation_scaling_factor = hyperparams['mutation_scaling_factor'] ** (1 + (maximum_iteration -
        ↪   current_iteration) / maximum_iteration)
        child = mutate(child, mutation_scaling_factor)
        return child

    def elitism(population, scores):
        sorted_population = sorted(zip(population, scores), key=lambda x: x[1])
        return [ind for ind, _ in sorted_population[:hyperparams['elitism_count']]]

    # Main logic for GenerateSample
    population = [np.array(individual) for key in sorted(history_populations.keys()) for individual in
    ↪   history_populations[key]]
    scores = [score for key in sorted(fitness_scores.keys()) for score in fitness_scores[key]]

    new_individuals = []
    elite_individuals = elitism(population, scores) if len(population) > 0 else []

    for elite in elite_individuals:
        new_individuals.append(elite)

    remaining_individuals_count = new_individuals_numbers - len(new_individuals)
    for _ in range(remaining_individuals_count):
        if len(population) > 0:
            new_individual = create_new_individual(population, scores)
        else:
            new_individual = np.random.randint(1, hyperparams['code_upperbound'] + 1, len(best_individual))
        new_individuals.append(new_individual)

    return new_individuals
```

B.2. Code for Ho-2

```
def GenerateSample(history_populations,fitness_scores,best_individual,new_individuals_numbers,current_iteration,
↪   maximum_iteration,hyperparameters):
    hyperparams = {
        'code_upperbound': hyperparameters.get('code_upperbound', 10),
        'mutation_rate': hyperparameters.get('mutation_rate', 0.1),
        'crossover_rate': hyperparameters.get('crossover_rate', 0.6),
        'selection_pressure': hyperparameters.get('selection_pressure', 1.5),
        'elitism': hyperparameters.get('elitism', True),
        'diversity_factor': hyperparameters.get('diversity_factor', 0.05),
        'variance_decay': hyperparameters.get('variance_decay', 0.98),
        'variance_min': hyperparameters.get('variance_min', 0.1),
        'tournament_size_factor': hyperparameters.get('tournament_size_factor', 0.2),
        'elite_diversity_boost': hyperparameters.get('elite_diversity_boost', 2.0),
        'random_individual_chance': hyperparameters.get('random_individual_chance', 0.05),
        'max_mutation': hyperparameters.get('max_mutation', 3)
    }

    # Calculate variance based on current iteration
    variance = max(hyperparams['variance_decay'] ** (current_iteration - 1), hyperparams['variance_min'])

    def mutate(individual):
        # Perform mutation on an individual with limited number of gene changes
        mutation_indices = random.sample(range(len(individual)), min(len(individual), hyperparams['max_mutation']))
        for i in mutation_indices:
            if random.random() < hyperparams['mutation_rate']:
                individual[i] = random.randint(1, hyperparams['code_upperbound'])
        return individual

    def crossover(parent1, parent2):
        # Perform uniform crossover between two parents
        child = np.array([parent1[i] if random.random() < 0.5 else parent2[i] for i in range(len(parent1))])
        return child

    def select_parent(population, scores):
        # Perform tournament selection
        tournament_size = int(len(population) * hyperparams['tournament_size_factor'])
        selected_indices = random.sample(range(len(population)), tournament_size)
        selected_scores = [scores[i] for i in selected_indices]
        winner_index = selected_indices[selected_scores.index(min(selected_scores))]
        return population[winner_index]

    def introduce_diversity(individual, diversity_boost=1.0):
        # Introduce diversity to an individual by adding Gaussian noise
        noise = np.random.randn(len(individual)) * variance * diversity_boost
        individual = np.round(individual + noise)
        individual = np.clip(individual, 1, hyperparams['code_upperbound']).astype(int)
        return individual

    def create_random_individual(length):
        return np.random.randint(1, hyperparams['code_upperbound'] + 1, length)

    # Convert history_population to a list of numpy arrays and fitness_scores to a list of scores
    population = [np.array(individual) for key in sorted(history_populations.keys()) for individual in
    ↪   history_populations[key]]
    scores = [score for key in sorted(fitness_scores.keys()) for score in fitness_scores[key]]

    # Generate new individuals
    new_individuals = []
    for _ in range(new_individuals_numbers):
        if random.random() < hyperparams['random_individual_chance']:
            new_individual = create_random_individual(len(best_individual))
        elif hyperparams['elitism'] and best_individual is not None and random.random() <
        ↪   hyperparams['diversity_factor']:
            # Add a mutated and diversified version of the best individual
            new_individual = mutate(introduce_diversity(best_individual.copy(),
            ↪   hyperparams['elite_diversity_boost']))
        elif len(population) > 0:
            # Create a new individual using crossover and mutation
            parent1 = select_parent(population, scores)
            parent2 = select_parent(population, scores)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_individual = introduce_diversity(child)
        else:
            # If there is no history population, create a random individual
            new_individual = create_random_individual(len(best_individual))
        new_individuals.append(new_individual)

    return new_individuals
```

B.3. Code for Ho-3

```
def GenerateSample(history_populations,fitness_scores,best_individual,new_individuals_numbers,current_iteration,
↪   maximum_iteration,hyperparameters):
    # Define default hyperparameters
    hyperparams = {
        'code_upperbound': hyperparameters.get('code_upperbound', 10),
        'mutation_rate': hyperparameters.get('mutation_rate', 0.2),
        'crossover_rate': hyperparameters.get('crossover_rate', 0.5),
        'selection_pressure': hyperparameters.get('selection_pressure', 2.0),
        'elitism': hyperparameters.get('elitism', True),
    }

    def mutate(individual):
        # Perform mutation on an individual
        for i in range(len(individual)):
            if random.random() < hyperparams['mutation_rate']:
                individual[i] = random.randint(1, hyperparams['code_upperbound'])
        return individual

    def crossover(parent1, parent2):
        # Perform crossover between two parents
        child = parent1.copy()
        for i in range(len(child)):
            if random.random() < hyperparams['crossover_rate']:
                child[i] = parent2[i]
        return child

    def select_parent(population, scores):
        # Perform tournament selection
        tournament_size = min(len(population), int(len(population) * hyperparams['selection_pressure']))
        selected_indices = random.sample(range(len(population)), tournament_size)
        selected_scores = [scores[i] for i in selected_indices]
        winner_index = selected_indices[selected_scores.index(min(selected_scores))]
        return population[winner_index]

    def create_new_individual(population, scores):
        # Create a new individual using crossover and mutation
        if hyperparams['elitism'] and best_individual is not None:
            parent1 = best_individual
        else:
            parent1 = select_parent(population, scores)

        parent2 = select_parent(population, scores)
        child = crossover(parent1, parent2)
        child = mutate(child)
        return child

    # Convert history_population to a list of numpy arrays and fitness_scores to a list of scores
    population = [np.array(individual) for key in sorted(history_populations.keys()) for individual in
    ↪   history_populations[key]]
    scores = [score for key in sorted(fitness_scores.keys()) for score in fitness_scores[key]]

    # Generate new individuals
    new_individuals = []
    for _ in range(new_individuals_numbers):
        if len(population) > 0:
            new_individual = create_new_individual(population, scores)
        else:
            # If there is no history population, create a random individual
            new_individual = np.random.randint(1, hyperparams['code_upperbound'] + 1, len(best_individual))
        new_individuals.append(new_individual)

    return new_individuals
```

B.4. Code of the algorithm discovered by tnGPS in the tnGPS components ablation experiment

```
def GenerateSample(history_populations, fitness_scores, best_individual, new_individuals_numbers, current_iteration,
↪   maximum_iteration, hyperparameters):

    def tournament_selection(populations, fitness_scores, tournament_size):
        selected_indices = []
        for _ in range(tournament_size):
            participants = choices(range(len(populations)), k=tournament_size)
            participants_fitness = [fitness_scores[i] for i in participants]
            winner_index = participants[np.argmin(participants_fitness)]
            selected_indices.append(winner_index)
        return [populations[i] for i in selected_indices]

    def uniform_crossover(parent1, parent2, crossover_rate):
        child = np.array([p1 if random() < crossover_rate else p2 for p1, p2 in zip(parent1, parent2)])
        return child

    def boundary_mutation(individual, mutation_rate, code_upperbound):
        for i in range(len(individual)):
            if random() < mutation_rate:
                individual[i] = 1 if random() < 0.5 else code_upperbound
        return individual

    # Retrieve hyperparameters with defaults
    tournament_size = hyperparameters.get('tournament_size', 3)
    crossover_rate = hyperparameters.get('crossover_rate', 0.7)  # Increased crossover rate for potentially better
    ↪  offspring
    mutation_rate = hyperparameters.get('mutation_rate', 0.05)  # Reduced mutation rate to maintain good traits
    code_upperbound = hyperparameters.get('code_upperbound', 100)
    elitism_count = hyperparameters.get('elitism_count', 1)  # Introducing elitism to ensure the best individual is
    ↪  carried forward

    current_population = history_populations[str(current_iteration - 1)]
    current_fitness = fitness_scores[str(current_iteration - 1)]

    # Sort the current population by fitness and apply elitism
    sorted_indices = np.argsort(current_fitness)
    elites = [current_population[i] for i in sorted_indices[:elitism_count]]

    # Generate new individuals, starting with the elites
    new_individuals = elites.copy()
    while len(new_individuals) < new_individuals_numbers:
        # Tournament selection
        parents = tournament_selection(current_population, current_fitness, tournament_size)
        # Uniform crossover
        child1 = uniform_crossover(parents[0], parents[1], crossover_rate)
        child2 = uniform_crossover(parents[1], parents[0], crossover_rate)
        # Boundary mutation
        child1 = boundary_mutation(child1, mutation_rate, code_upperbound)
        child2 = boundary_mutation(child2, mutation_rate, code_upperbound)
        # Add children to the new population
        new_individuals.extend([child1, child2])

    # Truncate in case we have extra individuals
    return new_individuals[:new_individuals_numbers]
```

B.5. Code of the algorithm discovered by tnGPS in the LLM models and interface description ablation experiment

```
def GenerateSample(history_populations, fitness_scores, best_individual, new_individuals_numbers, current_iteration,
↪  maximum_iteration, hyperparameters):

    # Hyperparameters with default values
    hyper = {
        'code_upperbound': hyperparameters.get('code_upperbound', 30),
        'mutation_rate': hyperparameters.get('mutation_rate', 0.1),
        'tournament_size': hyperparameters.get('tournament_size', 5),
        'elitism_rate': hyperparameters.get('elitism_rate', 0.1),
        'crossover_rate': hyperparameters.get('crossover_rate', 0.9),
        'diversity_factor': hyperparameters.get('diversity_factor', 0.1),
        'best_individual_influence': hyperparameters.get('best_individual_influence', 0.05)
    }

    def tournament_selection(populations, fitness_scores, tournament_size):
        tournament_contestants = choices(list(zip(populations, fitness_scores)), k=tournament_size)
        tournament_contestants.sort(key=lambda x: x[1])  # sort by fitness score, lower is better
        winner = tournament_contestants[0][0]  # return the individual with the best fitness
        return winner

    def mutate(individual, mutation_rate, code_upperbound):
        mutation_indices = [i for i in range(len(individual)) if random() < mutation_rate]
        for i in mutation_indices:
            individual[i] = randint(1, code_upperbound)
        return individual

    def crossover(parent1, parent2, crossover_rate, best_individual, best_influence):
        child = parent1.copy()
        if random() < crossover_rate:
            for i in range(len(parent1)):
                if random() < best_influence:
                    child[i] = best_individual[i]
                elif random() < 0.5:
                    child[i] = parent1[i]
                else:
                    child[i] = parent2[i]
        return child

    def introduce_diversity(population, diversity_factor, code_upperbound):
        for individual in population:
            if random() < diversity_factor:
                mutation_index = randint(0, len(individual) - 1)
                individual[mutation_index] = randint(1, code_upperbound)
        return population

    # Retrieve the latest population and their fitness scores
    elite_population = history_populations[str(current_iteration - 1)]
    elite_fitness = fitness_scores[str(current_iteration - 1)]

    # Calculate the number of elites based on the elitism rate
    number_of_elites = int(hyper['elitism_rate'] * new_individuals_numbers)

    # Sort the elite_population based on fitness and select the top individuals
    sorted_indices = np.argsort(elite_fitness)
    elites = [elite_population[i] for i in sorted_indices[:number_of_elites]]

    # Generate new individuals with crossover and mutation
    new_individuals = []
    while len(new_individuals) < new_individuals_numbers - number_of_elites:
        parent1 = tournament_selection(elite_population, elite_fitness, hyper['tournament_size'])
        parent2 = tournament_selection(elite_population, elite_fitness, hyper['tournament_size'])
        child = crossover(parent1, parent2, hyper['crossover_rate'], best_individual,
        ↪  hyper['best_individual_influence'])
        new_individuals.append(mutate(child, hyper['mutation_rate'], hyper['code_upperbound']))

    # Introduce diversity
    new_individuals = introduce_diversity(new_individuals, hyper['diversity_factor'], hyper['code_upperbound'])

    # Include elites in the new population pool
    new_individuals.extend(elites)

    # Ensure all values are within the specified range
    new_individuals = [np.clip(individual, 1, hyper['code_upperbound']) for individual in new_individuals]

    return new_individuals
```

B.6. Code for TNGA

```
def GenerateSample(history_populations,fitness_scores,best_individual,new_individuals_numbers,current_iteration,
↪   maximum_iteration,hyperparameters):
    Ranking=np.argsort(fitness_scores['{}'.format(current_iteration-1)])
    elite_num=int(len(fitness_scores['{}'.format(current_iteration-1)])*hyperparameters.get('elite_percentage',
    ↪  0.9))
    Ranking=Ranking[0:elite_num]
    populations_elite=[history_populations['{}'.format(current_iteration-1)][i].copy() for i in Ranking]
    fitness_scores_elite=[fitness_scores['{}'.format(current_iteration-1)][i] for i in Ranking]
    Rank_elite = np.argsort(fitness_scores_elite)
    p = [ np.maximum(np.log(hyperparameters.get('alpha', 100)/(0.01+k*5)), 0.01) for k in
    ↪  range(len(populations_elite)) ]
    prob = np.zeros(len(populations_elite))
    for idx, i in enumerate(Rank_elite): prob[i] = p[idx]
    new_individuals=[]
    for i in range(new_individuals_numbers//2):
        parents=choices(populations_elite, weights=prob, k=2)
        female=parents[0].copy()
        male=parents[1].copy()
        index=np.arange(len(male))
        np.random.shuffle(index)
        index=index[0:(len(male)//2)]
        tnp=female[index]
        female[index]=male[index]
        male[index]=tnp
        new_individuals.append(male)
        new_individuals.append(female)
    if np.mod(new_individuals_numbers,2)!=0:
        tnp=new_individuals[-1].copy()
        np.random.shuffle(tnp)
        new_individuals.append(tnp)
    for i in range(new_individuals_numbers):
        mask = np.random.uniform(0,1,[len(new_individuals[0])])<hyperparameters.get('mutation_rate', 0.25)
        for j in range(len(new_individuals[0])):
            if mask[j]:
                mutate_range=np.arange(1,hyperparameters.get('code_upperbound', 15)+1)
                mutate_range=np.delete(mutate_range, np.where(mutate_range == new_individuals[i][j]))
                np.random.shuffle(mutate_range)
                new_individuals[i][j]=mutate_range[0]
    return new_individuals
```

B.7. Code for TNLS

```
def GenerateSample(history_populations,fitness_scores,best_individual,new_individuals_numbers,current_iteration,
↪   maximum_iteration,hyperparameters):
     variance=hyperparameters.get('decay_rate', 0.99)**(current_iteration-2)
     if variance<hyperparameters.get('variance_LB', 0.3):
         variance=hyperparameters.get('variance_LB', 0.3)
     new_individuals=[]
     for i in range(new_individuals_numbers):
         tnp=np.array(best_individual)+np.random.randn(len(best_individual))*variance
         tnp=np.round(tnp)
         tnp[np.where(tnp>hyperparameters.get('code_upperbound', 15))]=hyperparameters.get('code_upperbound', 15)
         tnp[np.where(tnp<1)]=1
         tnp=tnp.astype(int)
         new_individuals.append(tnp)
     return new_individuals
```

B.8. Code for GREEDY

```
def GenerateSample(history_populations,fitness_scores,best_individual,new_individuals_numbers,current_iteration,
↪   maximum_iteration,hyperparameters):
     variance=hyperparameters.get('decay_rate', 0.99)**(current_iteration-2)
     if variance<hyperparameters.get('variance_LB', 0.3):
         variance=hyperparameters.get('variance_LB', 0.3)
     new_individuals=[]
     for i in range(new_individuals_numbers):
         tnp=np.array(best_individual)+np.random.randn(len(best_individual))*variance
         tnp=np.round(tnp)
         tnp[np.where(tnp>hyperparameters.get('code_upperbound', 15))]=hyperparameters.get('code_upperbound', 15)
         tnp[np.where(tnp<1)]=1
         tnp=tnp.astype(int)
         new_individuals.append(tnp)
     return new_individuals
```