Robust Heuristic Algorithm Design with LLMs

Pantea Karimi MIT Dany Rouhana Microsoft Pooria Namyar University of Southern California

Siva Kesava Reddy Kakarla Microsoft Research Venkat Arun
The University of Texas at Austin

Behnaz Arzani Microsoft Research

Abstract — We posit that we can generate more robust and performant heuristics if we augment approaches using LLMs for heuristic design with tools that explain why heuristics underperform and suggestions about how to fix them. We find even simple ideas that (1) expose the LLM to instances where the heuristic underperforms; (2) explain why they occur; and (3) specialize design to regions in the input space, can produce more robust algorithms compared to existing techniques — the heuristics we produce have a ~ 28× better worst-case performance compared to FunSearch, improve average performance, and maintain the runtime.

1 INTRODUCTION

This paper asks whether LLMs can help design *robust* heuristic (approximate) algorithms and whether "old-school" modeling techniques, like heuristic analysis and combinatorial reasoning, can help them do so more effectively¹.

We need to "robustify" heuristics. Deployed heuristics can fail in certain important edge-cases, which can cause catastrophic impact [9], but operators often deploy them anyway [8, 21, 30, 43, 44, 49, 50, 57, 61, 66] because they are faster or more efficient than the optimal. Prior work has found instances of such *deployed* heuristics that have severe performance problems under practical workloads [9, 11, 49].

It is hard to design robust heuristics. Researchers have tried to improve certain heuristics [1, 4, 5, 13, 15, 21, 23, 30, 34, 43, 44, 50, 51, 57, 65]. But the heuristic's performance is tightly coupled with the workloads and hardware — operators have to often re-design or change heuristics as those parameters change. We need to reason across the problem structure, the workload, the hardware, and the behavior of other systems that interact with the heuristic to robustify it.

Our goal is to automate this process so that operators can easily create robust heuristics for any hardware or workload. This allows us to lower the risk of deploying these heuristics.

Recent works use LLMs to improve heuristics [25, 41, 60], and companies have deployed the "synthetic" algorithms they produce [25]. These solutions use LLMs in a search process (e.g., genetic search) where the LLM produces new heuristics based on feedback on the performance of those it produced so far (§2). These tools "evaluate" the code the

LLM produces on random samples from the input space (or samples from their production workloads) — their goal is to improve the average performance of the heuristic and they often ignore important corner cases. We find these solutions only scratch the surface of what's possible: even simple ideas that strategically select inputs (to evaluate the code on) improve the worst-case performance of the heuristics they generate in each step by \geq 20% (Fig. 4)²

It is too much to ask LLMs to design robust heuristics: we show heuristics they generate sometimes underperform (Fig. 3). No matter how cleverly we prompt them, LLMs have a limited circuit size and heuristic design is not a polynomial time problem [2, 24, 58]. LLMs, on their own, often cannot infer why and when a heuristic may underperform. This limits their ability to improve the heuristic (§ 3).

We need heuristics that are resilient to edge-cases, adversarial traffic, and diverse workloads. Workloads change over time [56], and the performance of the heuristic on samples from past instances gives little insight about why the heuristic underperformed and often none about how it may perform on new workloads. This makes it hard to produce robust designs. Past research in networking [5, 50, 57, 65] is a good indicator that knowing why the heuristics underperform is the key that enables us (humans) to make progress (§ 3.3) and it stands to reason the same may hold for LLMs.

We think traditional techniques can help, but we need to enable the LLM to both use them and to interpret their outputs. This is hard.

It would help to provide the LLM with information about why the heuristic underperformed in each case. We need to convert these "explanations" into a form that benefits the LLM-based search: we find it is better to first use the LLM to distill these insights into "suggestions" on how to improve the heuristic (see Fig. 4).

Many heuristics are approximate solutions to NP-hard problems. So, no matter how cleverly we prompt the LLM, a general polynomial-time heuristic that would perform well in the corner cases may not exist. We hypothesize it may be easier to produce an ensemble of heuristics, each specialized to regions of the input space, instead of a generalist that would perform well across the entire space. We tried this

 $^{^1\}mathrm{We}$ discuss why LLM-based approaches are the right technique in $\S 5$

²We find the heuristics we generate (surprisingly) do not harm the runtime and even improve the average performance of the heuristic (Tab. 1).

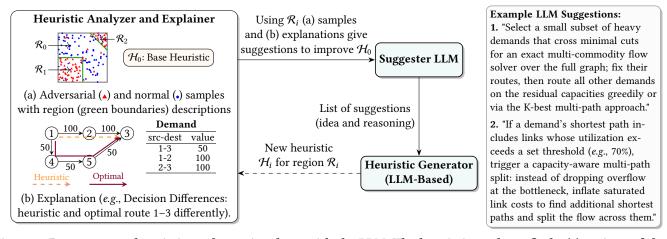


Figure 1: Robusta uses heuristic analyzers in a loop with the LLM. The heuristic analyzer finds: (a) regions of the input space where the heuristic underperforms $(\mathcal{R}_1, \mathcal{R}_2)$; (b) explanations to guide the search. For each region, our solution uses the LLM to suggest how to improve the heuristic for that region and then implements it based on those suggestions. We can run $(\mathcal{H}_1, \mathcal{H}_2)$ for inputs that come from their corresponding region.

idea: we found "good" subregions of the input space and used the LLM to design specialized strategies for each one — the heuristics the LLM produced were more robust (see \S 3.2) compared to the unmodified process. It is hard to produce good regions in general because it depends on the problem structure and impacts the quality of the result.

Some of these problems are easier to tackle in the networking domain: (1) we have well-defined protocols that standardize the network's behavior — we can leverage the predictability this brings to "simplify" the problem; (2) we have a network topology in which we see "repeated" behaviors which we may be able to use to scale our solutions. This intuition is the same that has enabled us to scalably apply techniques from programming languages to networking problems in network verification (e.g., [8, 10, 16, 37]). We expand on these opportunities as we discuss open questions. Summary. We introduce Robusta (Fig. 1), a new solution to LLM-based heuristic design; novel mechanisms that allow us to address some of the challenges we discussed above; and propose research directions that help address the others. The design starts with a base-heuristic and uses heuristic analyzers alongside calls to an LLM to partition the space into regions; "explain" why the heuristic underperforms; and derive "suggestions" on how to improve it. It then modifies the search to (1) create a specialized heuristic for each region; and (2) use the suggestions to find better heuristics. Our initial results show the mechanisms we introduce are viable.

Our contributions are as follows:

- We propose a novel architecture for LLM-based heuristic design through explanation-guided genetic search.
- We use a case study of a traffic engineering problem to show these mechanisms are viable. We introduce

- certain design principles and discuss techniques that we found ineffective. We show ROBUSTA finds heuristics that are 28× better than the baselines.
- We discuss open questions and the opportunity to solve them. We also describe how we may solve these challenges and why they may be more tractable problems to solve in the networking space.

For our initial prototype, we use existing heuristic analyzers and algorithms that explain heuristic performance, such as MetaOpt [49] and XPlain [31], but our architecture is general and can support any similar tool.

2 BACKGROUND

Researchers and practitioners have used LLMs for program synthesis. Many copilots provide an interactive framework to help write programs [7, 20, 27, 35, 53], sometimes with human guidance [12]. Others generate tests to enable LLMs to critique their own outputs [40, 54, 59]. Researchers have used formal methods to guide LLM-based synthesis across domains such as program generation [22, 47], loop-invariant inference [63], and network configuration management [46].

Recent work on LLM-based heuristic design presents a new breakthrough [3, 25, 29, 41, 60, 64]. We discuss Fun-Search [60] as an example. FunSearch uses genetic search algorithms where the LLM does cross-over (combines different heuristics to create stronger ones) and mutation (introduces random changes to the heuristic to explore the space).

FunSearch's algorithm starts with a *set* of base heuristics. These heuristics are the initial seed to "clusters" which the algorithm evolves over time. Users provide an "evaluation function" that scores the heuristics based on the heuristic's performance on random samples from the input space.

FunSearch iterates as follows: (1) it evolves the heuristics within each cluster for a pre-specified amount of time; (2) it then evaluates the heuristics in each cluster and assigns a score to the cluster based on the best performing heuristic in it; (3) the algorithm then removes half of the lowest scoring clusters and spawns new ones to replace them. To seed the new clusters, the algorithm uses the best heuristic from the clusters that survived the previous round. The algorithm terminates after a pre-set timeout.

3 THE PROMISE AND OPEN QUESTIONS

LLM-based program synthesis differs from FunSearch—and from our focus—in an important way: synthesized programs are typically judged as either correct or incorrect. Heuristics, by contrast, lack such a binary criterion. Their evaluation is based on performance, which is not only quantitative but often nuanced. Which parts of the input space matter? Should we judge the heuristic based on its average performance? on the tail performance? or the worst case? What input distributions should we use to evaluate the heuristic? should we focus on specific input instances or on the entire space?

If we measure its average performance, then we may potentially mask the heuristic's poor performance on rare (but critical) inputs because of its strong performance elsewhere. This is especially true when we evaluate the heuristic on arbitrary or random samples from the input space.

A single counter-example rarely captures the full extent of a heuristic's weaknesses. A heuristic's performance often suffers under many different "types" of inputs, which makes it harder to diagnose and improve. Many of these heuristics try to find polynomial-time approximations for NP-hard problems or faster heuristics for polynomial-time problems (e.g., in traffic engineering). Often, no single fast heuristic can perform well across the entire input space — we are more likely to succeed with specialized heuristics that combine multiple complementary strategies.

We next discuss these concepts in depth, show how they help generate better heuristics through a concrete example, and discuss open problems our community needs to solve in order to apply these techniques in practical settings.

3.1 Adversarial samples or random ones?

FunSearch and similar algorithms evaluate heuristics they generate on random samples over the input space and use the result as feedback to guide the search. But such random samples — especially those that come from a uniform (or other ad-hoc) distributions — can miss important regions of the input space. A heuristic may perform well on most of these inputs, yet underperform on important practical instances. Such instances occasionally do happen in real workloads and cause severe consequences [9, 11, 49].

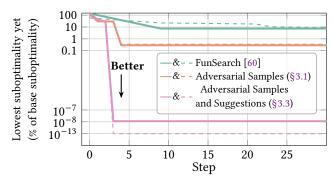


Figure 2: We compare FunSearch with solutions where we (1) focus on adversarial samples in each step (§3.1); and (2) also incorporate "suggestions" in each step (§3.3); on a traffic engineering problem. Dashed lines report the lowest (best) suboptimality achieved for a heuristic on training samples so far, and the solid lines measure it for a held-out set of adversarial samples. The suboptimality is the worst-case performance of a heuristic compared to the optimal across all samples.

Promise. If we evaluate the average case performance of a heuristic, we can mask the scenarios in which the heuristic underperforms, especially if these occur in small pockets of the input space. It is not easy for a user to modify the sampling distribution to fix this issue (to increase the probability that the evaluator draws samples from regions where the heuristic may underperform) because they may not be able to identify when, where, and how they should do so. But we hypothesize if we bridge this gap we can then guide the synthesis process towards more robust solutions.

Our preliminary results support this hypothesis. In experiments where we targeted Microsoft's traffic engineering heuristic [33, 48]—which "pins" small demands to their shortest paths and optimally routes the rest—we expose the LLM to adversarial inputs (input instances that cause the largest performance problems for the heuristic) and their individual performance suboptimalities (the performance gap between the heuristic and the optimal solution)³. We found that even this simple signal allowed FunSearch to create heuristics that outperformed (they had a lower suboptimality) what it found before (Fig. 2). We describe the experiment details in § 4.

Open questions. Ideally, we would revise the adversarial inputs at each iteration of the search process, since they evolve alongside the heuristic. But most tools that produce adversarial inputs require a mathematical model of the heuristic [2, 8, 49]. We found LLMs, out of the box, could not provide such models. Thus, in our experiments, we generated the adversarial instances only for the base heuristic and reused

 $^{^3}$ We define the (overall) suboptimality as the worst-case suboptimalities across all input instances

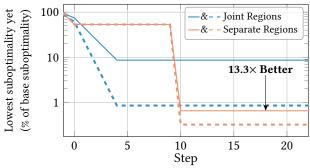


Figure 3: we create an approach which uses FunSearch to create specialized heuristics for each region (separate regions) with one that generates a single heuristic to operate over the entire space. We find specialized heuristics outperform monolithic ones. The dashed and solid lines show overall suboptimality so far on training and held-out samples, respectively.

them to evaluate all subsequent heuristics during the search. Even this limited signal was valuable.

Prior work has used LLMs to produce mathematical models for certain problems [36, 38] — we have reason to be optimistic. Several domain-specific languages (DSLs) already exist to make such modeling accessible—for example, MiniZinc [45] and AMPL [26]. XPlain [31] introduces a DSL that helps model heuristics as optimization problems. These DSLs may enable LLMs to model heuristics automatically.

Another challenge is scalability. SMT-based [2, 8, 10] and optimization-based heuristic analyzers [31, 49] are slow. This is acceptable for offline analysis of a single heuristic, but it becomes a bottleneck in search-driven synthesis frameworks that need to quickly iterate over many heuristics. We can mitigate this problem with domain-specific strategies. For example, we can use partitioning techniques inspired by NCFlow [1] and POP [51] to scale heuristic analysis for widearea networks [49]; or graph isomorphism for those that target data centers [6, 14, 62]; or use specialized encodings to analyze queues or schedulers [8, 49]. But we need more research to automatically find and apply the appropriate scaling strategy (and maybe even more than one).

3.2 Ensemble of heuristics or just one?

Inputs in different regions of the input space often exhibit shared structural properties which a specialized heuristic can exploit. An ensemble of specialists is likely to outperform generalists that FunSearch-like techniques produce.

Promise. We use XPlain [31] to partition the input space into regions where the base heuristic underperforms. We then compare the performance of two approaches: one where FunSearch generates a single general-purpose heuristic, and another where it synthesizes a separate heuristic for each region, which we then combine into an ensemble (Fig. 3).

In our experiments, one region of the input space consisted of scenarios with a few large demands that shared bottlenecks with many smaller flows. The base heuristic, which "pins" small demands to their shortest paths, performed poorly here: it pre-committed small flows to a route and left insufficient capacity for larger demands. The specialized heuristic reversed this order: it sorted demands by size and allocated capacity to larger flows before smaller ones.

While the heuristic above may not always perform well (it is slower since it does not remove as many demands from the optimization as the base and since it prioritizes large demands its mistakes are more costly) but it improves performance *in this region*. Microsoft deployed the "pinning" heuristic because such skewed demands are less common. But recent work has shown when such demands happen in practice they can degrade performance by 30% [49]. Through this specialized heuristic, we can have the best of both worlds and switch to the specialized heuristic when we see demands that fit this pattern.

Operators can either inspect the demand in each iteration to select an appropriate heuristic, or run multiple heuristics in parallel and pick the best.

Open questions. We used regions that trigger a similar behavior in the base heuristic [31]. But there are other viable approaches. For example, symbolic execution tools such as Klee [18] allow us to find equivalence classes of inputs that trigger the same code-paths in an algorithm — we can view each such code path in the optimal algorithm, the base heuristic, or the Cartesian product of both as a new region.

There is an opportunity to define these regions in a domainspecific way and based on practical insights. For example, graph-partitioning algorithms [17, 52] generate good equivalence classes for traffic engineering heuristics. We need to develop techniques to select the best approach automatically.

Our approach is expensive at larger scales. We hypothesize that the principle of bounded model checking may help. It may be possible to (1) derive the regions for lower-dimensional inputs and then project them into regions in the higher-dimensional space; and/or (2) find "concepts" that describe how to improve the heuristics based on inputs in lower-dimensions and then apply those concepts in the higher-dimensional space (this is somewhat similar to [55]); and/or (3) express partitions in a "scale-invariant" DSL. The LLM itself may help enable such solutions.

We have some evidence that this may work. We used our approach and a medium sized (see §4) topology to generate more robust heuristics and then evaluated the heuristics we generated on a large one (one with 196 nodes and 486 edges). The worst suboptimality (as determined by running a heuristic analyzer on the larger topology) was 46% on the larger topology when we ran the base heuristic. The heuristic

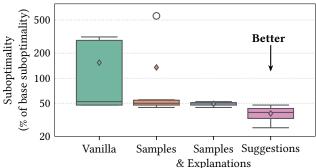


Figure 4: Explanations benefit the search. We consider one step of the search process and see that we can find better heuristics if we first "explain" why a heuristic underperforms. We can improve the approach further if we convert these explanations into suggestions on how to improve the heuristic. We evaluate all of the approaches on a held-out set of adversarial samples.

we created based on the smaller topology, without further modification, reduced this to 27%.

We defined regions statically (we define them once and we analyze the base heuristic to do so). The base heuristic we start with may not be a good one — this heuristic just seeds the process and is only meant as a starting point for the search. It may be unwise to define the regions based on this heuristic. We can solve this problem if we instead define the regions based on the optimal algorithm instead (but this can be too slow in most cases) or if we re-evaluate the regions as we find better heuristics during the search. How and when to apply each approach is an open problem.

3.3 Do explanations help?

Researchers have worked to improve networking heuristics for decades. SWAN [30] designed an algorithm that outperformed the traffic engineering solution Danna *et al.* [21] had proposed and Soroush [50] then design an algorithm that improved it further. Cassini [57] improves Themis [43]. TACCL [61] improves MSCCL [19] which is itself outperformed by [66]. PACKs [5] designs a new packet scheduling algorithm that improves upon SP-PIFO [4] and AIFO [65].

Most of these works identify *why* a heuristic underperforms and address that root cause directly. For example, to improve max-min fair resource allocation algorithms, researchers found the core difference between the heuristics and the optimal was in the heuristics' ability to approximately sort demands based on how much capacity the algorithm assigned to them [21, 30, 50]. Each new solution targeted this root cause and achieved better and better approximations more quickly. We find a similar methodology in the novel heuristics designed by the MetaOpt authors [49] for the demand-pinning heuristic in traffic engineering and for the SP-PIFO packet scheduling algorithms. The CCAC

authors [2] also did the same for congestion control. We thus think explanations are the key to better heuristic design. **Promise.** Fig. 4 confirms this. We evaluate whether explanations improve the quality of the heuristic the LLM produces in each step of the search (we can think of this as a single-shot experiment where we only run one step of the search). We show the result of the full (where we approximate the explanations in each step) search in Fig. 5.

The vanilla approach prompts the LLM to improve the base heuristic. The "samples" approach changes the prompt to include samples that cause the heuristic to underperform (this captures the idea that "adversarial samples" can help design better heuristics). The "samples and explanations" approach feeds the LLM decision differences (similar to that in prior work [31, 64]) that characterizes where the heuristic and the optimal took different actions for the same inputs in the sample-set. The suggestions approach first converts the samples and explanations into "suggestions" (through calls to the LLM) for how to improve the heuristic; feeds the suggestions into the LLM one at a time to create new heuristics; and returns the best one. Each approach prompts the LLM ten times to create a new heuristic and reports the best performance across these ten heuristics.

Suggestions are the most effective. This is because, if we directly include the samples and explanations, we increase the length of the prompt which degrades the LLM's performance (it is well-known that LLM's performance degrades on longer contexts [28, 39, 42]). Suggestions summarize the relevant information in these inputs and shorten the context.

Higher dimensional samples obscure the root cause for LLMs and humans alike and degrade performance further. Our results show the baseline improves on lower-dimensional samples. But we need large enough instances so that we can trigger the mechanisms that cause the heuristic to underperform (some problems may only manifest at large *enough* dimensions). Suggestions may help summarize such instances. **Open questions.** The above points to an interesting property. If the principles of bounded model checking apply then it is better to first find the smallest instance where we can observe the heuristic's performance problems and try to improve it with samples and explanations we find in that scale. This is hard as the right scale can depend on the heuristic itself and other aspects of the problem.

How can we explain why a heuristic underperforms? Prior works show initial steps towards such explanations [8–10, 31], but none can analyze the code the LLM generates.

While we show suggestions help FunSearch, it remains an open question (once we find a way to automatically generate suggestions to improve new heuristics) how to use these suggestions throughout the search and whether updated suggestions can result in further gains (in our evaluations in

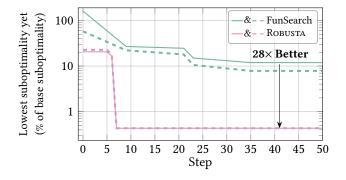


Figure 5: We evaluate our ideas end-to-end. We create specialized heuristics on 5 regions and use approximate suggestions. ROBUSTA finds heuristics with better worst-case performance.

 $\S 4$ we generate approximate explanations but exact explanations can help improve the results even more). The LLM often suggests multiple mechanisms for how we can improve the heuristic, we currently use each one to start the search but it remains open how to use them *during* the search to either create new clusters ($\S 2$) or update existing ones.

4 EVALUATION

We propose Robusta, that incorporates these ideas (Fig. 1). In each step, Robusta evaluates the new heuristics the LLM produced, finds new adversarial samples (and maybe new regions), and suggests how to improve the heuristic. We then use the LLM to implement the suggestions and improve the heuristics in each step. The rest is similar to FunSearch (§ 2).

We evaluate a simple execution of this design end-to-end. To mimic explanations throughout the process we "approximate" them: we use the adversarial inputs we found for the base heuristic as inputs to the heuristics the LLM produces in each step of the search and then map the output to the edges in the XPlain DSL (these edges capture the "actions" the heuristic takes — we ignore the node behaviors in the DSL which enforce the heuristic behavior as we have already concretized those actions). This approach is approximate because the initial adversarial regions may not contain samples that make the new heuristic underperform.

As the search progresses the heuristic improves and we have less and less samples where the heuristic underperforms. This continues until we no longer have enough information to generate meaningful suggestions — we no longer create suggestions after this point.

We experiment with a traffic engineering heuristic [49] which "pins" small demands to their shortest path and optimally routes others. We use a 20-node, 30-link sub-graph of the CogentCo topology from the Topology Zoo [32]. We impose a runtime limit of 120 second across all heuristics,

	Method	Max suboptimality (% of base suboptimality)	Mean suboptimality (% of base suboptimality)	Runtime (× the base)
	Robusta	0.5%	0.01 ± 0.002%	1.00×
	FunSearch	14 %	$2.07 \pm 0.097 \%$	0.96×

Table 1: Robusta outperforms FunSearch.

matching the runtime of the base heuristic on this topology. This implicitly gives feedback to the LLM about the runtime.

We use MetaOpt [49] to find adversarial samples where the base heuristic underperforms. MetaOpt [49] is a heuristic analyzer that takes a model of the optimal solution to a problem and the heuristic and finds input instances that cause that heuristic to underperform. To find multiple adversarial inputs we implement the idea in XPlain [31] which extends MetaOpt to find adversarial *subspaces* (regions of the input space where the heuristic underperforms).

We use Open-AI's o4-mini and repeat each experiment 20 times to account for the randomness in the results. To evaluate each approach we use 1500 held-out samples (we use the same set to evaluate each approach) which we do not expose the LLM to when we create the heuristics.

We evaluate the performance of each algorithm across 50 "step" where a step is one where each island produces a new heuristic. We report "normalized suboptimality" values where the suboptimality is the maximum difference between the performance of the heuristic and that of the optimal algorithm, and we normalize by the suboptimality of the base seed heuristic. Our results (Tab. 1 and Fig. 5) show Robusta produces heuristics that have 28× better worst-case performance, and \sim 200× better performance on average compared those FunSearch creates. The runtime of the heuristics the two approaches create is similar.

5 THE PATH FORWARD

We described how we can help LLMs design better heuristics and open questions in this space (§3). We did not discuss a number of other aspects of this problem:

Are LLMs the right method to use? We posed a specific question: can explanations help LLM-based search algorithms create better heuristics? But it is unclear whether LLMs are the right tool to use. We think they might be: they are flexible and provide a convenient user interface. We need more research to evaluate what the potential downsides of such solutions are and their potential impact.

Applications in other areas in networking. Recent work has applied heuristic analyzers to failure analysis problems [11] where they find the set of *probable* failures that cause the network to experience the worst-case performance compared to its healthy state. There may be an opportunity to extend the approaches we discuss here and use ROBUSTA to help with capacity planning under failures.

The impact on average performance and runtime. We spent most of this paper on how to make heuristics more robust. We control the runtime of the heuristics we generate through explicit timeouts which is why all of our improved heuristics have the same runtime as the base one. Our evaluation showed the new heuristics also improved average case performance. This may not always be the case and we need more research to balance between these two objectives.

REFERENCES

- [1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. 2021. Contracting wide-area network topologies to solve flow problems quickly. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). 175–200.
- [2] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2024. Towards provably performant congestion control. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). 951–978.
- [3] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2024. AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and treefinement. arXiv preprint arXiv:2412.06176 (2024).
- [4] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. {SP-PIFO}: Approximating {Push-In} {First-Out} behaviors using {Strict-Priority} queues. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). 59-76.
- [5] Albert Gran Alcoz, Balázs Vass, Pooria Namyar, Behnaz Arzani, Gábor Rétvári, and Laurent Vanbever. 2025. Everything matters in programmable packet scheduling. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). 1467–1485.
- [6] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk based planning of network changes in evolving data centers. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. 414–429.
- [7] Anysphere. [n. d.]. Curser. https://cursor.com/en
- [8] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal methods for network performance analysis. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23).
- [9] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. 2022. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 177–192.
- [10] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward Formally Verifying Congestion Control Behavior. In Proceedings of the 2021 ACM SIG-COMM 2021 Conference (SIGCOMM '21).
- [11] Behnaz Arzani, Sina Taheri, Pooria Namyar, Ryan Beckett, Siva Kakarla, and Elnaz Jallilipour. 2025. Raha: A General Tool to Analyze WAN Degradation. In Proceedings of the ACM SIGCOMM 2025 conference.
- [12] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021).
- [13] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. 2023. Virtual machine allocation with lifetime predictions. Proceedings of Machine Learning and Systems 5 (2023), 232–253.
- [14] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract

- topologies. In Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. 437–451.
- [15] Julia A Bennell, Lai Soon Lee, and Chris N Potts. 2013. A genetic algorithm for two-dimensional bin packing with due dates. *International Journal of Production Economics* 145, 2 (2013), 547–560.
- [16] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023* Conference. 122–135.
- [17] Peter Brucker. 1978. On the complexity of clustering problems. In Optimization and Operations Research: Proceedings of a Workshop Held at the University of Bonn, October 2–8, 1977. Springer, 45–54.
- [18] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In OSDI, Vol. 8. 209–224.
- [19] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 62–75.
- [20] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021).
- [21] Emilie Danna, Subhasree Mandal, and Arjun Singh. 2012. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In 2012 Proceedings IEEE INFOCOM.
 IEEE.
- [22] Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. 2023. DreamCoder: growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. Philosophical Transactions of the Royal Society A 381, 2251 (2023), 20220050.
- [23] Emanuel Falkenauer, Alain Delchambre, et al. 1992. A genetic algorithm for bin packing and line balancing.. In ICRA. Citeseer.
- [24] Azadeh Farzan and Zachary Kincaid. 2016. Linear Arithmetic Satisfiability via Strategy Improvement.. In IJCAI, Vol. 16. 735–743.
- [25] Hao Gao and Qingke Zhang. 2024. Alpha evolution: An efficient evolutionary algorithm with evolution path adaptation and matrix generation. Engineering Applications of Artificial Intelligence 137 (2024), 109202.
- [26] David M Gay. 2015. The AMPL modeling language: An aid to formulating and solving optimization problems. In Numerical Analysis and Optimization: NAO-III, Muscat, Oman, January 2014. Springer, 95–116.
- [27] Google. [n. d.]. Jules. https://jules.google/
- [28] Lavanya Gupta, Saket Sharma, and Yiyun Zhao. 2024. Systematic Evaluation of Long-Context LLMs on Financial Concepts. arXiv preprint arXiv:2412.15386 (2024).
- [29] Zhiyuan He, Aashish Gottipati, Lili Qiu, Xufang Luo, Kenuo Xu, Yuqing Yang, and Francis Y Yan. 2024. Designing Network Algorithms via Large Language Models. In Proceedings of the 23rd ACM Workshop on Hot Topics in Networks. 205–212.
- [30] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM. 15–26.
- [31] Pantea Karimi, Solal Pirelli, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Beibin Li, Pooria Namyar, and Behnaz Arzani. 2024. Towards Safer Heuristics With XPlain. In Proceedings of the 23rd ACM Workshop on Hot Topics in Networks. 68–76.
- [32] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. IEEE Journal on

- Selected Areas in Communications 29, 9 (2011), 1765-1775.
- [33] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, et al. 2023. {OneWAN} is better than two: Unifying a split {WAN} architecture. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), 515–529.
- [34] Berthold Kröger. 1995. Guillotineable bin packing: A genetic approach. European Journal of Operational Research 84, 3 (1995), 645–661.
- [35] Aman Kumar and Priyanka Sharma. 2023. Open AI Codex: An Inevitable Future? *International Journal for Research in Applied Science* and Engineering Technology 11 (2023), 539–543.
- [36] Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. 2023. Large language models for supply chain optimization. arXiv preprint arXiv:2307.03875 (2023).
- [37] Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xianlong Zeng, Chenren Xu, et al. 2024. A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures. In Proceedings of the ACM SIGCOMM 2024 Conference. 228–243.
- [38] Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. 2024. Towards foundation models for mixed integer linear programming. arXiv preprint arXiv:2410.08288 (2024).
- [39] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. arXiv preprint arXiv:2404.02060 (2024).
- [40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. Science 378, 6624 (2022), 1092–1097.
- [41] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of heuristics: towards efficient automatic algorithm design using large language model. In Proceedings of the 41st International Conference on Machine Learning. 32201–32223.
- [42] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. arXiv preprint arXiv:2307.03172 (2023).
- [43] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient {GPU} cluster scheduling. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 289–304.
- [44] Nick McKeown. 2002. The iSLIP scheduling algorithm for inputqueued switches. IEEE/ACM transactions on networking (2002).
- [45] Minizic. [n. d.]. Minizic. https://www.minizinc.org/
- [46] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What do LLMs need to synthesize correct router configurations?. In Proceedings of the 22nd ACM Workshop on Hot Topics in Networks. 189–195.
- [47] Prasita Mukherjee and Benjamin Delaware. 2024. Towards Automated Verification of LLM-Synthesized C Programs. arXiv preprint arXiv:2410.14835 (2024).
- [48] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. 2022. Minding the gap between fast heuristics and their optimal counterparts. In Proceedings of the 21st ACM Workshop on Hot Topics in Networks (HotNets '22). Association for Computing Machinery.
- [49] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth

- Kandula. 2024. Finding adversarial inputs for heuristics using multilevel optimization. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). 927–949.
- [50] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. 2024. Solving {Max-Min} Fair Resource Allocations Quickly on Large Graphs. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24).
- [51] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving large-scale granular resource allocation problems efficiently with pop. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 521–537.
- [52] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. 2019. Gap: Generalizable approximate graph partitioning framework. arXiv preprint arXiv:1903.00614 (2019).
- [53] GitHub & OpenAI. 2021. GitHub Copilot: Your AI Pair Programmer. GitHub product announcement.
- [54] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In 29th International Conference on Software Engineering (ICSE'07). IEEE, 75–84.
- [55] Sagar Patel, Dongsu Han, Nina Narodystka, and Sangeetha Abdu Jyothi. 2024. Toward Trustworthy Learning-Enabled Systems with Concept-Based Explanations. In Proceedings of the 23rd ACM Workshop on Hot Topics in Networks. 60–67.
- [56] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. 2023. {DOTE}: Rethinking (Predictive) {WAN} Traffic Engineering. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), 1557–1581.
- [57] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. {CASSINI}:{Network-Aware} Job Scheduling in Machine Learning Clusters. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). 1403–1420.
- [58] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. 2015. Reactive synthesis from signal temporal logic specifications. In Proceedings of the 18th international conference on hybrid systems: Computation and control. 239–248.
- [59] Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. arXiv preprint arXiv:2405.06682 (2024).
- [60] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. 2024. Mathematical discoveries from program search with large language models. Nature 625, 7995 (2024), 468–475.
- [61] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). 593–612.
- [62] Sucha Supittayapornpong, Pooria Namyar, Mingyang Zhang, Minlan Yu, and Ramesh Govindan. 2022. Optimal Oblivious Routing for Structured Networks. In IEEE INFOCOM 2022 IEEE Conference on Computer Communications.
- [63] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neurosymbolic Loop Invariant Inference. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 406–417.
- [64] Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. 2025. HeurAgenix: Leveraging LLMs for Solving Complex Combinatorial Optimization Challenges. arXiv preprint arXiv:2506.15196

(2025).

- [65] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference. 179–193.
- [66] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2025. Efficient {Direct-Connect} Topologies for Collective Communications. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). 705–737.

A APPENDIX

In this section, we provide an in-depth examination of the various components of our design.

A.1 Suggester LLM

ROBUSTA uses a per-region Suggester LLM (Fig. 1) to provide suggestions for writing heuristics.

For each region \mathcal{R}_i , constructed by the Heuristic Analyzer (Sec. 3.2), we run a two-stage Suggester LLM pipeline (Fig. A.1): (1) a Pattern Analysis model that abstracts the failure patterns of the current heuristic within \mathcal{R}_i (denoted by \mathcal{H}_i^t), and (2) a Suggest Improvement that provides suggestions to improve heuristic, that the Heuristic Writer subsequently uses

Pattern Analysis LLM. It takes as input a balanced batch of *adversarial* and *normal* samples in \mathcal{R}_i (Fig. A.1), and processes them based on the prompt given in Fig. A.2. The prompt includes a *Problem Description* that is problem-dependent and is provided by the operators.

Suggest Improvement LLM. It takes as inputs the following and processes them based on the prompt in Fig. A.3:

- (1) Pattern analysis result from the pattern analysis LLM.
- (2) Short, human-readable *explanations* automatically produced by the Analyzer (§3.3) (e.g., "the heuristic routes $1 \rightarrow 3$ via $1 \rightarrow 2 \rightarrow 3$, but the optimal uses $1 \rightarrow 4 \rightarrow 5 \rightarrow 3$ to bypass a congested hub cut").
- (3) The code of the current heuristic \mathcal{H}_i^t that has generated the adversarial samples.

Why two LLMs? The decomposition mirrors what we observed empirically in Fig. 4: a single prompt to both (i) distill adversarial patterns and (ii) give suggestions to improve the heuristic is too much to ask the LLM. Separating the *failure patterns* from the *code suggestion* yields ideas that are (a) more focused, (b) targeted at the root cause of underperformance, and (c) less entangled with superficial features.

A.2 Heuristic Writer

Robusta's *Heuristic Writer* is a FunSearch-style loop that repeatedly asks an LLM to produce a *new* heuristic from k parents. In each step, it evaluates the new heuristic on a training batch (randomly selected from samples of the region), and keeps the new heuristic only if it improves the current worst-case gap on the batch or is diverse from the rest. We pass some of the worst adversarial samples from the parents and the suggestions to the LLM (Fig. A.4). If the new heuristic fails, we attempt to fix it for a small number of times (e.g., 3 times) using a fix prompt call to an LLM (Fig. A.5).

Algorithm 1 ROBUSTA'S Heuristic Writer

Require: I (Num islands), T (max iterations) Require: \mathcal{H}_0 (base heuristic to seed all islands) Require: $\mathcal{D}_{\text{train}}$ (training batch), $\mathcal{D}_{\text{held}}$ (held-out set) Require: m (worst-m samples to show the LLM) Require: Π_{mut} (mutation prompt template (Fig. A.4)) Require: Π_{fix} (fix prompt template (Fig. A.5)) Require: S (Suggestions generated by Suggester LLM) Require: Compile, Sim (compiler and simulator/evaluator)

Require: R_{fix} (max automatic fix rounds) **Require:** A (archive size/pruning budget) **Require:** p (patience/early-stop window)

Require: DIVERSE(\cdot) (diversity predicate of new code) **Ensure:** Best heuristic (\mathcal{H}^*) and its held-out performance.

1: Initialize *I* islands with the base heuristic \mathcal{H}_0

2: **for** $t \leftarrow 1$ to T **do**

3: for all islands j ∈ {1,..., I} do run in parallel
 4: Select best parents in island j by tournament on worst-case gap.

5: Build mutation prompt (parent code, worst-m samples, S) using Π_{mut} .

6: Query LLM to get candidate code; compile and simulate on $\mathcal{D}_{\mathrm{train}}.$

if candidate fails to compile/simulate then

8: Attempt up to R_{fix} automatic fix rounds.

9: **if** candidate improves worst-case gap (or ties but is DIVERSE) **then**

10: Add it to the island j.

11: Prune archives to size *A*, re-initiate islands, and checkpoint
 12: **if** no improvement for *p* iterations or worst-case gap == 0

then

13: break

7:

14: **return** best heuristic overall; report held-out performance on $\mathcal{D}_{\text{held}}$.

Runtime choice. Keeping k=1 (one parent \rightarrow one child) and a small number of islands keeps the search simple, cheap, and interpretable.

Fig.A.6 shows an improved heuristic for region 1 that reduces the train and held-out gap to zero.

A.3 Robusta's Ensemble Heuristic

Instead of using to a single global heuristic, Robusta uses a regional ensemble heuristic \mathcal{E} , where

$$\mathcal{E} = \{ (\mathcal{R}_i, \mathcal{H}_i^{\star}) \}_{i=1}^N,$$

where each region \mathcal{R}_i is a region of input space discovered by the Heuristic Analyzer (§3.2/§A.1), and \mathcal{H}_i^{\star} is the best heuristic *specialized* to that region, obtained by evolving the common base \mathcal{H}_0 with the Suggester + Writer loop (§1). For each input, ROBUSTA runs the corresponding heuristic for the region to which the input belongs.

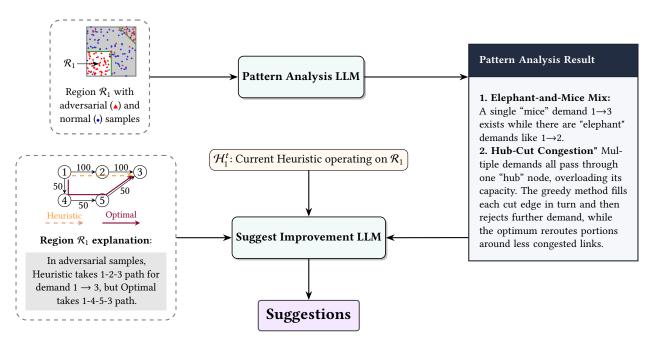


Figure A.1: Suggester LLM Pipeline. For each region \mathcal{R}_i , we run a two-stage pipeline: (1) Pattern Analysis LLM, which, given a balanced batch of adversarial and normal samples, abstracts the failure patterns of the current heuristic; and (2) Suggest Improvement LLM, which, using the pattern-analysis result, Analyzer-provided explanations (§3.3), and the code of heuristic, proposes concrete suggestions.

From \mathcal{H}_0 to \mathcal{H}_i^{\star} . Starting from the shared base heuristic \mathcal{H}_0 , we run, for each region \mathcal{R}_i , the two–stage Suggester (pattern analysis \rightarrow suggestions) followed by the Heuristic Writer (§A.2). The Writer produces a sequence $\mathcal{H}_0 \rightarrow \mathcal{H}_i^1 \rightarrow \ldots \rightarrow \mathcal{H}_i^{\star}$ that monotonically reduces the worst-case gap inside \mathcal{R}_i on training batch.

Pattern Analysis Prompt

Problem Description:

You are an expert in analyzing heuristic performance difference between the optimal solution and the heuristic solution in the Traffic Engineering problem. In this problem, we have a topology with nodes and directed edges with limited capacity. The inputs are the demands between the nodes. The goal is to route the maximum amount of traffic between the nodes in the network. Your final goal is to help design a better heuristic. Be concise and to the point.

Instructions

Please analyze these samples and identify patterns causing performance gaps between the heuristic and the optimal solution:

Tasks:

- 1. Compare the adversarial and non-adversarial sample sets (top num_samples each) and list patterns that correlate with a large heuristic-optimal gap.
- 2. For each pattern, provide a concise natural-language description.
- 3. Combine the findings with region description (green boundary).

Figure A.2: Pattern Analysis Prompt

Suggest Improvement Prompt

Problem Description:

You are an expert in analyzing heuristic performance difference between the optimal solution and the heuristic solution in the Traffic Engineering problem. In this problem, we have a topology with nodes and directed edges with limited capacity. The inputs are the demands between the nodes. The goal is to route the maximum amount of traffic between the nodes in the network. Your final goal is to help design a better heuristic. Be concise and to the point.

We have analyzed the performance of a heuristic and the optimal solution on a set of samples.

Pattern Analysis:

Pattern Analysis Result

Heuristic code:

```
\mathcal{H}_0: Base Heuristic
```

Explanations:

We also found out that the following decisions are the most likely to cause the gap:

Decision differences from Heuristic Analyzer

Task:

Please suggest ideas for improvements to the heuristic:

- 1. What modifications could prevent these gaps?
- 2. What additional network metrics should be considered?
- 3. What alternative routing strategies might work better?
- 4. How can we better handle congestion and load balancing?
- 5. Is there a way to run optimal on a subset of the problem? For example, a subset of demands or graph?
- 6. Propose ideas for improvements.

Your task is to list up to {n} concrete, different idea that would reduce the gap.

In order to do your task:

- 1. Examine the adversarial patterns and the decision differences.
- 2. From those patterns, extract up to one improvement idea likely to improve the heuristic.
- 3. For each idea, provide a detailed (at least 100 words), code-agnostic explanation and reasoning.
- Requirements -
- · Do not write code, only suggest ideas.
- Do not suggest ML approaches requiring lots of training data.
- · Provide a thorough explanation of each idea.
- · Explain so the reader can implement it themselves.
- Output Format -

Figure A.3: Suggest Improvement Prompt

Robusta's Heuristic Writer Prompt

Problem Description:

You are an expert in analyzing heuristic performance difference between the optimal solution and the heuristic solution in the Traffic Engineering problem. In this problem, we have a topology with nodes and directed edges with limited capacity. The inputs are the demands between the nodes. The goal is to route the maximum amount of traffic between the nodes in the network. Your final goal is to help design a better heuristic. Be concise and to the point.

Task

Your task is to design a new heuristic different than the Parent heuristics.

```
Parent Heuristic (1):
```

```
Here is a parent heuristic: python {code}
```

- Worst Performing Samples for the Parent Heuristic (1) -

The *Parent* heuristic performed poorly on the following samples compared to the optimal solution: Examples of adversarial ($_{\circ}$) samples for the parent (1): . . .

- Suggestions to improve the Parent Heuristic (1) -

You can use the following observations/suggestions to improve the parent heuristic:

```
Suggestions
```

... Add (k) Parents

- Requirements -

Based on the parent heuristics above, first analyze the pros and cons of each, and then design a new heuristic that performs better. You can use the suggestions to improve the parent heuristics if you want.

```
- Output Format -
[
    {
      "code": "def run_heuristic(...):\n # Your implementation here\n",
      "reasoning": "..."
    }
]
```

Figure A.4: Robusta's Heuristic Writer Mutation Prompt

Figure A.5: Fix prompt given to the LLM when a mutated heuristic fails to compile or simulate.

Figure A.6: Robusta's improved heuristic for region 1 (\mathcal{H}_1)

```
from typing import Dict, Any, List
from collections import defaultdict
def run_heuristic(graph: Graph) -> Dict[str, Any]:
   # Phase 1: Greedy routing on top-K shortest paths.
   demands = [d for d in graph.demands if d.demand > 0]
   for d in sorted(demands, key=lambda x: x.demand, reverse=True):
       remaining = d.demand
       try:
           paths = graph.get_shortest_paths_between_pair(d.source.name, d.target.name, k=3)
       except Exception:
           paths = graph.find_all_paths_between_pair(d.source.name, d.target.name)
       for p in paths:
           cap = p.min_remaining_capacity
           if cap <= 1e-9:
              continue
           flow = min(cap, remaining)
           if flow <= 1e-9:
              continue
           graph.add_flow_to_path(p, flow)
           remaining -= flow
           if remaining <= 1e-6:</pre>
              break
   # Phase 2: Local re-optimization around high-utilization edges.
   utilization = graph.get_edge_utilization() # {edge: utilization}
   hotspots = sorted([e for e, u in utilization.items() if u > 0.8],
                         key=lambda e: utilization[e], reverse=True)[:3]
   for edge in hotspots:
       # Build region around the hot edge
       region = {edge.source, edge.target}
       for e in graph.edges:
           if e.source in region or e.target in region:
              region.add(e.source)
              region.add(e.target)
       region_edges = [e for e in graph.edges if e.source in region and e.target in region]
       region_demands = [d for d in graph.demands if d.source in region and d.target in region]
       if not region_demands:
           continue
```

Figure A.7: Robusta's improved heuristic for region 1 (\mathcal{H}_1) (cont.)

```
# Construct subgraph
   name_map = {n.name: Node(n.name) for n in region}
   sub_edges = [
       Edge(name_map[e.source.name], name_map[e.target.name], e.capacity)
       for e in region_edges
   sub_demands = [
       Demand(name_map[d.source.name], name_map[d.target.name], d.demand)
       for d in region_demands
   subg = Graph(list(name_map.values()), sub_edges, sub_demands)
   # Reset flows in region
   for e in region_edges:
       e.flow = 0.0
   graph.active_paths = {
       p: f for p, f in graph.active_paths.items()
       if all(ed not in region_edges for ed in p.edges)
   }
   # Solve exact multi-commodity flow on the subgraph
   try:
       result = find_optimal_flows(subg)
       opt_graph = result.get("graph", subg)
       for path_sub, flow in opt_graph.active_paths.items():
           if flow <= 1e-9:
              continue
           real_edges = []
           for e_sub in path_sub.edges:
              real = graph.get_edge(e_sub.source.name, e_sub.target.name)
              if real is None:
                  real_edges = []
                  break
              real_edges.append(real)
           if real_edges:
              graph.add_flow_to_path(Path(real_edges), flow)
   except Exception:
       continue
# Compute and return metrics
total_met = sum(graph.active_paths.values())
total_unmet = sum(d.demand for d in graph.demands) - total_met
return {
   "total_met_demand": total_met,
   "total_unmet_demand": total_unmet,
   "graph": graph
}
```