LLM PROGRAM OPTIMIZATION VIA RETRIEVAL AUGMENTED SEARCH

Sagnik Anupam, Alexander Shypula, Osbert Bastani
University of Pennsylvania
{sanupam, shypula, obastani}@seas.upenn.edu

ABSTRACT

With the advent of large language models (LLMs), there has been a great deal of interest in applying them to solve difficult programming tasks. Recent work has demonstrated their potential at program optimization, a key challenge in programming languages research. We propose a blackbox adaptation method called Retrieval Augmented Search (RAS) that performs beam search over candidate optimizations; at each step, it retrieves in-context examples from a given training dataset of slow-fast program pairs to guide the LLM. Critically, we find that performing contextual retrieval based on an LLM-generated natural language description significantly outperforms retrieval based on the source code. In addition, we propose a method called AEGIS for improving interpretability by decomposing training examples into "atomic edits" that are significantly more incremental in nature. We show that RAS performs 1.8× better than prior state-of-the-art blackbox adaptation strategies, and that AEGIS performs 1.37× better while performing significantly smaller edits.

1 INTRODUCTION

Given the success of large language models (LLMs) in writing code, there has been significant recent interest in applying them to solve programming tasks. A particularly interesting task is program optimization, a long-standing problem in the programming languages literature that has recently gained increased urgency due to the end of Moore's law. Recent work has shown that LLMs have difficulty with this task out-of-the-box (Shypula et al., 2024)—intuitively, data on program performance is simply not widely available in traditional training datasets, making adaptation necessary.

To address this problem, they propose the "Performance Improving Edits (PIE)" benchmark, and then use it to test a number of carefully designed adaptation strategies to identify effective algorithms for improving performance, including *blackbox* (i.e. prompting-based) adaptation strategies such as instruction prompting Mishra et al. (2022), in-context learning Brown et al. (2020), chain-of-thought prompting Wei et al. (2022), and retrieval augmented generation Lewis et al. (2020). They find *dynamic code retrieval* to be the most effective; this approach retrieves a handful of slow-fast program pair examples from the training set at test time that are most relevant to the current instance (measured using code embedding similarity). These pairs are then used as in-context examples to prompt the LLM. Intuitively, this approach is the only one that makes effective use of the training set, which contributes to its success.

This existing approach is "end-to-end" in the sense that it takes an input program and asks an LLM to directly output an optimized version of that program. However, this strategy differs significantly from how modern compilers work. Rather than making edits inspired by a handful of end-to-end examples, they systematically modify the program through a series of *compiler passes*, each of which is designed to perform a specific kind of optimization. These optimizations are inspired by existing examples, but in a way that generalizes them so they apply to new programs. Thus, a natural question is whether breaking end-to-end optimization into more incremental steps can improve performance.

Inspired by modern compiler design, we propose two novel retrieval-based adaptation strategies. First, we propose *retrieval augmented search* (RAS), which combines two insights to improve dynamic retrieval. First, rather than retrieve based on the code itself, it uses *contextual retrieval*, where it

retrieves examples from the training set based on an LLM-generated natural language description of the program, abstracting the core algorithms and data structures used by the program from how they are implemented on a superficial level. Second, rather than retrieve a fixed set of programs, we perform beam search by iteratively performing the retrieve-optimize-evaluate loop. These two techniques result in a state-of-the-art blackbox technique for adapting LLMs to program optimization.

However, this technique still produces large changes that can be hard to interpret. To further address this issue, we propose *Atomic Edit GuIded Search* (AEGIS), which leverages a preprocessing step to distill generalizable insights from the training data. In particular, we prompt the LLM to decompose a single slow-fast program pair in the training set into a sequence of *atomic edits*, which are incremental modifications associated with a natural language description of the edit, and then explain why the edit might improve performance. The description is intended to be generalizable, abstracting away specifics of the training example from which they are derived. After generating a dataset of atomic edits and examples associated with each edit, when given a new program, we use RAS to first search over incremental edits to this program. Each edit to this program is achieved by retrieving the most relevant atomic edit in our database and then prompting the LLM to apply this atomic edit to the new program. We then perform beam search over sequences of incremental edits to select the resulting program that achieves the greatest performance gain while preserving correctness.

We evaluate our approach using the PIE benchmark Shypula et al. (2024) for C++ program optimization. We show that RAS significantly outperforms dynamic retrieval, a state-of-the-art blackbox adaptation strategy, achieving an 8.01× average speedup compared to 4.42× for dynamic retrieval. Furthermore, AEGIS achieves a 6.08× average speedup, while reducing the average edit size (measured by string edit distance) by 17% when compared to RAS and by 30% when restricting to the first edit in the search process (which is the most substantial one). Hence, RAS performs 1.8× better than dynamic retrieval, while AEGIS performs 1.37× better. These results demonstrate that RAS and AEGIS are promising strategies for blackbox adaptation of LLMs to code optimization.

2 Related Work

Code Optimization. Code optimization has long been a problem of interest for researchers in software engineering and compilers. However, these approaches typically operate at a lower level of abstraction and are incapable of producing high-level optimizations such as changing the algorithms and data structures used. As a consequence, there has been recent interest in leveraging LLMs to augment existing, symbolic techniques. One approach has been to identify code optimizations that are missed by compilers by using LLMs to mutate different programs and use a differential testing method to analyze changes in binary size Italiano & Cummins (2024). Another approach that directly uses LLMs to perform program optimization is the Search-Based LLM (SBLLM) Gao et al. (2024), which proposes an evolutionary search framework to iteratively optimize Python and C++ programs. Separately, they use the BM25 technique to retrieve code snippets displaying useful optimizations, and then ask an LLM to improve their best-performing programs based on these snippets. However, in their framework, retrieval and search are not integrated, and they do not use contextual retrieval. Furthermore, they only report speedups of 1.55× on the PIE benchmark (using GPT-4), so even the existing dynamic retrieval approach studied in PIE substantially outperforms their approach. Finally, Qiu et al. (2025) studies the capabilities of LLMs for Python program optimization, finding significant gaps compared to human experts. We focus on optimizing C++ code since we can measure performance in a reproducible way using a system simulator, as proposed in PIE Shypula et al. (2024).

Code retrieval. Retrieval augmented generation is broadly known to improve code generation Wang et al. (2024). The specific idea of dynamically retrieving relevant in-context examples from a larger training set was first proposed in Poesia et al. (2022) and was later shown to be highly effective for program optimization Shypula et al. (2024). Recently, MapCoder Islam et al. (2024) has shown that retrieving "previously seen" programming examples can improve code generation on the HumanEval benchmark.

While contextual retrieval has recently been popularized for LLMs Anthropic (2024), the idea of annotating code to improve code search has long been studied extensively in software engineering. Older techniques such as Portfolio McMillan et al. (2011) rely on information retrieval methods such as PageRank. More recent work has proposed neural techniques such as CODEnn Gu et al. (2018), which trained a deep neural network to generate embeddings of code snippets and their

Algorithm 1 Retrieval Augmented Search (RAS)

input: $p_0, \Pi_{\text{train}}, F_{\text{opt}}, F_{\text{context}}, R, \phi$
for $i \in [1, ..., m]$ **do**
  $\Pi_i \leftarrow \text{top-}k\{((p, p'), d_\phi(p_{i-1}, p)) \mid (p, p') \in \Pi_{\text{train}}\}$
  $p_i^j \sim F_{\text{opt}}(\pi_i^j, p_{i-1})\ (\forall j \in [k])$ ▷ $\Pi_i = \{\pi_i^j\}_{j=1}^k$
  $p_i \leftarrow \arg\max_{j \in [k]} R(p_i^j)$
end for
return p_m

natural language descriptions; these embeddings could then be matched with embeddings of natural language user queries. The idea of automatically generating the natural descriptions for code snippets artificially was proposed in CoaCor Yao et al. (2019), which trains a bidirectional LSTM to generate natural language descriptions optimized for use by a retrieval model.

3 Retrieval Augmented Search

In this section, we describe our retrieval augmented search (RAS) algorithm for program optimization (overview in Figure 1 and pseudocode in Algorithm 1).

3.1 Problem Formulation

In the program optimization problem, the goal is to take a program $p \in \mathcal{P}$ as input, and output an optimized program $p' \in \mathcal{P}$ that is semantically equivalent to p. Typically, we are additionally given a set of test cases $\{(x_i, y_i)\}_{i=1}^k$ to check correctness; then, denoting the output of program p on input x as $p(x)$, we are searching for programs p such that $p(x_i) = y_i$ for all $i \in \{1, ..., k\}$. While test cases do not guarantee semantic equivalence, they are widely used in machine learning for checking program equivalence (Chen et al., 2021; Li et al., 2022).

We focus on reducing running time, which we denote $R(p) \in \mathbb{R}$. Since we want the fastest correct program, we let $R(p) = -\infty$ if p does not pass one of the given test cases. In practice, measuring a speedup can be difficult due to the stochastic nature of program execution. Recent work has proposed benchmarks that seek to mitigate this issue. The approach used by the PIE benchmark is to measure performance using a system simulator (specifically, gem5 Binkert et al. (2011)), which provides deterministic emulation of hardware, enabling fully reproducible results. Finally, we also set $R(p) = -\infty$ if evaluating p in gem5 times out.

To aid in adaptation, we assume we are given a training set of slow-fast program pairs $\Pi = \{(p, p')\}_{j=1}^n$, where p is an unoptimized program and p' is a hand-optimized program. For instance, the PIE benchmark constructs such a dataset based on sequences of submissions from individual participants in competitive programming challenges Shypula et al. (2024). Given a sequence of submissions $p_1, ..., p_k$, they include pairs $(p_i, p_{i'})$ where $i < i'$ and where $p_{i'}$ is at least 10% faster than p_i according to gem5, i.e., $R(p_{i'}) \geq 1.1 \cdot R(p_i)$. They also provide a subset of *high-quality* training pairs that achieve a more substantial speedup by selecting a subset of the pairs $(p_{i'}, p_i)$ with the highest speedups $\frac{R(p_{i'})}{R(p_i)}$.

Finally, we are interested in blackbox adaptation techniques, which do not adjust the weights of the LLM; instead, they focus on prompting the LLM to improve performance. These prompts can be dynamic (e.g., include dynamically retrieved training examples), multi-turn (e.g., iteratively refine an example based on feedback), or incorporate search (e.g., incrementally apply a sequence of prompts.

3.2 General Framework

We describe the general Retrieval-Augmented Search (RAS) framework for program optimization. In particular, RAS assumes that it is given a training set $\Pi_{\text{train}} = \{(p, p')\}_{j=1}^n$ of slow-fast program pairs, and a new program $p_0 \in \mathcal{P}$ to be optimized. In addition, it assumes it is given a retrieval strategy, which can be expressed as a distance function $d : \mathcal{P} \times \mathcal{P} \to \mathbb{R}_{\geq 0}$ between pairs of programs.

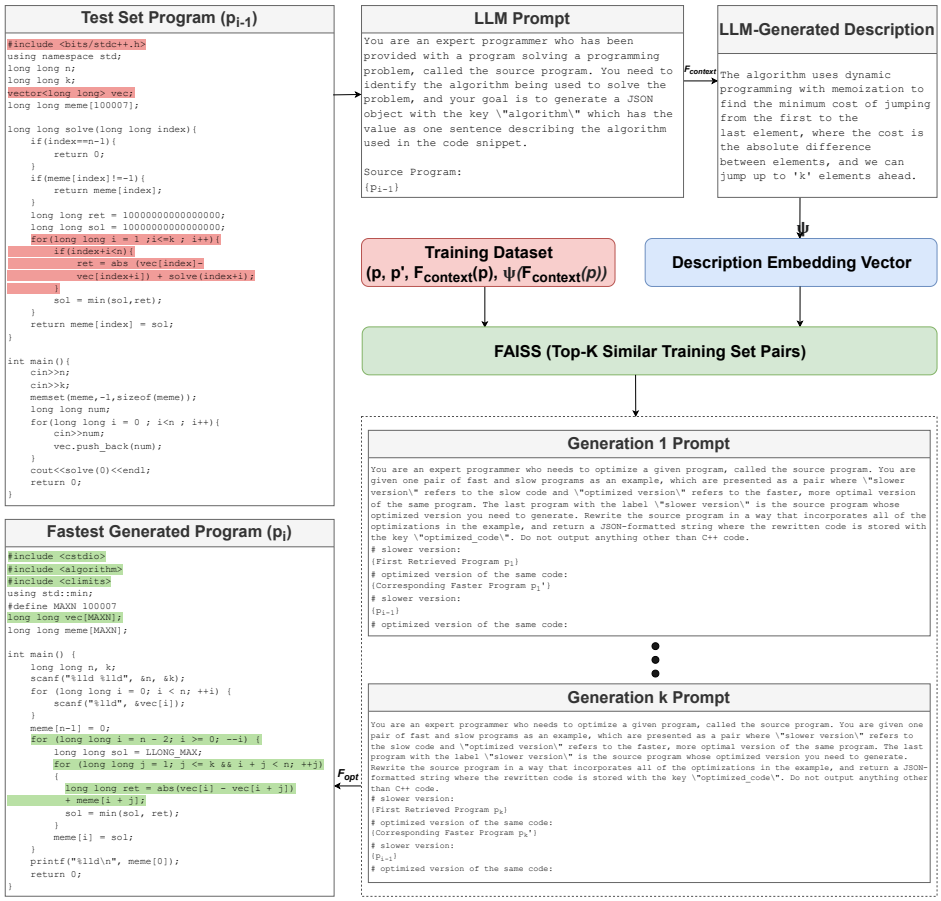


Figure 1: **RAS Framework**: For a given slow program p_{i-1}, we use F_{context} to generate a program description and Ψ to generate its corresponding description embedding vector. We retrieve similar training set programs using FAISS and pass them to F_{opt}. The fastest program generated by F_{opt} is p_i.

Typically, the strategy is defined by an embedding model $\phi : \mathcal{P} \to \mathbb{R}^d$, in which case we can define the distance based on the L_2 distance between the embedding vectors of two programs:

$$d_\phi(p, q) = \|\phi(p) - \phi(q)\|$$

Our framework also assumes blackbox access to an LLM F_{opt}, which takes as input an in-context example of a slow-fast program pair $\pi \in \mathcal{P}^2$, along with a new program p. Then, we can sample optimized versions $p' \sim F_{\text{opt}}(\pi, p)$ of p from F_{opt}. In our implementation, F_{opt} is provided with a system prompt instructing it to try and optimize p.

Now, RAS performs a variation of beam search to optimize p_0, where at each step, it additionally retrieves in-context examples from the training set Π_{train}. In particular, at the ith iteration of beam search (starting from $i = 1$), let p_{i-1} be the current program. Then, we retrieve the top k programs from Π_{train} to form the in-context dataset:

$$\Pi_i = \text{top-}k\{((p, p'), d(p_{i-1}, p)) \mid (p, p') \in \Pi_{\text{train}}\}.$$

Here, top-k selects the k *new* slow-fast pairs (p, p') with the smallest distances $d(p_{i-1}, p)$, using FAISS Douze et al. (2024) for vector search. For any retrieved example π_i^j, we call π_i^j a new pair if F_{opt} did not use π_i^j to sample an earlier best-performing program $p_{\text{opt}} \in \{p_1, \ldots, p_{i-1}\}$. Note that retrieval is performed based on the slow program p; intuitively, we want a slow program that is similar to p_{i-1} so we can apply similar optimizations to p_{i-1} as the ones encoded by the pair (p, p'). Now, for each retrieved example $\pi_i^j \in \Pi_i$, we sample an optimized version of p_{i-1} using π_i^j:

$$p_i^j \sim F_{\text{opt}}(\pi_i^j, p_{i-1}).$$

Finally, we choose p_i to be the fastest program that correctly passes all test cases:

$$p_i = \arg\max_{j \in [k]} R(p_i^j),$$

where $[k] = \{1, ..., k\}$. If no program passes all of the test cases (i.e., $R(p_i^j) = -\infty$ for all $j \in [k]$), or if all programs time out, then we set $p_i = p_{i-1}$. We continue this process for m steps, producing a sequence of programs $p_1, ..., p_m$. Finally, we return p_m. If there is no program at step m that passes all of the test cases and does not time out, we return the source program p_0. Note that the hyperparameters of our algorithm are the number of in-context examples k and the number of iterations m; we describe the choices we use in our experiments in Section 5.

3.3 CONTEXTUAL RETRIEVAL

Our instantiation of RAS uses contextual retrieval to identify relevant in-context examples. We compute $\phi(p)$ by first using an LLM F_{context} to generate a natural language description (i.e., the "context" in contextual retrieval) of p (denoted $s = F_{\text{context}}(p)$), and then applying an embedding model ψ to obtain a vector $\psi(s) \in \mathbb{R}^d$, i.e.,

$$\phi(p) = \psi(F_{\text{context}}(p)).$$

For training examples $(p, p') \in \Pi_{\text{train}}$, we can precompute the embeddings, so the LLM F_{context} only needs to be run once for each one. To construct F_{context}, we use a blackbox LLM that is instructed to describe features such as the algorithms and data structures used by the program; this prompt is shown in Figure 1, alongside an example of a pair (p, s) of program p and its description s.

Finally, we also compare to an ablation that directly embeds the given program—i.e., $\phi(p) = \psi(p)$ for some embedding model ψ; we call this approach *code retrieval*. This is the approach taken in prior work on program optimization Shypula et al. (2024).

4 ATOMIC EDIT GUIDED SEARCH

Next, we describe our approach called *Atomic Edit GuIded Search (*AEGIS*)* designed to improve the interpretability of our framework (overview in Figure 2 and pseudocode in Algorithm 2). AEGIS is inspired by modern compilers, which are designed to perform a sequence of *passes*, which incrementally transform the program to improve performance. Breaking down optimizations into smaller steps has the potential to improve interpretability since the changes from one step to the next may be easier for the programmer to understand. We propose to generate *atomic edits*, which are pairs of programs (p, p') that are semantically equivalent and roughly differ by a single code optimization.

To realize this goal, AEGIS replaces the original training dataset Π_{train} with a dataset of atomic edits Π_{atomic}, and then uses RAS in conjunction with Π_{atomic}. By retrieving atomic edits, we can guide the underlying LLM F_{opt} to perform incremental optimizations rather than large changes. AEGIS constructs Π_{atomic} by using an LLM F_{decomp} to decompose each pair $(p, p') \in \Pi_{\text{train}}$ into atomic edits; then, it aggregates together all of the atomic edits it discovers into the new training set Π_{atomic} of atomic edits.

Specifically, we instruct F_{decomp} to describe the differences between the each slow-fast program pair $(p, p') \in \Pi_{\text{train}}$ as a list; then, the output of F_{decomp} is a list of *natural language edits* $[s_1, \ldots, s_r] \sim F_{\text{decomp}}(p, p')$, where each s_i is a natural language description of an edit in (p, p'). Next, we apply each edit in sequence to the slow program p to obtain a sequence of programs. We do so by initializing $p_0 = p$, and then prompting an LLM F_{edit} to apply natural language edit s_i to p_{i-1} to obtain the next program $p_i \sim F_{\text{edit}}(p_{i-1}, s_i)$ in the sequence; here, F_{edit} is instructed to apply the edit to the given program. Assuming the natural language edits accurately describe how p' is obtained from p, then the final program p_r in this sequence should resemble the original optimization p' of p; in particular, p_r should also be an optimized version of p.

We construct our atomic edit dataset using pairs from the resulting sequence. In particular, for each tuple (s_i, p_{i-1}, p_i), we ask an LLM F_{gen} to generalize s_i so that it can applied to a wider variety of programs; the resulting description $e_i \sim F_{\text{gen}}(s_i, p_i)$ is what we refer to as an atomic edit. Then, our

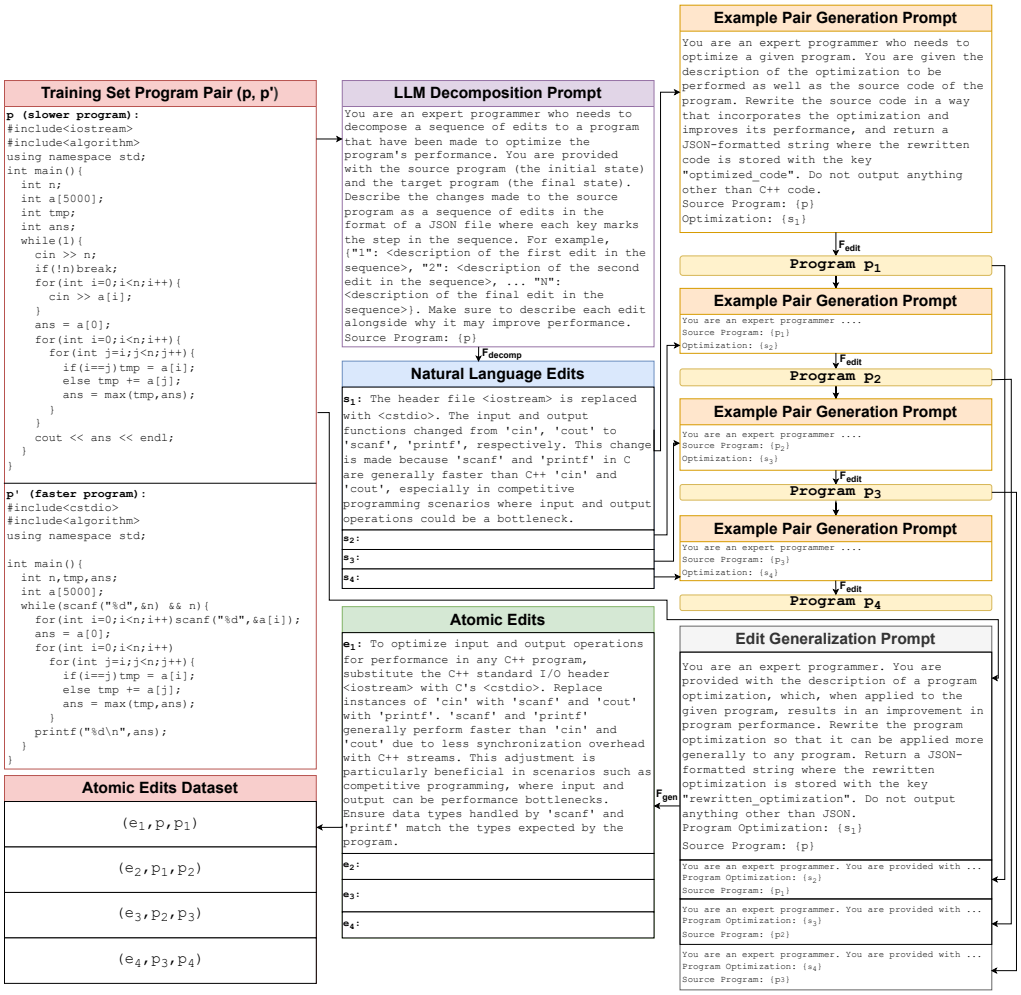


Figure 2: **AEGIS Framework**: For a given training set program pair (p, p'), we identify the natural language edits using F_{decomp}, and then generate intermediate programs implementing each edit by using F_{edit}. Finally, the natural language edits are generalized by F_{gen} to construct atomic edits.

Algorithm 2 Atomic Edit-Guided Search (AEGIS)

input: $\Pi_{\text{train}}, F_{\text{decomp}}, F_{\text{edit}}, F_{\text{gen}}, F_{\text{opt}}, F_{\text{context}}, R$
$\Pi_{\text{atomic}} \leftarrow \varnothing$
for $(p, p') \in \Pi_{\text{train}}$ **do**
  $[s_1, \ldots s_r] \sim F_{\text{decomp}}(p, p')$
  for $i \in [1, ..., n]$ **do**
    $p_i \sim F_{\text{edit}}(s_i, p_{i-1})$
    $e_i \sim F_{\text{gen}}(s_i, p_i)$
    $\Pi_{\text{atomic}} \leftarrow \Pi_{\text{atomic}} \cup \{(e_i, (p_{i-1}, p_i))\}$
  end for
end for
return Π_{atomic}

dataset of atomic edits is

$$\Pi_{\text{atomic}} = \bigcup_{(p,p') \in \Pi_{\text{train}}} \{(e_i, (p_{i-1}, p_i))\}.$$

Finally, we can use RAS with Π_{atomic}, with a slight modification to account for some of the extra information. Specifically, we modify the LLM F_{opt} for program optimization to include both the atomic edit—i.e., given an atomic edit (e, π) and a program p, we sample an optimized version

$$p' \sim F_{\text{opt}}(e, \pi, p).$$

Intuitively, e provides instructions on how p should be optimized, and π shows one example of how e can be applied.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmark. Our experiments are based on the PIE benchmark Shypula et al. (2024), a dataset of slow-fast C++ program pairs constructed from submissions by human programmers to CodeNet Puri et al. (2021). Since competitive programmers iteratively refine their code submissions for better performance, the authors of PIE construct this dataset by first identifying a sequence of programs submitted by the same programmer to solve a single problem. They filter out any incorrect submissions, and then construct slow-fast pairs by executing the C++ submissions on the gem5 simulator Binkert et al. (2011) to measure the running time of the code, discarding any pairs whose difference in performance improvement is less than 10%. We use 4080 high-quality pairs from the PIE dataset as our training set Π_{train}, and 973 test set pairs as a held-out test set Π_{test}. These high-quality pairs are constructed by taking up to 4 pairs in the PIE benchmark's training set with the highest speedup for each competitive programming problem. Importantly, the train-test split in PIE is based on the competitive programming problem being solved, so the training and test set programs are semantically different.

Baselines. We compare our approach to dynamic retrieval, the highest performing blackbox adaptation strategy studied in PIE Shypula et al. (2024). This approach also dynamically retrieves in-context examples from Π_{train}. There are two key differences between our approach and theirs. First, they use retrieval based on the embedding of the code itself rather than contextual retrieval (i.e., code retrieval). Second, they do not perform sequential search; instead, given a program p, they retrieve k in-context examples $\Pi \subseteq \Pi_{\text{train}}$ to provide to the LLM F'_{opt}, and then take multiple samples

$$p^1, ..., p^h \sim F'_{\text{opt}}(\Pi, p).$$

They return the fastest correct program among the h choices.

In addition, we also compare to a "no contextual" ablation of our approach that uses PIE's strategy for retrieval but with search; in particular, it performs code retrieval instead of contextual retrieval. One iteration proceeds as with dynamic retrieval, but we perform multiple iterations. In particular, let p_0 be the initial program; on the ith iteration (starting from $i = 1$), we sample k in-context examples $\Pi \subseteq \Pi_{\text{train}}$ using code retrieval, draw samples $p_i^1, ..., p_i^h \sim F'_{\text{opt}}(\Pi_i, p_{i-1})$, and then let $p_i = \arg\max_{j \in [h]} R(p_i^j)$; as in RAS, we let $p_i^j = p_{i-1}^j$ if $R(p_i^j) = -\infty$ for all $j \in [h]$.

We also consider a "instruct only" approach studied in PIE that performs neither retrieval (i.e., it does not use Π_{train}) nor search; instead, we simply instruct the LLM F''_{opt} to optimize the given program p to obtain an optimized version $p' = F''_{\text{opt}}(p)$, i.e., F''_{opt} is an unadapted LLM. The prompt used in the "instruct only" setting is described in Appendix A, and the remaining prompts are described in Appendix B.

Finally, we include the "human" speedup—for an initial program p, it is the speedup achieved by the fastest correct program p' written by the human participant who wrote p.

Hyperparameters. In our approaches (RAS and AEGIS with contextual retrieval), we use $k = 8$ retrievals and $m = 4$ beam search steps and take $h = 1$ sample per generated prompt. For our baselines, we normalize computation according to the number of calls to the LLM F_{opt}, F'_{opt}, or F''_{opt}. In this calculation, note that for F'_{opt}, the number of retrievals $k = |\Pi|$ does not affect the number of calls $F'_{\text{opt}}(\Pi, p)$, since all examples are included in a single call. Then, for our dynamic retrieval baseline, we retrieve $k = 4$ examples (the same as used in PIE) and take $h = 32$ samples. For our "no contextual" ablation, we retrieve $k = 4$ examples, take $h = 8$ samples per iteration, and use $m = 4$

Approach	Mean Best Speedup	% Optimized
RAS	**8.01**	**0.9640**
No Contextual	5.80	0.8520
Dynamic Retrieval	4.43	0.8191
Instruct Only	2.31	0.5447
Human	3.63	0.9887

Table 1: Results comparing RAS to baselines.

Approach	Mean Best Speedup	% Optimized
AEGIS	**6.08**	**0.9065**
No Contextual	3.85	0.7554
Instruct Only	2.31	0.5447
Human	3.63	0.9887

Table 2: Results comparing AEGIS to baselines.

iterations (the same as our approach). For our "instruct only" ablation, we take $h = 32$ samples and use $m = 1$ iterations. We note that this is different from the standard *pass@k* metric used to evaluate LLM code performance in previous work (such as Chen et al. (2021), where k refers to the number of samples taken from the LLM, which we denote as h in our case. We use k to denote the number of retrieved examples used in the prompt, as done in Shypula et al. (2024).

Compute. All experiments were performed using the `gpt-4o-2024-08-06` model from OpenAI serving as F_{opt}, F'_{opt}, F''_{opt}, F_{decomp}, F_{edit}, F_{gen}, and F_{context}. We use OpenAI's `text-embedding-3-large` as the embedding model ψ. We run the gem5 simulator on a server with 2× Intel(R) Xeon(R) Gold 6342 CPUs (96 cores total).

Metrics. Running gem5 on all test cases to evaluate a single program can be prohibitively computationally expensive due to the large overhead of running gem5. Instead, we measure running time averaged across a subset of 5 randomly selected test cases; these 5 test cases are fixed ahead-of-time. To validate this strategy, we check the correlation between running times on the full test suite vs. our 5 random test cases across all programs in the PIE test set; we find a strong correlation (Pearson's $r = 0.89$, $p < 0.001$; Spearman's $\rho = 0.86$, $p < 0.001$), suggesting that 5 test cases suffices to obtain an accurate estimate of running time.

We report results on the held-out test set $\Pi_{\text{test}} \subseteq \mathcal{P}$ of 973 unoptimized programs provided by the PIE benchmark. Our main metric is "mean best speedup", which is the speedup

$$\text{Speedup}(p, p') = \max\left\{\frac{\text{RunningTime}(p')}{\text{RunningTime}(p)}, 1\right\}$$

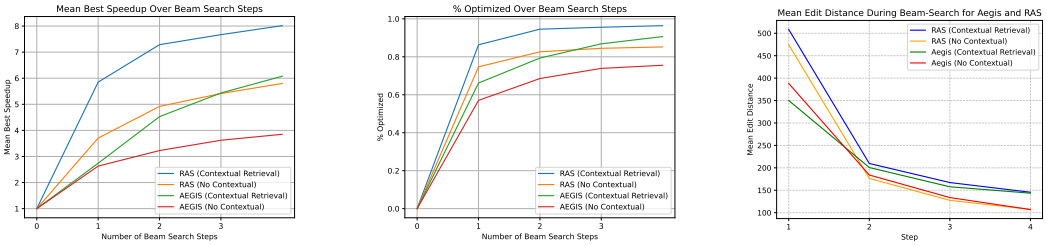
of the final program p' compared to the original program p, averaged across all test programs $p \in \Pi_{\text{test}}$, where The minimum speedup is set to 1 since we can always return p. We also report "% optimized", which is the number of test programs p for which the optimized program p' is at least $1.1\times$ as fast as p. While this metric is not the main goal of our system, it helps capture the diversity of programs that can be optimized using a given approach.

5.2 RESULTS

We show results for RAS in Table 1 and for AEGIS in Table 2. First, note that RAS significantly improves performance compared to all baselines, when using both the original PIE training set as well as our atomic edit training set. Dynamic retrieval was by far the best blackbox adaptation approach studied in the original PIE paper, yet our approach is able to almost double its performance in terms of mean best speedup. Our ablation demonstrates that both search and contextual retrieval are roughly equally important, since ablating contextual retrieval about halves the performance improvement compared to dynamic retrieval.

Method	Mean Edit Distance
AEGIS	**213.05**
RAS	257.77
AEGIS (No Contextual)	**203.24**
RAS (No Contextual)	221.49

Table 3: Comparisons of Edit Distances over Steps between AEGIS and RAS.



(a) Mean Best Speedup (b) % Optimized (c) Mean Edit Distance

Figure 3: Mean Best Speedup, %Optimized and Mean Edit Distance across beam search steps.

While AEGIS diminishes performance, it still achieves a significant improvement. Indeed, it outperforms all ablations (both ablations of AEGIS and those of RAS); the only approach it does not outperform is the full RAS approach.

Metrics across beam search iterations. Next, in Figure 3, we study the effect of using search techniques by reporting our various metrics across iterations of beam search. We focus on our results for our approach compared to our "No Contextual" ablation (since "Dynamic Retrieval" and "Instruct Only" do not perform search).

Figure 3 (a) shows results for "Mean Best Speedup". As can be seen, while the first step of beam search provides the greatest benefit, it continues to provide benefit for all approaches, especially when using contextual retrieval. Since we request the LLM F_{context} to describe the algorithm used for the current best-performing program p_i at each iteration i of the beam search, we hypothesize that F_{context} can update its description to include algorithmic updates made in the previous iteration, thus enabling it to retrieve more relevant examples. We also see greater continuing improvements for AEGIS, likely because atomic edits constrain optimization to change the program more slowly. Additional iterations may help further close the gap between AEGIS and RAS. We provide an example of how AEGIS and RAS both optimize the same program in Appendix C.

Next, Figure 3 (b) shows results for "% Optimized". These results converge substantially more quickly, likely because the first iteration is already enough to get above 1.1× speedup for most programs. Nevertheless, we continue to see gains for our AEGIS approach, again suggesting that continuing search may close the performance gap.

Interpretability. A key motivation for AEGIS is that it should provide greater interpretability by making smaller edits. To study this objective, we consider two metrics. Our main metric is the character-level edit distance of pairs of programs (p_i, p_{i+1}) encountered as part of the search process, with lower edit distances indicating more incremental changes; we consider the edit distance averaged across all pairs of programs and across all programs in the test set.

We summarize results for AEGIS and RAS in Table 3, including results for the "no context" ablations of each approach. As can be seen, AEGIS significantly reduces mean edit distance in both cases. Furthermore, in Figure 3, we show how the mean edit distance changes across steps. As can be seen, AEGIS significantly reduces mean edit distance in the first step, from about 500 to 350. These results suggest that RAS is performing significant optimizations in the first step, and the subsequent steps have smaller edit distance simply because the optimizations are more incremental. Even a single uninterpretable step can make the entire sequence less interpretable, so these results further emphasize the effectiveness of our approach.

6 CONCLUSION

We have proposed RAS and AEGIS, two methods for LLM-guided program optimization that incorporate beam search and retrieval to iteratively optimize a given program. We achieve significant speedups in the blackbox setting (i.e., without any fine-tuning), outperforming existing LLM-based program optimization techniques. AEGIS also aims to improve interpretability by decomposing training examples into "atomic edits" that represent incremental optimizations rather than large changes. We believe that our approach provides a compelling strategy for adapting LLMs to code optimization in the blackbox setting, and may be effective in other code generation tasks as well.

Limitations. A key limitation of both our approaches is that they are more computationally expensive to execute due to our use of beam search. AEGIS also requires additional training-time compute since it uses LLM-generated code to construct its atomic dataset. Nevertheless, we believe our methods pave a promising path towards effective application of LLMs to code optimization in practice.

REFERENCES

Anthropic. Contextual retrieval. `https://web.archive.org/web/20250121234912/https://www.anthropic.com/news/contextual-retrieval`, 2024. Accessed: 2025-01-23.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

Gao, S., Gao, C., Gu, W., and Lyu, M. Search-based llms for code optimization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 254–266. IEEE Computer Society, 2024.

Gu, X., Zhang, H., and Kim, S. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 933–944, 2018.

Islam, M. A., Ali, M. E., and Parvez, M. R. MapCoder: Multi-agent code generation for competitive problem solving. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.269. URL `https://aclanthology.org/2024.acl-long.269/`.

Italiano, D. and Cummins, C. Finding missed code size optimizations in compilers using llms. *arXiv preprint arXiv:2501.00655*, 2024.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378 (6624):1092–1097, 2022.

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 111–120, 2011.

Mishra, S., Khashabi, D., Baral, C., Choi, Y., and Hajishirzi, H. Reframing instructional prompts to gptk's language. In *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 589–612, 2022.

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., and Gulwani, S. Synchromesh: Reliable code generation from pre-trained language models. In *The Tenth International Conference on Learning Representations*, 2022.

Puri, R., Kung, D., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., and Reiss, F. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

Qiu, R., Zeng, W. W., Tong, H., Ezick, J., and Lott, C. How efficient is llm-generated code? a rigorous & high-standard benchmark. *The Thirteenth International Conference on Learning Representations*, 2025.

Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J. R., Yang, Y., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., and Yazdanbakhsh, A. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

Wang, Z. Z., Asai, A., Yu, X. V., Xu, F. F., Xie, Y., Neubig, G., and Fried, D. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2024.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Yao, Z., Peddamail, J. R., and Sun, H. Coacor: Code annotation for code retrieval with reinforcement learning. In *The world wide web conference*, pp. 2203–2214, 2019.

A COMPARING INSTRUCTION PROMPTING AND EXPERT PROGRAMMER SYSTEM ROLES

In our "Instruct Only" baseline, we experiment with two prompts: an instruction-prompting approach (as described in the results of the original PIE benchmark Shypula et al. (2024)), and an "expert programmer" system role. We provide the exact prompts for our approaches here and whenever we refer to programs or retrieved natural language optimizations, we enclose them in braces. Our prompts are as follows:

A.1 INSTRUCTION PROMPTING (IP)

Given the program below, improve its performance:

Program: {Program to be optimized}

Optimized Version:

A.2 EXPERT PROGRAMMER SYSTEM ROLE (EPSR)

System Role: You are an expert programmer who needs to optimize a given program. You are given the source code of the program. Rewrite the source code in a way that optimizes performance such that the program executes faster, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.
User Role: Source Code: {Program to be optimized}

A.3 PROMPT RESULT COMPARISON

We evaluate the two prompts on our dataset of 973 programs by taking $k = 32$ samples for $m = 1$ iteration of search. Our results are presented in Table 4.

Approach	Mean Best Speedup	% Optimized
EPSR	**2.31**	0.5447
IP	2.16	**0.5632**

Table 4: Results comparing differences in metrics due to prompts in Instruct Only setting

Since we observe a slight increase in Mean Best Speedup in the setting with an expert-level system role, we use it in all our other prompts for to maximize efficacy. The "Instruct Only" setting results we report in Tables 1 & 2 use this expert-programmer system role prompt, which is used by F''_{opt}.
.

B PROMPTS FOR EXPERIMENTAL RESULTS

B.1 RAS

B.1.1 PROGRAM DESCRIPTION GENERATION

This prompt is used by F_{context}.

System Role: You are an expert programmer who has been provided with a program solving a programming problem, called the source program. You need to identify the algorithm being used to solve the problem, and your goal is to generate a JSON object with the key "algorithm" which has the value as one sentence describing the algorithm used in the code snippet.
User Role: Source Program:
{Program to be optimized}

B.1.2 GENERATING PROGRAMS WITH CONTEXTUAL RETRIEVAL

This prompt is used by F_{opt}.

System Role: You are an expert programmer who needs to optimize a given program, called the source program. You are given one pair of fast and slow programs as an example, which are presented as a pair where "slower version" refers to the slow code and "optimized version" refers to the faster, more optimal version of the same program. The last program with the label "slower version" is the source program whose optimized version you need to generate. Rewrite the source program in a way that incorporates all of the optimizations in the example, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.
User Role:
slower version:
{Retrieved Slow Program}
optimized version of the same code:
{Retrieved Faster Program}

slower version:
{Program to be optimized}
optimized version of the same code: \n

B.1.3 GENERATING PROGRAMS WITH DYNAMIC CODE RETRIEVAL

This is the prompt used in both the "No Contextual" and "Dynamic Retrieval" settings for RAS, as well as the "No Contextual" setting for AEGIS. It is passed to the model F'_{opt}.

System Role: You are an expert programmer who needs to optimize a given program, called the source program. You are given several pairs of fast and slow programs, called examples, which are presented as pairs where "slower version" refers to the slow code and "optimized version" refers to the faster, more optimal version of the same program. The very last program with the label "slower version" is the source program whose optimized version you need to generate. Rewrite the source program in a way that incorporates all of the optimizations in the examples, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.
User Role:
slower version:
{Retrieved Slow Program 1}
optimized version of the same code:
{Retrieved Faster Program 1}

slower version:
{Retrieved Slow Program 2}
optimized version of the same code:
{Retrieved Faster Program 2}

slower version:
{Retrieved Slow Program 3}
optimized version of the same code:
{Retrieved Faster Program 3}

slower version:
{Retrieved Slow Program 4}
optimized version of the same code:
{Retrieved Faster Program 4}

slower version:
{Program to be optimized}
optimized version of the same code: \n

B.2 AEGIS

B.2.1 GENERATING NATURAL LANGUAGE EDITS

This prompt is used by F_{decomp}.

System Role: You are an expert programmer who needs to decompose a sequence of edits to a program that have been made to optimize the program's performance. You are provided with the source program (the initial state) and the target program (the final state). Describe the changes made to the source program as a sequence of edits in the format of a JSON file where each key marks the step in the sequence. For example, "1": <description of the first edit in the sequence>, "2": <description of the second edit in the sequence>, ... "N": <description of the final edit in the sequence>. Make sure to describe each edit alongside why it may improve performance.
User Role:
Source Program: {Slow Program from Training Set Program Pair}
Target Program: {Faster Program from Training Set Program Pair}

B.2.2 GENERATING PROGRAM SEQUENCE FROM NATURAL LANGUAGE EDITS

This prompt is used by F_{edit}.

System Role: You are an expert programmer who needs to optimize a given program. You are given the description of the optimization to be performed as well as the source code of the program. Rewrite the source code in a way that incorporates the optimization and improves its performance, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.
User Role:
Source Program: {Previous Program in Sequence}
Optimization: {Optimization to be applied to generate next program in the sequence}

B.2.3 GENERATING ATOMIC EDITS FROM NATURAL LANGUAGE EDITS

This prompt is used by F_{gen}.

System Role: You are an expert programmer. You are provided with the description of a program optimization, which, when applied to the given program, results in an improvement in program performance. Rewrite the program optimization so that it can be applied more generally to any program. Return a JSON-formatted string where the rewritten optimization is stored with the key "rewritten_optimization". Do not output anything other than JSON.
User Role:
Program Optimization: {Natural Language Edit}
Program: {Program in program sequence that the edit was applied to}

B.2.4 GENERATING PROGRAMS WITH CONTEXTUAL RETRIEVAL

This prompt is used by the modified F_{opt} when generating programs with AEGIS.

System Role: You are an expert programmer who needs to optimize a given program, called the source program. You are given the description of an optimization that is to be performed on the given program, as well as an example showing how to apply the optimization on an example program (called the example source) to get a target program (called the example target). Rewrite the source code in a way that incorporates all of the optimizations, and return a JSON-formatted string where the rewritten code is stored with the key "optimized_code". Do not output anything other than C++ code.
User Role: Source Program:
{Program to be optimized}
Optimization:
{Atomic edit retrieved via contextual retrieval}

Example Source:
{Slower program in retrieved example pair}
Example Target:
{Faster program in retrieved example pair}

C COMPARISON BETWEEN RAS AND AEGIS

In Figures 5 and 4, we show an example of the optimization trajectory taken by each RAS and AEGIS. As can be seen, RAS concentrates a large number of edits in the first step. In contrast, the edits performed by AEGIS are spread out more evenly across different steps.

Test Set Program

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    long long int n,m,i,r,s=0,b,c,j;
    scanf("%lld",&n);
    long long int a[n+10];
    long long int ar[100001]={0};

    for(i=0;i<n;i++){
        cin>>a[i];
        s+=a[i];
        ar[a[i]]++;
    }

    cin>>m;
    long long int f[m+10];

    for(i=0;i<m;i++){
        cin>>b>>c;
        s+=(c-b)*ar[b];
        ar[c]+=ar[b];
        ar[b]=0;
        cout<<s<<endl;
    }
}
```

RAS Step 1

```
#include <cstdio>
#include <algorithm>
using namespace std;
long long ar[100001] = {0};

inline long long read() {
    long long s = 0;
    char c = getchar();
    while (c < '0' || c > '9') c =
getchar();
    while (c >= '0' && c <= '9') s =
(s * 10) + c - '0', c = getchar();
    return s;
}

int main() {
    long long n = read();
    long long s = 0;
    for (int i = 0; i < n; ++i) {
        int a = read();
        s += a;
        ar[a]++;
    }
    long long m = read();
    while (m--) {
        int b = read();
        int c = read();
        if (b != c) {
            s += (c - b) * ar[b];
            ar[c] += ar[b];
            ar[b] = 0;
        }
        printf("%lld\n", s);
    }
    return 0;
}
```

RAS Step 2

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

long long ar[100001] = {0};
char input_buffer[8000000], *input_ptr
= input_buffer;

inline long long read() {
    long long s = 0;
    char c = *input_ptr++;
    while (c < '0' || c > '9') c =
*input_ptr++;
    while (c >= '0' && c <= '9') s =
(s * 10) + c - '0', c = *input_ptr++;
    return s;
}

void buffered_print(long long x) {
    char out_buffer[20];
    char *out_ptr = out_buffer;
    if (x == 0) {
        putchar('0');
    } else {
        while (x > 0) {
            *out_ptr++ = '0' + (x %
10);
            x /= 10;
        }
        while (out_ptr != out_buffer)
{
            putchar(*--out_ptr);
        }
    }
    putchar('\n');
}

int main() {
    fread(input_buffer, 1, 8000000,
stdin);
    long long n = read();
    long long s = 0;
    for (int i = 0; i < n; ++i) {
        int a = read();
        s += a;
        ar[a]++;
    }
    long long m = read();
    while (m--) {
        int b = read();
        int c = read();
        if (b != c) {
            s += (c - b) * ar[b];
            ar[c] += ar[b];
            ar[b] = 0;
        }
        buffered_print(s);
    }
    return 0;
}
```

RAS Step 3

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

long long ar[100001] = {0};
char input_buffer[8000000], *input_ptr
= input_buffer;

inline long long read() {
    long long s = 0;
    char c = *input_ptr++;
    while (c < '0' || c > '9') c =
*input_ptr++;
    while (c >= '0' && c <= '9') s =
(s * 10) + c - '0', c = *input_ptr++;
    return s;
}

void buffered_print(long long x) {
    static char out_buffer[20];
    static char *out_ptr = out_buffer;
    if (x == 0) {
        putchar('0');
    } else {
        while (x > 0) {
            *out_ptr++ = '0' + (x %
10);
            x /= 10;
        }
        while (out_ptr != out_buffer)
{
            putchar(*--out_ptr);
        }
    }
    putchar('\n');
}

int main() {
    fread(input_buffer, 1, 8000000,
stdin);
    long long n = read();
    long long s = 0;
    for (int i = 0; i < n; ++i) {
        int a = read();
        s += a;
        ar[a]++;
    }

    long long m = read();
    while (m--) {
        int b = read();
        int c = read();
        if (b != c) {
            s += (c - b) * ar[b];
            ar[c] += ar[b];
            ar[b] = 0;
        }
        buffered_print(s);
    }
    return 0;
}
```

RAS Step 4

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

long long ar[100001] = {0};
char input_buffer[8000000], *input_ptr =
input_buffer;

inline long long read_int() {
    long long s = 0;
    char c = *input_ptr++;
    while (c < '0' || c > '9') c =
*input_ptr++;
    while (c >= '0' && c <= '9') s = (s *
10) + c - '0', c = *input_ptr++;
    return s;
}

void write_long(long long x) {
    static char out_buffer[20];
    static char *out_ptr = out_buffer;
    if (x == 0) {
        putchar('0');
    } else {
        while (x > 0) {
            *out_ptr++ = '0' + (x % 10);
            x /= 10;
        }
        while (out_ptr != out_buffer) {
            putchar(*--out_ptr);
        }
    }
    putchar('\n');
}

int main() {
    fread(input_buffer, 1,
sizeof(input_buffer) - 1, stdin);
    long long n = read_int();
    long long s = 0;
    for (int i = 0; i < n; ++i) {
        int a = read_int();
        s += a;
        ar[a]++;
    }

    long long m = read_int();
    while (m--) {
        int b = read_int();
        int c = read_int();
        if (b != c) {
            s += (c - b) * ar[b];
            ar[c] += ar[b];
            ar[b] = 0;
        }
        write_long(s);
    }
    return 0;
}
```

Test Set Program

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    long long int n,m,i,r,s=0,b,c,j;
    scanf("%lld",&n);
    long long int a[n+10];
    long long int ar[100001]={0};

    for(i=0;i<n;i++){
        cin>>a[i];
        s+=a[i];
        ar[a[i]]++;
    }

    cin>>m;
    long long int f[m+10];

    for(i=0;i<m;i++){
        cin>>b>>c;
        s+=(c-b)*ar[b];
        ar[c]+=ar[b];
        ar[b]=0;
        cout<<s<<endl;
    }
}
```

AEGIS Step 1

```
#include <cstdio>
using namespace std;

int main()
{
    long long int n, m, i, r, s = 0,
b, c, j;
    scanf("%lld", &n);
    long long int a[n + 10];
    long long int ar[100001] = {0};

    for (i = 0; i < n; i++) {
        scanf("%lld", &a[i]);
        s += a[i];
        ar[a[i]]++;
    }

    scanf("%lld", &m);
    for (i = 0; i < m; i++) {
        scanf("%lld %lld", &b, &c);
        s += (c - b) * ar[b];
        ar[c] += ar[b];
        ar[b] = 0;
        printf("%lld\n", s);
    }

    return 0;
}
```

AEGIS Step 2

```
#include <cstdio>

int main()
{
    long long int n, m, i, r, s = 0,
b, c, j;
    std::scanf("%lld", &n);
    long long int a[n + 10];
    long long int ar[100001] = {0};
    for (i = 0; i < n; i++) {
        std::scanf("%lld", &a[i]);
        s += a[i];
        ar[a[i]]++;
    }
    std::scanf("%lld", &m);
    for (i = 0; i < m; i++) {
        std::scanf("%lld %lld", &b,
&c);
        s += (c - b) * ar[b];
        ar[c] += ar[b];
        ar[b] = 0;
        std::printf("%lld\n", s);
    }
    return 0;
}
```

AEGIS Step 3

```
#include <cstdio>

int main()
{
    long long int n, m, i, s = 0, b,
c;
    std::scanf("%lld", &n);
    long long int a[n + 10];
    long long int ar[100001] = {0};
    for (i = 0; i < n; i++) {
        std::scanf("%lld", &a[i]);
        s += a[i];
        ar[a[i]]++;
    }
    std::scanf("%lld", &m);
    for (i = 0; i < m; i++) {
        std::scanf("%lld %lld", &b,
&c);
        s += (c - b) * ar[b];
        ar[c] += ar[b];
        ar[b] = 0;
        std::printf("%lld\n", s);
    }
    return 0;
}
```

AEGIS Step 4

```
#include <cstdio>
#include <algorithm>
#define fast
ios base::sync with stdio(false);cin.tie(0);
#define int long long
#define mod 1000000007
const int N = 100001;
int cnt[N];

int32_t main() {
    int n, m, s = 0;
    std::scanf("%lld", &n);
    for (int i = 0; i < n; ++i) {
        int x;
        std::scanf("%lld", &x);
        s += x;
        cnt[x]++;
    }
    std::scanf("%lld", &m);
    for (int i = 0; i < m; ++i) {
        int b, c;
        std::scanf("%lld %lld", &b, &c);
        if (b != c && cnt[b] > 0) {
            s += (c - b) * cnt[b];
            cnt[c] += cnt[b];
            cnt[b] = 0;
        }
        std::printf("%lld\n", s);
    }
    return 0;
}
```

Figure 4: We show a randomly selected example optimization trajectory where RAS and AEGIS implement similar optimizations to achieve similar speedups. The final speedup of RAS is 10.06×, compared to 9.58× for AEGIS. We have highlighted lines that have changed from the previous step in orange, while lines that change in the next step have been highlighted in red. For reference, the human speedup on this example is 1.8×. Here RAS implements an optimization to replace `cin` and `cout` alongside an optimization to ensure that variables are not needlessly updated when `b = c` in Step 1. AEGIS implements a `cin` and `cout` replacement in Step 1 and refines it until Step 3, and then implements the `b!=c` check in Step 4.

Figure 5: We show a randomly selected example optimization trajectory where RAS significantly outperforms AEGIS. Here, we demonstrate the improvements made at each step of RAS vs. AEGIS. The final speedup of RAS on this example is 7.34×, compared to 2.35× for AEGIS. We have highlighted lines that have changed from the previous step in orange, while lines that change in the next step have been highlighted in red. For reference, the human speedup on this example is 1.37×.