From Baselines to Beyond:

An Exploratory Study on Model Effectiveness in Performance Tuning

Huawei Zhang

2347426

hxz036@student.bham.ac.ukl

Abstract

This report is based on the Configuration Performance Tuning task from ISE Lab 3. Using the provided baseline and datasets, I designed and implemented six intelligent tuning methods and evaluated their performance. Among them, two methods—one model-based (Linear FLASH) and one model-free (Linear BestConfig)—achieved superior results compared to the baseline. Additionally, an improved version of the configuration tuning strategy is proposed. This work also investigates the strengths and weaknesses of various initial sampling strategies, explores the theoretical behaviors of different models across systems, and empirically validates two hypotheses regarding the nature of configuration parameters and their impact on system performance.

Introduction

As modern software systems grow increasingly complex, it becomes ever more difficult—for both humans and algorithms—to understand how a growing number of configuration options affect system behavior. For instance, tuning a database system has been shown to be an NP-hard problem [1], and the cost of exhaustively evaluating configurations can be prohibitively high. Yet, software configurations have a critical impact on overall system performance. Therefore, there is a pressing need for intelligent methods that can automatically determine near-optimal configurations with as few evaluations as possible.

While working on Lab 3, reflected on past experiences manually tuning software systems and identified a potentially more effective initial sampling strategy than random sampling.

Motivated by the importance of intelligent software tuning and the desire to test this hypothesis, I chose this topic as the focus of the project.

Related Work

In the field of configuration performance tuning, existing methods are typically categorized into model-free tuning, model-based tuning, and several emerging approaches such as cross-environment tuning, cost-aware tuning, and code-sensitive tuning. However, due to the constraints of the simulated datasets used in this study—specifically, the presence of only a single performance objective and the absence of data from multiple environments or source code features—this work focuses exclusively on model-free and model-based tuning methods. The newer tuning paradigms mentioned above are not applicable under these conditions. In the following sections, my review representative methods from each of the two major categories and discuss their limitations in the context of this task.

3.1 Model-Free Tuning

Model-free optimization does not rely on predictive models; instead, it directly evaluates the performance of each sampled configuration on the target system or dataset. Compared to model-based methods, model-free approaches offer high accuracy since the performance measurements are obtained through actual execution. However, the primary drawback is the high computational cost, as evaluating each configuration can be expensive. Below, is my introduce four classical model-free tuning methods.

3.1.1 Random Search: Random Search blindly samples configurations from the entire search space. While its results are often suboptimal, it can be surprisingly effective when lucky, and in some cases, it may even outperform more sophisticated methods. Moreover, it is immune to several common problems such as the curse of dimensionality, exponential complexity, local optima, and overfitting. In this study, Random Search serves as the baseline for comparison.

3.1.2 BestConfig: Proposed by Zhu et al. [2], BestConfig adopts a two-stage local search framework consisting of Divide and Diverge Sampling (DDS) and Recursive Bound and Search (RBS). In the DDS phase, the configuration space is partitioned into subregions to ensure wide coverage and diversity. In the RBS phase, the algorithm iteratively narrows the search space around high-potential areas based on previously observed performance. BestConfig is commonly applied to single-objective optimization tasks. Given its suitability for this type of tuning problem, I included it as one of the core methods in experiments.

3.1.3 Meta Multi-Objectivization (MMO) : MMO proposed by Chen and Li [3], introduces an auxiliary objective during the search process to reshape the configuration space. This auxiliary objective helps the algorithm escape poor local optima and encourages more effective exploration. However, in the setting, no suitable auxiliary objectives could be identified from the available dataset. Moreover, incorporating excessive prior knowledge may introduce bias. As a result, MMO was not adopted in this study.

3.1.4 NSGA-II: NSGA-II is a well-known multi-objective evolutionary algorithm that has been applied to ORM configuration optimization by Singh et al. [4]. It encodes configurations as binary strings and evolves them across generations using genetic algorithms. However, due to its evolutionary nature, NSGA-II tends to converge slowly and incurs high computational overhead. Given these limitations, i didnt include NSGA-II in evaluation.

3.2 Model-Based Tuning

Model-based optimization leverages predictive models to estimate configuration performance, thereby reducing the number of expensive direct evaluations. These approaches alternate between measuring actual performance and updating the learning model to guide future sampling. Below, i summarize 3 classical model-based tuning methods.

3.2.1 FLASH: proposed by Nair et al. [5], is based on the principle that "exact search is unnecessary." Unlike traditional Bayesian optimization methods that rely on Gaussian Processes (GPs), FLASH employs CART (Classification and Regression Trees) as the surrogate model, which helps mitigate issues related to high dimensionality and smoothness assumptions. The original FLASH framework also includes a sampling strategy called BAZZA; however, this component was not used in the implementation due to practical constraints. Given its simplicity and efficiency, FLASH was considered well-suited to this task and was therefore adopted in my study.3.2.2

3.2.2 BOCA: introduced by Chen et al. [6], extends Bayesian optimization for configuration tuning by replacing the GP surrogate with a Random Forest model. This substitution not only improves scalability but also enhances performance on binary or categorical configuration spaces. However, since most of the systems in my study do not exhibit strongly binary characteristics, BOCA was not included in the experimental evaluation.

3.2.3 Bayesian Optimization: BO is a classical framework for optimizing expensive black-box functions with unknown analytical forms. In the context of configuration tuning, BO employs a Gaussian Process to model the performance surface and an acquisition function to balance exploration and exploitation. Despite its limitations—such as scalability issues and sensitivity to hyperparameters—BO serves as the foundation for several more advanced methods. I implemented a standard BO pipeline as a baseline for comparison.

In this study, two rounds of experiments were conducted: Baseline Surpassing (Part 4) and Further Optimization (Part 5). Across both stages, I designed one custom method (lab3\_forier.py), implemented three existing approaches (lab3\_bayesian.py, lab3\_bestconfig.py, lab3\_flash.py), and proposed enhancements to two of them (lab3\_bestconfig\_linear.py, lab3\_flash.py). Additionally, a novel variation of the DDS (Divide and Diverge Sampling) strategy was introduced, featuring linear time complexity under the assumption of independent configuration options. The study also aimed to empirically validate two hypotheses related to the structure and influence of configuration parameters.

4. Beyond the Baseline

4.1 Experimental Methodology

as described in Part 3 (Related Work), I evaluated six classical configuration tuning methods and ultimately chose to implement three of them—BestConfig, FLASH, and Bayesian Optimization—to compare against a random search baseline. Among these, BestConfig and FLASH served as the primary models for baseline comparison, while Bayesian Optimization was included as a foundational model preceding FLASH.

For this experiment, eight systems (datasets) from the ISE Lab were used. Each dataset consists of a set of configurable options (features) and their corresponding performance outcomes. All columns except the last represent configuration settings, while the final column indicates performance, where lower values are better. Although this setup simplifies the complexity of real-world tuning problems, reducing them to a single-objective minimization task in n-dimensional configuration space, it still serves as a reasonable approximation of realistic scenarios.

To evaluate method performance across different systems and budget levels, I tested each algorithm under six budgets. For each system and budget, the best observed performance was recorded and saved as a .csv file. The results were visualized by subtracting the baseline performance from each method's result, producing comparative plots. In these visualizations, positive values on the x-axis indicate better performance than the baseline, while negative values indicate worse performance. Also to ensure the reproducibility of the results, all methods in the experiments were initialized with a fixed random seed of 42.

Algorithms Compared (5 in total, will be 2 more in part 5):1. Random Search (Baseline) 2. BestConfig (Original) 3. FLASH (No BZZA) 4. Bayesian Optimization (classical).

Budget Settings: 20, 50, 100, 200, 500, 1000

Evaluation Metrics: Performance of the best-found configuration (lower is better);

Seed: 42

Result: 1. Cross-system comparisons of each method 2. Overall budget-wise comparisons

4.1.1 Baseline Method: In this report, random search was used as the required baseline method. It uniformly samples configurations across the search space without incorporating any prior knowledge or model guidance.

4.1.2 BestConfig Method: This is a classic model-free tuning method that follows a two-stage local search strategy: a broad initial exploration followed by focused exploitation in promising regions. The implementation closely follows the original design, using a DDS + RBS framework, with the code provided in the lab3\_bestconfig.py file.

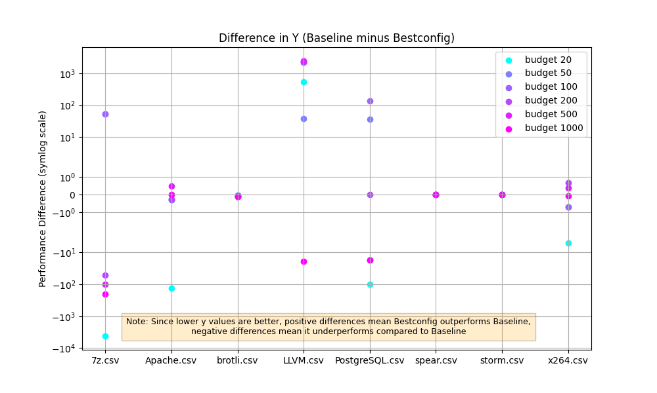
DDS Stage (Divide and Diverge Sampling): Each parameter's possible values are partitioned into k equal-sized intervals (by default, k = 2). A Cartesian product is then taken across all parameter intervals to generate representative subspaces of the configuration space. One valid configuration is randomly sampled from each combination to form the initial set of configurations.

RBS Stage (Recursive Bound and Search): The best-performing configuration from the initial samples is selected as the current optimum. For each parameter, an upper and lower bound is defined around this configuration, and recursive sampling is conducted within the reduced subspace. If a better configuration is found, the bounds and current optimum are updated accordingly. This process continues until the query budget is exhausted.

4.1.3 FLASH and Bayesian Methods: To further explore model-based tuning approaches, I first implemented a standard Bayesian Optimization (BO) pipeline and then derived a simplified FLASH implementation by modifying the BO structure. The implementations are available in lab3\_bayesian.py and lab3\_flash.py, respectively.

Bayesian Optimization (BO): BO was the first model-based method implemented in this study. It uses the Optimizer class from the skopt library to model the performance surface via a Gaussian Process (GP). Each parameter is encoded as either an Integer or Categorical type. The process begins by evaluating 5 initial configurations to train the GP model. Then, using the acquisition function gp\_hedge (default in skopt), the most promising configuration is selected for evaluation. The GP model is iteratively updated, and the best configuration found within the budget is recorded.

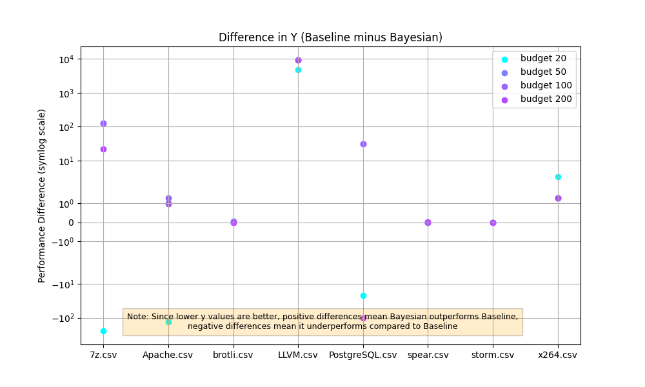
FLASH: To build FLASH, I was started by simplifying the multi-objective BAZZA component from the original paper into a greedy sampling strategy. Although I attempted to replicate BAZZA, the use of GP-based sampling often caused severe slowdowns or failures, as with standard BO. While switching from GP to a decision tree regressor could theoretically address this, model-based methods were computationally expensive on my windows laptop, and I do not have so much time to test it. Moreover, my task focuses on single-objective optimization, where BAZZA is less applicable. As a result, I opted for a simplified FLASH model by replacing GP with a CART (Classification and Regression Tree) regressor and discarding the BAZZA component.

4.2 Experimental Results

4.2.1 BestConfig Results

In Figure 1, BestConfig exhibited relatively average performance. On several systems, its tuning results were comparable to or even worse than—those of Random Search. However, it was significantly faster than the other two methods. Additionally, I observed that BestConfig frequently exhausted its entire budget during the DDS phase, leaving no opportunity to proceed to the more refined RBS stage.

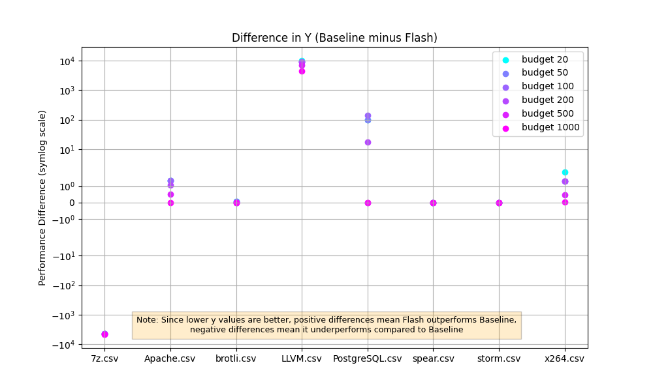
Figure

4.2.2 Bayesian Optimization Results

In Figure 2 BO consistently outperformed both the baseline and BestConfig across most systems. However, BO suffered from significant computational efficiency issues, which prevented us from completing the experiments under the higher budgets of 500 and 1000 queries.

Figure

4.2.3 FLASH Results

In Figre3, FLASH outperformed both the baseline and BestConfig on most systems. While its performance was slightly inferior to Bayesian Optimization (BO) in some cases, FLASH achieved a much lower runtime compared to BO, offering a more practical trade-off between performance and efficiency.

Figure

4.3 Result Analysis

4.3.1 BestConfig Result Analysis

In high-dimensional configuration spaces, BestConfig often consumes the entire budget during the DDS phase, making it impossible to proceed to the RBS phase. In fact, DDS itself may not even complete due to budget constraints, resulting in the loss of its intended coverage and diversity. For example, in most systems with 8 configurable options and a budget of 100, searching the space with even a modest partition size of k = 2 requires evaluating 2^8 = 256 configurations, which already exceeds the allowed budget. Under such constraints, the method fails to sufficiently explore the configuration space, leading to 1. Inadequate sampling coverage during the DDS phase, 2. An incomplete or entirely missing RBS phase, and 3. Overall behavior that closely resembles blind random search.

4.3.2 Bayesian Optimization Result Analysis

First, the inferior performance of Bayesian Optimization (BO) compared to FLASH can be attributed to the tendency of BO to "smooth" the configuration space. This causes its Gaussian Process (GP) surrogate model to perform poorly in highly discrete or non-smooth configuration spaces.

Second, BO's extremely slow runtime is likely due to its computational complexity. Training a GP model has a time complexity of O(n^3), where n is the number of sampled data points. As sampling proceeds, the runtime grows significantly, making BO impractical for large-scale tuning tasks.

4.3.3 FLASH Result Analysis

The relatively fast performance of FLASH can be attributed to its use of CART decision trees in place of GPs, which substantially reduces both training and inference complexity to approximately O (n \* log n). This makes FLASH highly scalable and more suitable for tuning tasks with larger budgets. However, due to the simplified greedy sampling strategy (as opposed to BAZZA's multi-objective approach), FLASH may suffer from premature convergence and a lack of exploration. This explains why, under higher budgets (e.g., beyond 200), FLASH occasionally underperforms compared to BO, which retains a stronger global search capability.

4.3 Reflections on Baseline-Surpassing Experiments

Overall, both FLASH and Bayesian Optimization performed as expected, demonstrating solid performance under various settings. In contrast, although BestConfig is theoretically known for its strong exploration capability, the experiments revealed significant practical limitations when operating under realistic budget constraints. This suggests a clear need to improve or adapt the algorithm to make it more budget efficient.

Furthermore, these observations raise an important question: Could there be underlying structural relationships between configuration options and performance outcomes? Motivated by this possibility, I propose two optimization strategies in the next section, each based on a different real-world assumption, and apply them to enhance the existing algorithms.

5. Further Optimization

5.1 Two Key Assumptions

To address the limitations observed in the previous section, I propose two optimization strategies based on the following real-world assumptions, both inspired by my personal experience with performance tuning in video games:

Assumption 1: The impact of configuration options on performance is smooth and continuous. Through manual tuning of graphical settings in games, I observed that changes to certain options (e.g., resolution) tend to affect frame rates in a linear or exponential fashion. This suggests the possibility of modeling the configuration-performance relationship as a continuous function, which could enable function approximation-based optimization.

Assumption 2: Most configuration options are independent of each other. This assumption is also drawn from gaming experience: different configuration settings often target distinct hardware components (e.g., resolution primarily affects GPU memory), and adjusting one option has minimal effect on the performance influenced by others. For example, increasing texture resolution mainly consumes GPU memory, but if the memory is not fully utilized, it has negligible impact on other subsystems or the overall frame rate.

2 A Model Based on Assumption 1 — Curve Fitting via Fourier Transform

If the relationship between certain configuration options and performance is smooth and can be approximated by a functional form (e.g., linear, exponential, or periodic), then it is theoretically possible to estimate the optimal configuration with only a small sampling budget. To test this hypothesis, I designed and implemented a Fourier Transform-based regression method.

The core idea is as follows: using DDS or random sampling, I first collected a small set of configurations and their corresponding performance values. I then treated each parameter as a one-dimensional signal and the performance value as the target function. For each parameter, I applied a Fourier series fit to approximate the underlying function and infer high-potential configurations via frequency analysis. These estimated optima were then sampled and evaluated.

This method can be effective when the impact of configuration options exhibits periodic or predictable patterns, allowing the Fourier decomposition to approximate the performance surface and skip redundant exploration. However, if the parameters are inherently discrete or non-continuous (such as Boolean or categorical settings), the approximation breaks down. Therefore, this method also serves as a test of how continuous or discrete the dataset truly is. I implemented this approach in the file lab3\_forier.py.

5.3 Optimization Based on Assumption 2 — Linear DDS Sampling

As discussed in Section 4.3.1, the original DDS (Divide and Diverge Sampling) phase of BestConfig suffers from exponential complexity, often exhausting the query budget before meaningful search can begin.

This rapid growth in complexity stems from the assumption that configuration options interact with each other. However, based on Assumption 2—i.e., most configuration options are independent, I argue that we can perform k samples per parameter instead of k ^ d combinations and still achieve reasonable coverage. Under this assumption, I propose an optimized variant called Linear DDS, which reduces the complexity from O (k ^ d) to O (k ⋅ d) while maintaining basic diversity. If the assumption does not hold and the parameters are in fact interdependent, the method should perform no better than random sampling—thus indirectly validating or refuting the assumption.

Key Characteristics of Linear DDS: 1. Linear Complexity: Only k samples are drawn per parameter, reducing the total number of samples to k ⋅ d, a drastic improvement over k ^ d. 2. Coverage: Each parameter’s value space is divided and sampled to maintain basic diversity.

To clarify the implementation, I include the pseudocode below:

Note: In this implementation, I set all non-target parameters to their mode (most frequent value) when generating a sample. This is because using random values often led to configurations with missing performance data in the provided dataset. However, in real-world tuning tasks, where every configuration is measurable, it would be more appropriate to use fully random values instead.

Input:

- data: Configuration dataset (with multiple parameters and performance column)

- k: Number of intervals per parameter (e.g., k = 2)

- budget: Maximum number of samples allowed (e.g., 100)

Output:

- initial\_samples: List of configurations sampled via Linear DDS

Step 1: Initialize parameter intervals

for each configuration parameter col in data:

unique\_vals ← sorted(unique values of col)

intervals[col] ← divide unique\_vals into k equal chunks

Step 2: Initialize sample set

initial\_samples ← empty list

Step 3: Perform Linear DDS sampling

for each configuration parameter col\_index, col:

col\_intervals ← intervals[col]

for each subrange\_index, subrange in col\_intervals:

val ← randomly sample one value from subrange

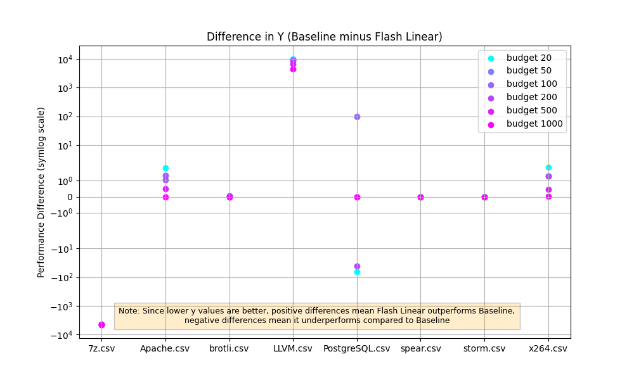
sample ← default configuration (all params set to mode)

sample[col\_index] ← val # replace current parameter

Add sample to initial\_samples

Return initial\_samplesl

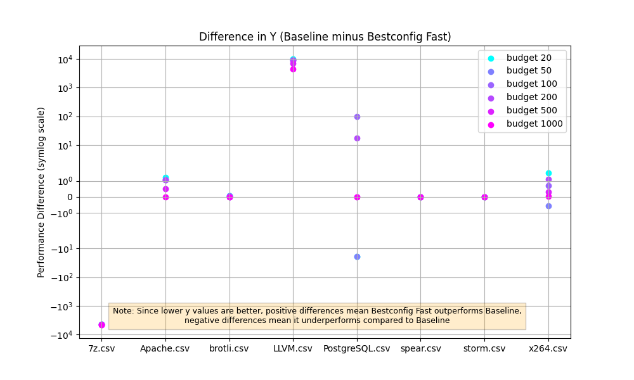
5.4 Experimental Setup

The experimental setup in this section follows the same structure as described in Section 4.1. In addition to the methods tested previously, three additional methods were evaluated: 1.Fourier-based curve fitting (Section 5.2) 2.BestConfig (Fast) — a modified version of BestConfig using Linear DDS instead of the original exponential DDS, and BestConfig（Fast） 3. FLASH (Linear Init) — a simplified version of FLASH where Linear DDS replaces the original random initialization.FLASH（Linear ）

All other experimental conditions—including datasets, evaluation budgets, metrics, and visualization techniques—remain unchanged from Section

5.5 Experimental Results

Figure

The results of the additional experiments are summarized as follows:

The two versions of FLASH (see Figure 3 and Figure 4) produced very similar performance, indicating that FLASH’s effectiveness stems primarily from its underlying surrogate model (CART) rather than its initial sampling strategy.

图表, 散点图

AI 生成的内容可能不正确。The performance of BestConfig Fast, which incorporates Linear DDS (see Figure 1 and Figure 5), was significantly better than the original version.

Figure

As shown in Figure 6, the Fourier-based regression method consistently underperformed the baseline across all tested systems.

It should be noted that the BestConfig method using linear DDS and the two FLASH methods, although the performance of the tuning performance is about the same but in their own speed of operation is several orders of magnitude worse, the first method takes 10 seconds to predict the two FLASH takes 20 minutes to complete the prediction.

Figure

5.6 Reflections on Methods and Assumptions

5.6.1 Fourier-Based Regression Performed Worse Than the Baseline

The assumption behind Fourier Transform is that the target function is globally smooth and exhibits some form of periodicity. However, performance functions in real-world systems are often non-linear and non-smooth, especially in configuration tuning scenarios. My experimental results show that the Fourier-based regression method consistently underperformed the baseline across all systems. This clearly suggests that the assumption of smooth, continuous performance curves does not hold in these datasets.

5.6.2 Assumption 2 Is Largely Valid

The second assumption—that configuration options are mostly independent is holds in most cases. This is evidenced by the fact that Linear DDS significantly improved the performance of BestConfig, especially under tight query budgets (e.g., ≤100). However, there were still a few systems where BestConfig with Linear DDS performed much worse than the baseline, indicating potential parameter interactions.

For FLASH, replacing random initialization with Linear DDS did not yield significant performance differences, reinforcing the idea that FLASH's effectiveness is primarily due to its surrogate model (CART) rather than its initial sampling strategy. That said, the substantial improvements in BestConfig clearly demonstrate that Linear DDS effectively reduces budget waste and enhances configuration space coverage, particularly when query budgets are limited.

These findings suggest that Linear DDS can serve as a general-purpose strategy for improving model-free tuning methods, and potentially even hybrid or model-based methods that suffer from poor initialization or inefficient sampling.

5.6 Further Validation

From the earlier experiments in Sections 4.1 and 5.4, I observed that Linear BestConfig consistently outperformed the baseline when using random seed 42. However, this raised a question: Was the improvement simply due to a lucky initialization from Linear DDS under that particular seed?

To eliminate this possibility, I designed an additional experiment specifically targeting this concern. In this test, I focused solely on Linear BestConfig, running it 100 times with different random seeds (1 to 100). All other experimental conditions—including datasets, evaluation metrics, and budgets—were kept identical to those in Section 5.4.

The final results were visualized and summarized in tables. For each system, I calculated the average performance difference from the baseline across the 100 runs. Due to performance limitations of FLASH, I did not include Linear FLASH in this validation experiment.

5.6.1 Results and Reflection

The results indicate that Linear BestConfig significantly outperformed the baseline in 4 out of 8 systems, performed similarly to the baseline in 3 systems, and underperformed in just 1 system (7z).

These findings provide several important insights:

The improvements observed in earlier experiments were not due to random luck, but rather a robust advantage provided by the Linear DDS sampling strategy.

While the method is not universally superior across all systems, it demonstrates consistent and repeatable advantages in many practical scenarios, particularly under tight query constraints.

The one failure case (7z) does not contradict the assumption of option independence. Instead, it highlights a limitation of BestConfig's RBS (Recursive Bound and Search) phase in highly irregular configuration spaces. The 7z dataset appears to lack any clear performance patterns, making it particularly challenging for local search methods like RBS to be effective. This observation is also consistent with earlier experiments, where other models also struggled to predict or optimize performance on 7z.

图表, 散点图

AI 生成的内容可能不正确。图表, 散点图

AI 生成的内容可能不正确。

图形用户界面

AI 生成的内容可能不正确。图表, 散点图

AI 生成的内容可能不正确。

图表, 直方图

AI 生成的内容可能不正确。图表

AI 生成的内容可能不正确。

图表

AI 生成的内容可能不正确。图表, 散点图

AI 生成的内容可能不正确。

6. Conclusion

Based on the preceding analysis and experiments, I draw the following conclusions regarding the configuration tuning task and experimental environment:

6.1 I ultimately proposed and implemented two methods that successfully outperformed the baseline: BestConfig Linear and FLASH Linear. BestConfig Linear is computationally efficient but exhibits limitations on certain systems due to its reliance on local search. FLASH Linear is slower to execute but leverages a decision tree model that is less prone to local optima compared to RBS. Depending on the characteristics of the system, one method may be more suitable than the other.

6.2 I introduced an improved version of the DDS (Divide and Diverge Sampling) strategy—Linear DDS—which demonstrated strong performance across a wide range of systems.

6.3 My experiments validated two important characteristics of the datasets: The relationship between configuration options and system performance is weakly correlated in most cases. The mapping from configuration values to performance is generally non-continuous, making traditional regression approaches less applicable.

6.4 I evaluated the effectiveness of several established tuning algorithms (e.g., BestConfig, Bayesian Optimization, FLASH) under the specific constraints of this task.

6.5 I demonstrated that traditional curve-fitting techniques, such as Fourier-based regression, are ineffective for this type of tuning problem.

7. Final Solution

As the final deliverable, I provide multiple ways to utilize the proposed methods in practice:

7.1 A graphical user interface (GUI) is available by running bestconfig\_gui.py, which allows users to interactively explore configuration tuning using the proposed methods.

7.2 You can evaluate your own datasets by running main.py, which supports standard input formats and automatically runs the tuning pipeline.

7.3 The methods I implemented are modular and reusable. You can directly import them as Python modules and integrate them into your own codebase.

7.4 To conduct systematic experiments compatible with this task, you can use the command-line interface:

main.py -na allows you to specify parameters such as method name, seed, target system, and more.

7.5 You may also refer to the pseudocode provided in the report, and reuse or adapt my implementation of Linear DDS to deploy your own customized version of this sampling strategy.

8.Artifact

The full source code, dataset, and replication instructions can be accessed via the following

GitHub repository: https://github.com/z588585/lab3

9.Reference

Zhang, et al., "Tuning database configuration via machine learning", VLDB 2020.

Zhu et al., "BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning", USENIX ATC 2014.

Chen and Li, "Meta multi-objectivization for software configuration optimization", ICSE 2019.

Singh et al., "NSGA-II for ORM configuration", JSS 2018.

Nair et al., "FLASH: Fast learning of software configuration performance", TSE 2018.

Chen et al., "BOCA: Optimizing Configuration via Random Forest based Bayesian Optimization", ASE 2020.