Heuristic Analysis from Source Code via Symbolic-Guided Optimization

Pantea Karimi[1] Siva Kesava Reddy Kakarla[3] Ryan Beckett[3] Santiago Segarra[2]
Pooria Namyar[3] Mohammad Alizadeh[1] Behnaz Arzani[3]

[1]MIT [2]Rice University [3]Microsoft Research

Abstract — Large-scale systems rely on heuristics to tackle NP-hard problems such as traffic engineering, virtual machine placement, and packet scheduling. While these heuristics are efficient, they can exhibit severe performance gaps under certain workloads, which leads to outages or costly over-provisioning. This risk has motivated tools that attempt to find inputs that cause worst-case underperformance. But, to use these tools in practice, heuristic developers need to rewrite heuristics as formal mathematical models—a process that is time-consuming, error-prone, and excludes many real-world algorithms.

We introduce MetaEase, a practical general-domain analyzer that directly analyzes a heuristic's source code and eliminates the need for formal modeling. MetaEase combines code-aware input generation with guided search to uncover worst-case scenarios efficiently, even for heuristics with randomness (e.g., various traffic engineering schemes) or non-convex behavior (e.g., bin packing for virtual machine placement).

In most cases, across five problem domains, and eight heuristics, MetaEase matched or exceeded MetaOpt, a state-of-the-art optimization-based heuristic analyzer; in the remainder, it remained competitive and often ran faster. Against black-box optimization baselines, it won in 88% of settings and ranked in the top two otherwise. MetaEase analyzed Arrow [89], a recent networking heuristic that none of the state-of-the-art heuristic analyzers can analyze. We revealed previously unknown performance gaps in Arrow.

1 Introduction

Network researchers routinely face challenging optimization problems where it is infeasible to compute an exact solution (NP-hard or otherwise computationally expensive) [3, 5, 9, 25, 35, 52, 53, 57, 58, 69, 72, 77, 83, 86–88, 88]. For practical reasons, most systems rely on heuristics—non-optimal algorithms that trade accuracy for speed and scalability. Examples include "best-fit" or "first-fit" algorithms for VM placement [15], demand pinning for traffic engineering [57], and approximate algorithms for packet scheduling [7–9, 85]. However, heuristics can have severe failure modes that degrade performance and impact user experience [9, 12, 57]. For instance, Microsoft previously employed a traffic engineering heuristic in its wide-area network that, for certain inputs, underperforms the optimal solution by 30% [57]. To mitigate this impact, they must either over-provision the network by 30%, drop 30% of the demand, or *detect such cases and switch to an alternative heuristic* [57].

Recent performance analyzers [9, 12, 33, 57] compare a heuristic's performance against an (often optimal) benchmark, and find inputs that produce the largest performance gap to enable developers to anticipate and mitigate risks. These tools reveal problematic inputs, guide heuristic refinement, and support input-aware switching among heuristics with complementary failure modes (see [57]). They solve:

$$I^* = \arg\max_{I \in \mathbb{R}^n} \text{Benchmark}(I) - \text{Heuristic}(I) \qquad (1)$$

to find the input(s) I^* that maximize the performance gap between the heuristic and the benchmark for a specific instance of the problem.

For example, in traffic engineering, the input I represents traffic demand, while Benchmark(I) and Heuristic(I) denote the total flow the optimal solution and the heuristic route, respectively. The objective is to find the demand I^* that minimizes the traffic the heuristic routes relative to the optimal. Throughout this discussion, the input I (e.g., demand in traffic engineering) is an unspecified variable in this problem.

Existing performance analyzers that compute I^* have two main limitations. First, they are difficult to use. To apply them, we need to express the heuristic in specialized mathematical forms, such as a convex optimization model [57], feasibility or satisfiability constraints [33, 57], or a network flow model [43, 44]. These tools demand substantial expertise in formal methods or optimization theory; it is hard to even model a simple traffic engineering heuristic (see §2). We need to re-model after each change to the heuristic.

Second, existing analyzers support a limited range of heuristics. MetaOpt [57] only analyzes heuristics that are expressed as convex optimization or a set of feasibility constraints. FPerf [9], CCAC [12], and Virelay [33] are custom-designed for specific domains (queue management, congestion control, and scheduling, respectively). Many common classes of heuristics, such as ML-based heuristics, fall outside their scope.

Tool	No Heuristic Optimization/Modeling	Structure-aware Search	General Domain
MetaEase	✓	✓	✓
Black-box Search	✓	✗	✓
MetaOpt [57]	✗	✓	✓
Virelay [33]	✗	✓	✗
XPlain [43]	✗	✓	✓
FPerf [9]	✗	✓	✗

Table 1: We compare MetaEase with prior analyzers. Unlike black-box methods, MetaEase leverages heuristic and optimal structure to guide the search, and unlike model-based approaches, it does not require a mathematical model of the heuristic. Here, we refer to tools that use information from the heuristic or the benchmark to guide their exploration (as opposed to treating them as black box) as "structure aware".

While most heuristic developers have little experience in formal analysis to model heuristics mathematically, we observe that *implementations* of heuristics are often readily available – after all, one must execute the heuristic to use it! Similarly, benchmark models are usually available and rarely pose a problem[1]. Motivated by this, our goal is to directly find the inputs that make a heuristic *implementation* underperform its benchmark as much as possible, with only access to its source code.

We could treat the heuristic as a black box when we solve the problem using off-the-shelf techniques such as sample-based gradient ascent [71], Bayesian optimization [18], hill climbing [27], and simulated annealing [46]. These methods sample inputs $I \in \mathbb{R}^n$, compute the performance gap for each (i.e., Benchmark(I) − Heuristic(I)), and iteratively move toward worse-performing regions. However, such a search quickly becomes impractical: for each sample, we must solve the benchmark — often an NP-hard or otherwise costly optimization — to evaluate the outcome and compute the next search direction. Worse, the search frequently gets stuck in local optima and requires multiple runs from different starting points. This approach scales poorly, especially on large problem instances (see Fig. 15 and §6.1).

We present MetaEase, a performance analyzer that enables heuristic developers to directly evaluate a heuristic *implementation* and overcomes the limitations of black-box search. MetaEase takes as input (1) the heuristic implementation (written in C) and (2) a benchmark, specified as an optimization model whose solution represents the best achievable performance for the problem. Developers may trade off speed and provide an implementation of the benchmark instead. MetaEase finds the input I^* that maximizes the performance gap between the benchmark and the heuristic. Developers can optionally impose constraints to restrict the search (e.g., to consider only likely inputs). Tab. 1 shows how MetaEase improves upon the state of the art. Unlike prior analyzers [9, 33, 43, 57], MetaEase does not require a mathematical model of the heuristic. It relies solely on a benchmark formulation—typically well studied and easy to specify for most problems. Crucially, users define the benchmark once per domain (e.g., multi-commodity flow for traffic engineering), while heuristics change frequently and vary widely. This separation enables MetaEase to analyze diverse heuristics without repeated modeling effort.

MetaEase addresses black-box search limitations: on the high-level MetaEase runs gradient ascent [2] search on Eq. 1, but unlike black-box approaches, it uses the heuristic's structure to guide its search and has a mechanism to reduce expensive benchmark calls. It exploits the linearity of differentiation in Eq. 1 and expresses the ascent direction as $\nabla\text{Benchmark}(I) - \nabla\text{Heuristic}(I)$, which allows it to compute the two terms separately. This gradient-based approach provides search progress for Eq. 1 without repeatedly solving the benchmark (§8.2).

For the heuristic term, we fit a smooth surrogate (a Gaussian process [75]) to Heuristic(I) within a small neighborhood of input I and differentiate it to estimate $\nabla\text{Heuristic}(I)$. As the search progresses, MetaEase updates the surrogate to approximate Heuristic(I) locally. We also show this surrogate-guided direction is more efficient than black-box approaches (Fig. 15).

By itself, when we use it to solve non-convex problems such as ours, a simple gradient-based search such as this will take a long time to converge. Even when it does (because the problem is non-convex), it may converge to local optima. Our key contribution is to exploit the structure of the problem we are trying to solve in order to mitigate these issues.

MetaEase's second key idea is one of the two key innovations we introduce to enable this: we use symbolic execution tools for static analysis of the heuristic's code to select initial seed points for the search. Black-box methods often start from random inputs, many of which fall into the same "equivalence class" where the heuristic behaves identically. Instead, MetaEase uses KLEE [19], a robust and well-known symbolic execution tool [14, 20], to extract inputs that exercise different code paths. We then run gradient ascent from each seed in parallel and report the maximum performance gap across all searches. Our results show that MetaEase finds gaps up to 44× larger than those from random initialization (§8.1).

To our knowledge, MetaEase is the first system that lets developers analyze their *existing implementations (as code)* of deployed heuristics and quantify when and by how much they underperform relative to a benchmark. MetaEase can analyze heuristic implementation, and we show that it supports non-convex and probabilistic heuristics, which prior work in this space cannot analyze. We used MetaEase to analyze heuristics from five common problem domains in networking and distributed systems (vector bin packing, WAN traffic engineering, Arrow [89], knapsack, and maximum weight matching). MetaEase *matched or exceeded* the performance gap the state-of-the-art found (which in most cases solves the problem optimally) in 67% of our experiments, was within 1−15% for

[1] For any domain, we only need to model the benchmark once and can reuse it across heuristics.

[2] Gradient ascent iteratively updates $x \leftarrow x + \eta \nabla f(x)$ to maximize f.

30% of cases, and within ∼35% in the rest. MetaEase gives up optimality guarantees (it no longer can guarantee that it finds the input that causes the worst possible underperformance) to provide ease-of-use. We have open-sourced MetaEase for the community at https://github.com/microsoft/MetaEase.

2 Motivation and MetaEase Overview

Our goal is to find inputs that maximize the performance gap between the heuristic and the benchmark (Eq. 1). Several tools already address this problem. We explain how existing heuristic analyzers operate and use an example to show their limitations. In this discussion, we focus on tools that support a broad range of heuristics and exclude those limited to specific domains [9, 10, 12, 33] (we describe these works in detail in the related work).

2.1 How existing methods solve the problem

Heuristic analyzers. We know of two general-purpose heuristic analyzers: MetaOpt [57] and XPlain [43]. MetaOpt solves a bi-level optimization problem and takes as input a mathematical model of the heuristic (and the benchmark), provided either as an optimization problem or as a set of constraints — it supports convex or feasibility heuristics. XPlain [43] lets users model heuristics in a network-flow-based domain-specific language and compiles the model into an optimization problem that MetaOpt supports. Virelay [33] is a scheduling heuristic analyzer that uses formal methods: it encodes heuristic performance as satisfiability constraints (generalizing feasibility constraints) and uses an SMT solver (Z3 [28]) to find instances where the heuristic underperforms. Virelay's SMT-based encoding may be adaptable to other domains.

Black-box methods. Black-box methods (e.g., multi-start hill climbing [27], simulated annealing [46], sample-based gradient [17]) treat both benchmark and heuristic as black boxes. They do not require a mathematical model but are often sample-inefficient, seed-sensitive, and stall in local optima. We discuss the challenges of these approaches in §2.3.

2.2 Why existing heuristic analyzers are hard to use

Consider a simple traffic engineering heuristic that operates as follows (Code 1): (1) route the largest 20% of demands optimally, ignoring the rest; (2) freeze these allocations and construct a residual graph by adjusting capacities based on the previous step; and (3) allocate the remaining demands optimally on this residual graph. The goal of this heuristic is to reduce the number of variables in the traffic engineering optimization problem, which enables faster solutions.

One might expect that it is straightforward to use existing tools to analyze this heuristic since MetaOpt and XPlain support the underlying convex traffic engineering optimization. The heuristic essentially solves two optimization instances: first for the largest 20% of demands, then for the remainder. Indeed, MetaOpt and XPlain model a similar heuristic called "demand pinning," which fixes small demands to the shortest paths before it routes the rest.

```
Demands SelectTop20Percent(Demands D) {
    sort_desc(D);
    int k = max(1, 0.2 * size(D));
    return top_k(D, k);
}
```

(a) Expressing top-20% selection in code

Input: $D = \{d_k\}$

1: $\tau \leftarrow \lfloor 0.2|D| \rfloor$

2: $r_k \leftarrow \mathtt{Rank}(d_k, [d_i]_{i \in D})$

3: $c_k \leftarrow \mathtt{IsLeq}(r_k, \tau) \quad (c_k \in \{0,1\})$

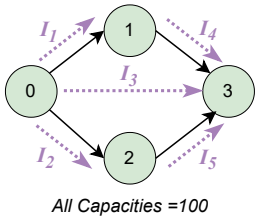
4: $\sum_k c_k = \tau$

5: $d_k^{\text{crit}} \leftarrow \mathtt{Multiplication}(c_k, d_k)$

6: $d_k^{\text{non}} \leftarrow \mathtt{Multiplication}(1 - c_k, d_k)$

(b) Expressing top-20% selection in MetaOpt using its helper functions

Figure 1: How we can express the "top-20% of demands" as (a) code, and (b) in MetaOpt. MetaOpt's encoding requires much more expertise and optimization background.



(a) Simple topology.

Seed	1	2	3	4	5
Initial	$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \end{bmatrix}$	$\begin{bmatrix} 70 \\ 270 \\ 130 \\ 180 \\ 240 \end{bmatrix}$	$\begin{bmatrix} 20 \\ 280 \\ 340 \\ 4 \\ 60 \end{bmatrix}$	$\begin{bmatrix} 170 \\ 340 \\ 60 \\ 360 \\ 140 \end{bmatrix}$	$\begin{bmatrix} 230 \\ 380 \\ 280 \\ 250 \\ 320 \end{bmatrix}$	$* \begin{bmatrix} 1 \\ 1 \\ 200 \\ 1 \\ 1 \end{bmatrix}$
Final Gap	0.0	0.0	0.0	0.0	**398.2**	
Time (min)	20	20	20	20	**5**	

(b) Hill-Climbing results.

Figure 2: Black-box approaches struggle to find large performance gaps. Hill-Climbing stalls under random initialization, yielding small gaps and long runtimes, and succeeds only with carefully selected inputs (*).

It is trivial to select the top 20% of demands in code (Fig. 1a). In contrast, it is hard to do so in MetaOpt because demands are variables within the verifier (the input vector $\boldsymbol{I}$ in Eq. 1), which makes it impossible to order them explicitly during problem formulation. To find the top 20%, we must encode the sorting process itself—a task that is difficult to model. We show how to achieve this in MetaOpt (Fig. 1b). Encoding sorting is inherently complex (the exact details are not critical here). A further complication is that the two optimizations must execute sequentially, which is difficult to express (see App. A for details). These challenges are not unique to our setting; similar issues arise in max–min fair traffic engineering methods [58], which also require explicit modeling of sorting-based allocations.

This example shows why it is hard to model even a simple heuristic as an optimization. The MetaOpt authors employ big-M techniques and methods to "convexify" optimizations for such cases. These techniques are difficult for non-experts to devise and apply. Similar challenges occur when one expresses the heuristic as SMT constraints in Virelay.

In our experience, encoding this simple heuristic in MetaOpt required roughly two full days of effort by an author familiar with both MetaOpt and the TE domain, even with AI assistance (see Fig. 19); whereas implementing it as code took less than five minutes. Because developers must repeat this modeling effort for *every new or modified heuristic*, the re-modeling effort quickly accumulates.

2.3 Why Black-Box search algorithms do not work

A naive approach is to treat the problem as a black-box search: we run both the benchmark and the heuristic on selected input instances and navigate the input space to find larger performance gaps. But the benchmark/optimal solution for most problems we study is slow—heuristics often approximate NP-hard problems—so each instance is expensive to evaluate. This approach also overlooks information in the heuristic's source code and the benchmark's mathematical representation.

Consider a highly simplified traffic engineering (TE) example (Fig. 2). We ran a multi-start hill-climbing [27] search. With random demand initializations, the search stalls (Fig. 2) and fails to uncover a meaningful performance gap. Only after we manually inject a specially crafted input (marked as * in the table) does it discover a meaningful gap, and even then each instance takes minutes to evaluate.

These issues are common in black-box algorithms: they often converge to poor local optima, are highly sensitive to initialization, and run slowly (see also Fig. 9 and §7).

2.4 MetaEase overview

MetaEase strikes a balance between black-box methods and model-based analyzers. Similar to black-box methods, it does not require a mathematical formulation of the heuristic, but it uses the heuristic's code structure to guide the search. We show MetaEase's high-level design in Fig. 3.

MetaEase enables developers to analyze any heuristic directly from its *code*. It requires a mathematical model only for the benchmark. This burden is minimal because we need to model each problem domain once and MetaEase already models many. These benchmarks are often fixed, canonical, reusable, and well-optimized. For example, Arrow [89] models the optimal cross-layer Optical–IP problem and MetaOpt [57] models the optimal vector bin packing and packet scheduling solutions. Users may provide a benchmark implementation, though this may reduce efficiency.

MetaEase finds inputs that cause the heuristic to underperform relative to the benchmark. It (1) uses the linear separability of the performance gap objective (Eq. 1) to do gradient ascent, and (2) smartly generates seeds from static analysis of the heuristic's code to improve search efficiency. We further introduce novel techniques that allow us to navigate non-differentiable regions in (§3.4).

(1) Gradient ascent: The Gradient Ascent Module (Fig. 3) exploits the separability of $\text{Gap}(I) = \text{Benchmark}(I) - \text{Heuristic}(I)$ to compute gradients for the benchmark and heuristic independently. When the benchmark is provided as an optimization formulation, we derive the Lagrangian (§3.1) and compute its gradient directly, avoiding repeated (costly) optimal runs. This approach scales better and outperforms black-box methods (see §6.1 and §8.2). If the optimization formulation is unavailable, MetaEase uses the same mechanism that it uses for the heuristic to approximate the benchmark gradient.

To estimate the heuristic's gradient, we adopt a surrogate-based approach [24, 30, 32, 40, 70], suited for unknown or non-differentiable objectives. At each step, MetaEase fits a Gaussian process to a local region and uses it to compute the heuristic's gradient (Fig. 5, §3.2). When gradient ascent moves beyond this region, we retrain the process. Gaussian processes approximate any function with sufficient data [42, 54] and provide closed-form gradients [54].

(2) Seed generation: The seed generator module (Fig. 3) mitigates poor local optima. Unlike black-box methods that rely on random restarts, we symbolically analyze the heuristic's code using KLEE [19]. KLEE explores branch structures (e.g., if/else conditions) and returns concrete inputs that traverse distinct execution paths (Fig. 4). Each path defines an input region where the heuristic follows the same control flow, and forms *equivalence classes* of distinct behaviors.

MetaEase performs "path-aware" gradient ascent within each equivalence class and reports the maximum performance gap across classes. This approach does not have to cross non-differentiable regions of the heuristic's code because it treats path conditions as boundaries (§3.4).

In the end, MetaEase reports the true gap by executing both the benchmark and the heuristic on the discovered final input, independent of the Lagrangian-based objective used to guide the search.

3 MetaEase Approach

MetaEase uses the heuristic implementation to guide the search, which allows it to handle non-convex and non-differentiable heuristics—cases that prior methods cannot address. In this section, we show how MetaEase overcomes the key challenges outlined earlier. MetaEase:

- does not solve the costly benchmark repeatedly when it maximizes the performance gap (§3.1);
- uses a local surrogate model to estimate the gradients of arbitrary heuristics (§3.2);
- selects informed seeds for gradient-based search to prevent poor local optima (§3.3); and
- ensures stable progress even in the presence of non-differentiabilities (§3.4).

Some of our ideas are inherently novel (e.g., symbolic execution and path-based gradients to identify equivalence classes and help avoid issues with non-differentiable regions in the input space). Others are well-known in the optimization literature [17, 75], but we integrate them into MetaEase in way that allows it to function as a general-purpose heuristic analyzer. We aim to maximize $\text{Gap}(I)$ without repeatedly solving the

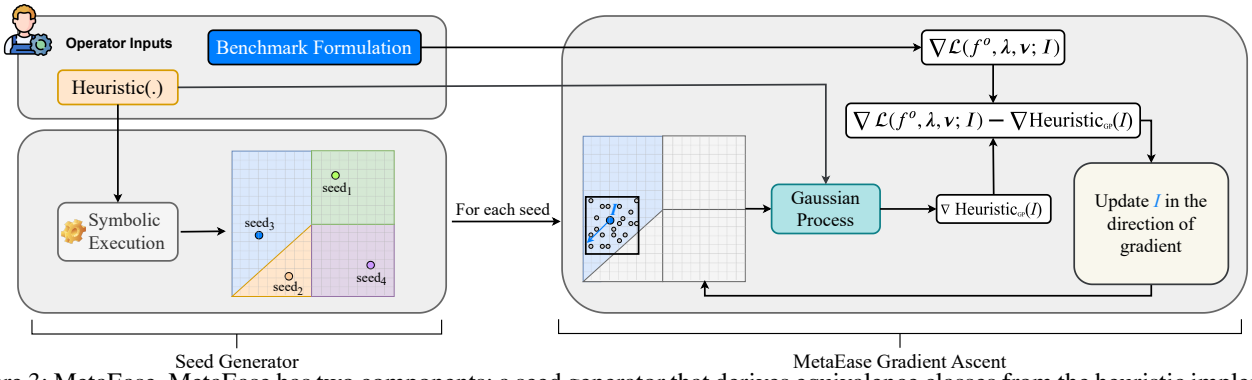


Figure 3: MetaEase. MetaEase has two components: a seed generator that derives equivalence classes from the heuristic implementation through symbolic execution, and a gradient ascent process that uses the benchmark formulation to compute the gradient of the Lagrangian and the heuristic code for an approximate gradient to guide the search.

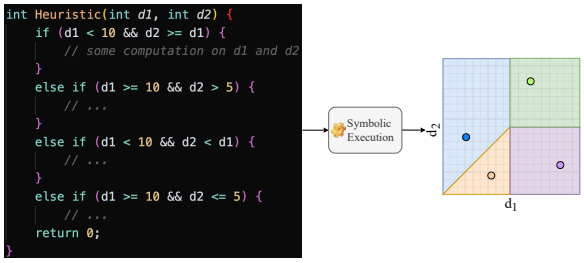


Figure 4: KLEE [19] symbolically explores the heuristic code and generates inputs for distinct *code paths*. Each path represents an equivalence class of inputs sharing the same control flow; here, KLEE produces one input per branch condition.

costly benchmark. We use gradient ascent. Since gradients are linear, we decompose $\nabla\big(\text{Benchmark}(I)-\text{Heuristic}(I)\big)$ as $\nabla\text{Benchmark}(I)-\nabla\text{Heuristic}(I)$, and estimate each term separately. We start with the benchmark.

3.1 How MetaEase estimates ∇Benchmark(I)

We use an established Lagrangian-based method to estimate the gradient [17, 26, 55]. We describe the technique below.

Benchmark as an optimization. The benchmark is itself, in general, an optimization:

$$\text{Benchmark}(I)=\max_{x^o} f_{\text{B}}(x^o;I) \tag{2}$$
$$\text{s.t. } g_i(x^o;I)\geq 0,\ i=1,\ldots,m, \quad h_j(x^o;I)=0,\ j=1,\ldots,p.$$

Here, x^o are the benchmark's decision variables (e.g., how much flow to allocate on each path in traffic engineering), g_i are the m inequality constraints (e.g., the capacity constraints), and h_j are the p equality constraints (e.g., flow conservation). The *variables* I are inputs to the benchmark problem (e.g., the demand matrix). Intuitively, the benchmark defines the optimal outcome for a given input I—for example, in traffic engineering, it computes the maximum total flow that we can route.

Step 1. Lagrangian. The gradient $\nabla\text{Benchmark}(I)$ captures the change in the benchmark's *optimal value* under small perturbations of I. We would have to re-solve the benchmark for each perturbation of I to directly compute it, which is costly. Instead, we use the benchmark's *Lagrangian*, which combines the objective and the constraints:

$$\mathcal{L}(x^o,\lambda,\nu;I)\triangleq f_{\text{B}}(x^o;I)+\sum_{i=1}^{m}\lambda_i g_i(x^o;I)+\sum_{j=1}^{p}\nu_j h_j(x^o;I) \tag{3}$$

Here, $\lambda_i \geq 0$ are dual variables for the m inequality constraints g_i, and ν_j are dual variables for the p equality constraints h_j. We can approximate how the benchmark's optimal objective changes with the change of I when we analyze the gradients of $\mathcal{L}$ with respect to these variables. This allows us to estimate $\nabla\text{Benchmark}(I)$ without repeatedly solving the full optimization.

Step 2. Duality. The Lagrangian approximates the benchmark's optimal value. By *weak duality* [17], we find an upper bound on the true benchmark value for input I if we maximize the Lagrangian over the benchmark's decision variables x^o:

$$\text{Benchmark}(I)\leq \sup_{x^o}\mathcal{L}(x^o,\lambda,\nu;I), \quad \lambda\geq 0.$$

Here sup (supremum) simply means the largest value of $\mathcal{L}$ over all possible choices of x^o.

When the benchmark is convex, *strong duality* [17] tells us something stronger: we can exactly recover the benchmark value if we optimize in two stages— we first take the maximum over x^o and then the minimum over the dual variables λ,ν:

$$\text{Benchmark}(I)=\min_{\nu,\lambda\geq 0}\sup_{x^o}\mathcal{L}(x^o,\lambda,\nu;I).$$

This "max–min" structure is called a *saddle-point*: one set of variables (x^o) aims to maximize the objective, while the dual variables aim to minimize it. For clarity, we denote ascent variables in green and descent variables in red.

Step 3. Compute the gradient. Our goal is to estimate $\nabla\text{Benchmark}(I)$—the sensitivity of the benchmark's optimal value to changes in I. The saddle-point form is useful because the Lagrangian already encodes how both the objective and the constraints respond to I. Empirically, even for non-convex benchmarks, we can find a reliable direction for

∇Benchmark(I) if we follow the gradients of the saddle-point expression.

We treat the saddle-point as a joint optimization and update each variable according to its role:

- maximize over variables x^o (the benchmark's decisions),
- minimize over the dual variables λ, ν (the penalties on constraints), and
- ascend in the input I (the parameter we care about).

In practice, we take an ascend step for x^o and I, and descend for λ and ν. We approximate ∇Benchmark(I) through these coordinated updates, which allows us to not re-solve the benchmark optimization in each step.

When a benchmark's optimization model is unavailable, developers can provide its implementation. In this case, to compute the benchmark's gradient, we use the same surrogate-based method as for the heuristic. We describe how we estimate ∇Heuristic(I) next.

3.2 How MetaEase estimates ∇Heuristic(I)

Now that we can estimate ∇Benchmark(I), we also need ∇Heuristic(I). Unlike the benchmark, we do not have a mathematical formulation of the heuristic, only its implementation. We therefore use a surrogate-based approach [24,30,32,40,70]. The idea is to build a *local surrogate* around the current input $I \in \mathbb{R}^n$ and use the surrogate's gradient as an estimate of the heuristic's gradient (Fig. 5).

We define a hypercube (the Δ-neighborhood) centered at I with side length $\Delta > 0$. We draw N samples from this hypercube, run the heuristic implementation on each sample, and fit a Gaussian process (GP) model to the results. We then use the GP's closed-form gradient to approximate ∇Heuristic(I) and update I along that direction. As the search progresses, MetaEase refits the GP whenever I leaves its current hypercube to ensure the surrogate reflects local behavior.

Gaussian processes are a natural fit because they provide smooth, differentiable surrogates with analytically computable gradients [75] (App. B). Intuitively, the GP fits a smooth curve through the heuristic's local outputs, which lets us differentiate the surrogate rather than the heuristic itself. This yields efficient gradient estimates and reduces runtime. Compared with finite-difference methods, where we would have to evaluate the heuristic along all n coordinate directions[3], MetaEase requires far fewer heuristic evaluations (§8.3).

3.3 How MetaEase avoids bad local optima

When an optimization problem is non-convex, gradient-based methods can get trapped in local optima [17]. Tools such as MetaOpt use mixed-integer solvers to solve the optimization directly and largely avoid this issue. MetaOpt supports only convex heuristics or those expressible as feasibility problems. In contrast, MetaEase accepts *any* heuristic implementation, which may be non-convex or otherwise ill-behaved. Therefore,

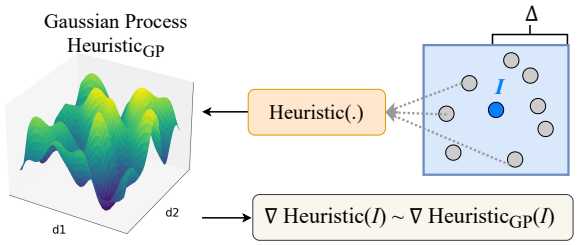


Figure 5: MetaEase fits a local Gaussian-process (GP) surrogate to heuristic evaluations near the current input and uses GP's analytical gradient to estimate gradient of heuristic.

we need a strategy for cases where gradient-based search converges to poor local optima.

A common mitigation is random multi-start, which restarts the search from multiple random initial seeds [46]. Although this approach improves coverage of the input space, it does not guarantee that we find an optimal or near-optimal solution. As problem size increases, the number of starting seeds we need grows substantially [46].

MetaEase uses the heuristic code to find search seeds. To increase the likelihood that MetaEase finds larger gaps than random seeding, we use the heuristic's *implementation*. We identify *equivalence classes* of inputs that trigger the same behavior in the heuristic and use a representative from each class as seeds. We then run the gradient search we described earlier within each class and retain the best gap across classes.

MetaEase uses KLEE to find equivalence classes. We use KLEE [19], which is a mature, robust, and well-known symbolic execution tool [2,14,19,20], to symbolically execute the heuristic and produce representative inputs for distinct code-paths. [4] Symbolic execution is a well-studied concept in formal methods that is used to automatically derive test cases from a target program's source code. For MetaEase, the test inputs serve as the seed values for its search. In most cases, each code-path corresponds to a set of distinct "choices" the heuristic makes on a given input. This means the heuristic treats inputs that follow the same code-path in the same way — the property we seek in an equivalence class. KLEE returns a set of input seeds $I \in$ KLEE(Heuristic) where each input has a different *code-path signature*, h. We run gradient ascent in parallel across multiple equivalence classes starting from KLEE points. In some cases, KLEE discovers only a single code-path for a heuristic. This indicates that symbolic execution has not exposed any meaningful branching behavior; in such cases, we revert to random multi-start.

3.4 How MetaEase avoids heuristic non-differentiability

Many heuristics contain conditional logic. For example, the demand pinning heuristic we studied in MetaOpt pins small

[3] Finite differences estimate $\nabla f(I)$ by computing $f(I)$ and $f(I+\delta e_j)$ for each dimension $j = 1,\ldots,n$.

[4] For clarity, we use the terms code-path and equivalence class interchangeably in the remainder of this paper.

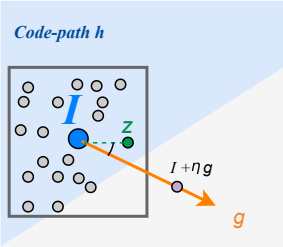


Figure 6: One step of gradient ascent in MetaEase. We move along the estimated gap gradient ($\eta \cdot g$); if the step remains within the same code path, we take it; otherwise, we project to teh most-aligned point z that preserves the path. See Algo. 1.

demands before it optimally routes the remainder; the heuristic in §2 sorts demands and routes the top 20% first; and the first-fit decreasing heuristic for bin packing sorts items and then places them from smallest to largest. Such branching decisions create highly non-convex and non-differentiable regions in the gap function. In these regions, the gradient can assume extreme values, which causes gradient ascent to repeatedly overshoot or undershoot the true optimum (§8.2).

To mitigate this, MetaEase restricts its search *within* an equivalence class of inputs, i.e., inputs that follow the same code path in the heuristic. Empirically, we find that within each equivalence class the heuristic is much smoother, which makes gradient ascent more stable and effective. While this is not guaranteed to always find the global optimum, in practice it consistently leads to higher gaps.

Fig. 6 and Algo. 1 show how MetaEase performs a *path-aware* gradient step. After it computes the benchmark gradient (§3.1) and use the GP surrogate to compute the heuristic gradient (§3.2), MetaEase first attempts a regular gradient ascent step. If the step remains in the same code path, it is taken directly. If instead it crosses into a new path, MetaEase projects the update back into the current equivalence class: from the GP training samples it selects the point whose direction is most aligned with the gradient and moves there instead.

This path-aware technique keeps the search within a single equivalence class, and avoids large jumps across non-differentiable regions. In practice, this allows MetaEase to explore both sides of non-differentiable boundaries separately and reliably uncover higher performance gaps (§8.2). For further details, see App. C.

4 Optional Optimizations

MetaEase can analyze any heuristic implementation that operates on numerical inputs. In this section, we discuss optional optimizations that improve MetaEase's runtime and memory efficiency for larger problem instances. Users can disable these optimizations to run the vanilla MetaEase implementation, which still analyzes the heuristic and computes the worst-case gap, though with higher time and memory costs. We evaluate the benefits of these optimizations in §8.

Algorithm 1: Path-aware gradient step in MetaEase

Input: Block size Δ, samples N, step size η

$h \leftarrow \text{path}(I)$ // current code path
$\mathbf{X} \leftarrow \{u \in \text{SAMPLEBOX}(I,\Delta) \mid \text{path}(u) = h\}$
$\mathbf{Y} \leftarrow [\text{Heuristic}(x) \mid x \in \mathbf{X}]$
$\text{Heuristic}_{GP} \leftarrow \text{FITGP}(\mathbf{X},\mathbf{Y})$
Compute $g_b \leftarrow \nabla\text{Benchmark}(I)$
Compute $g_h \leftarrow \nabla\text{Heuristic}_{GP}(I)$
$g \leftarrow g_b - g_h$
$\tilde{I} \leftarrow I + \eta \cdot g$
if $\text{path}(\tilde{I}) = h$ **then**
  $I \leftarrow \tilde{I}$ // stay in same path
else
  Pick $z \in \mathbf{X}$ most aligned with g
  $I \leftarrow z$ // project back into path

4.1 Projected gradients

When we study large problem instances, we have to look at higher dimensional inputs. This increases the runtime of two MetaEase components: (1) KLEE [19] may take longer to run; (2) we may need more samples to train the Gaussian Process.

To maintain low runtime despite these increases, we use the technique of *projected gradients* [21]. We select a subset of K input dimensions and *freeze* the remaining dimensions. We then apply MetaEase in the subspace defined by the selected K dimensions. Afterward, we unfreeze these values and choose a new subset of dimensions to continue the search. This approach mitigates the slowdowns in (1) and (2) but may affect the quality of the resulting solution.

Developers can use this feature to quickly estimate the heuristic's worst-case performance. We discuss runtime optimizations combined with this approach in App. F.

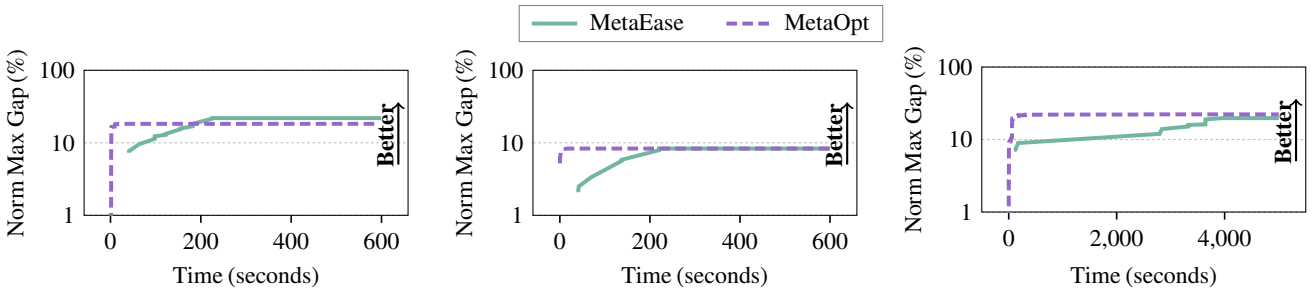
4.2 Domain customization

As in most verification problems, domain knowledge improves runtime. Users can input additional *hints* to help MetaEase. Hints are optional, domain-specific assumptions that users provide to symbolic execution. These hints help prune "obviously" uninteresting (easy) inputs and improve scalability. Hints do not affect correctness. MetaEase works without hints, but KLEE may explore many trivial cases and become slower without them. We translate hints to constraints before we use them. Developers can specify two categories of hints:

Range constraints: intervals that restrict the input range;

Hardness predicates: conditions over the input space that developers believe cause the heuristic to underperform.

Consider our earlier example: developers know that demands stressing certain edge capacities in the graph are critical—if no demand overloads an edge, we can satisfy all demands without difficulty. Developers can encode this condition as a predicate in MetaEase (see App. E for an example).



(a) Abilene (b) SWAN (c) B4

Figure 7: MetaEase and MetaOpt exhibit similar behavior and reach comparable performance gaps over time in Demand Pinning for production topologies.

5 Evaluation Overview

MetaEase applies across domains such as wide-area and optical-IP cross-layer traffic engineering, knapsack problems, vector bin packing, and maximum weight matching. We evaluate MetaEase on deterministic (greedy, sort-based, and conditional), randomized, and DNN-based heuristics. Our results show that MetaEase consistently ranks among the top two approaches. In 14 experiments, it matches the worst-case gap of MetaOpt, which computes the exact optimum when run to completion. In six additional experiments, it achieves at least 85–99% of MetaOpt's performance and outperforms all black-box methods in 22 out of 25 experiments.

5.1 Experiment setup

Implementation. We use *KLEE* [19] to compute heuristic equivalence classes and *OR-tools* [64] to solve benchmark optimization problems. We used each problem's optimal solution as a benchmark. To train Gaussian process surrogates for a given input, we uniformly sample $N = 100$ points from a hypercube centered on that input. We set the hypercube size to $\Delta = 1\%$ of the input space (§3.2)[5].

Metrics. We define the *normalized gap* as the performance difference between the benchmark and the heuristic, divided by the maximum value either can obtain on any input. For instance, in the case of traffic engineering, the normalized gap corresponds to the difference in total flow, divided by the sum of link capacities in the network. For each method, we report the *maximum normalized gap* they discovered. We also track runtime. All experiments run on an Intel Xeon E5-2630 v4 CPU @ 2.20 GHz with 24 cores and 23 GB of memory, and use all available threads. See App. G for detailed MetaEase runtimes.

6 MetaEase vs MetaOpt

We evaluate the relative performance gap MetaEase finds compared to MetaOpt. MetaOpt is an open-source optimization-based heuristic analyzer our team previously developed and supports several heuristics (see Tab. 2). For heuristics that are difficult or impossible to model in MetaOpt, we compare against black-box baseline methods. For cases in which MetaOpt requires additional time to converge (and may exceed MetaEase 's time budget), we report those extended results in §G.1.

	Benchmark	Heuristic
TE	MaxFlow [49, 57]	DP [48] (conditional) POP [61] (random) Alg in §2 (sorting) DOTE [65] (DNN)
VBP	Optimal [82]	FFD [63]
Knapsack	Optimal [82]	Sort-based greedy [31]
Optical–IP TE	Optimal [89]	Arrow [89]
MWM	Optimal [82]	Greedy [82]

Table 2: Overview of the domains and heuristics we explored in this work. (TE: Traffic Engineering, DP: Demand Pinning, VBP: Vector Bin Packing, MWM: Maximum Weight Matching). Arrow is randomized and optimization-based.

6.1 Traffic Engineering

We evaluated several types of traffic engineering heuristics (Tab. 2) to show MetaEase is general: Demand Pinning, which is conditional; POP, which involves randomness; and the heuristic in §2, which relies on sorting.

We use the total demand the algorithm routes as our performance metric. Following MetaOpt, we normalize the performance gap by the sum of all link capacities. We evaluate heuristics on two large topologies (Cogentco, Uninett2010) from [1] and three public production topologies: B4 [39], Abilene [79], and SWAN [35] [6]. The average link capacity in each topology is 200 Gbps. For Cogentco and Uninett2010, we restrict the routing to the 4-shortest paths. For other topologies, we use all the available paths. We use projected gradients with $K = 16$ and allow demands to vary within 2× the average link capacity. **For Demand Pinning [48]**, MetaEase discovers performance gaps larger or within 90% of what MetaOpt discovers across all topologies (Tab. 3); especially on larger topologies (Cogentco

[5] We measured the variance of all problems and found $N = 100$ balances training time and Gaussian Process accuracy. See §8.3 for an example.

[6] See Tab. 9 for the details of these topologies.

	Abilene	B4	Cogentco	Swan	Uninett2010
DP	122%	91%	451%	99%	129%
PoP	98%	89%	88%	91%	65%

Table 3: Gap Ratio (%) of MetaEase relative to MetaOpt.

Topology	**Δ Time (h)** (MetaOpt – MetaEase)	**MetaEase Gap (%)**	**MetaOpt Gap (%)**
Abilene	0.50	30%	0.02%
B4	0.88	25.47%	25.47%
Uninett2010	11.6	12.86%	0%
Cogentco	10.4	4.36%	0%

Table 4: For the heuristic in §2, MetaEase finds the same or a larger performance gap faster than MetaOpt on all topologies.

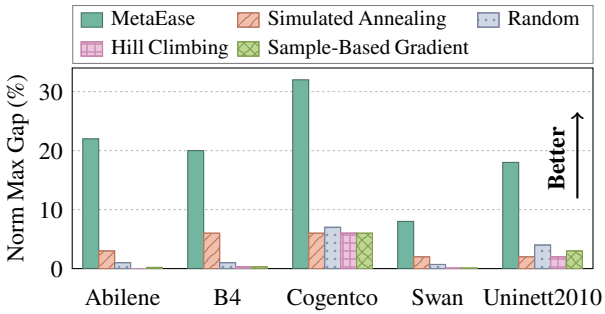


Figure 8: For Demand Pinning, MetaEase finds larger gaps compared to all the black-box baselines for the same time budget. We normalize the gap by the total capacity in each topology.

Setting (#Balls, #Dims)	(10,1)	(10,2)	(15,1)	(15,2)	(20,1)	(20,2)
Gap Ratio (%) MetaEase / MetaOpt	100%	66%	100%	75%	100%	250%

Table 5: MetaEase finds comparable gaps to MetaOpt across different problem sizes for FFD.

and Uninett2010), where MetaOpt takes longer to find the performance gap. Fig. 7 shows the evolution of the normalized maximum gap over time for production topologies, where both methods exhibit similar behavior. MetaEase finishes much faster than MetaOpt when we use it to analyze Demand Pinning. This is because of the simple control flow (e.g., threshold branches like "if demand < threshold then X, else Y"), where the maximum gap to optimal typically occurs at the boundaries. Symbolic execution efficiently generates inputs near these boundaries, which gives MetaEase an advantage.

For POP [61], we compare MetaEase against MetaOpt in Tab. 3. POP divides node pairs and their demands uniformly at random into partitions and assigns each partition an equal share of link capacities. It then solves the original optimization problem (e.g., max-flow) independently for each partition. We report the expected gap. MetaEase's results are within 12% of MetaOpt on most topologies and within 35% for Uninett2010. MetaOpt ran faster than MetaEase for POP. Analyzing POP using MetaEase is slower because POP runs separate optimizations for each partition and lacks distinct code paths. Thus, KLEE produces no output, impacting convergence.

For the example heuristic in §2, Tab. 4 shows the comparison results. We observe an unexpected result: the heuristic is optimal on the SWAN topology, and neither MetaEase nor MetaOpt found a scenario where it underperforms. On larger topologies, MetaEase finds scenarios with up to 30% larger performance gaps than MetaOpt within the same time. We set MetaOpt's timeout to 4× that of MetaEase and applied its partitioning technique for efficiency. Despite these adjustments, MetaOpt still could not find scenarios where the heuristic underperforms on most topologies.

6.2 Vector Bin Packing (VBP)

MetaEase finds equal or better performance gaps than MetaOpt in 4 of 6 settings (Tab. 5). VBP is commonly used in Virtual Machine Placement [63], where the goal is to pack multi-dimensional items into bins of fixed capacity. Performance in VBP depends on the number of used bins, with fewer being better. The optimal form of VBP is APX-hard [84], so many practitioners use first fit decreasing (FFD), a greedy and iterative heuristic [63].

For FFD, MetaEase nearly always matches MetaOpt. In one large-scale setting, MetaEase's performance gap is 2.5× larger than that of MetaOpt – MetaOpt failed to improve its objective even after 80 hours. Although MetaEase takes longer (due to KLEE), it continues to progress to a higher gap.

7 MetaEase vs Black-Box Approaches

We compare MetaEase and black-box methods including random sampling, hill climbing [27], simulated annealing [46], and sample-based gradient ascent. We run each baseline 10 times with different random seeds under the same time budget as MetaEase and report the maximum gap across runs.

7.1 MetaEase can analyze *any* numerical heuristic

For Demand Pinning [48], compared to black-box baselines, MetaEase finds up to 7.3× larger performance gaps under the same time budget (Fig. 8). Fig. 9 shows the progress of each algorithm over time for publicly available topologies. We set the demand pinning threshold to 5% of average link capacities.

DOTE [65]. We evaluate DOTE, a DNN-based traffic engineering heuristic that MetaOpt does not support. In this case, we compare MetaEase against the DNN analyzer in [59]. MetaEase finds the largest performance gap among all the baselines (Tab. 6). We used the open-source DOTE implementation [66] and trained it on the Abilene topology until it reached 99.95% test accuracy. We compare against the solution from [59] (which the authors provided) as well as black-box search methods.

For POP [61], Fig. 10 shows that MetaEase finds larger perfor-

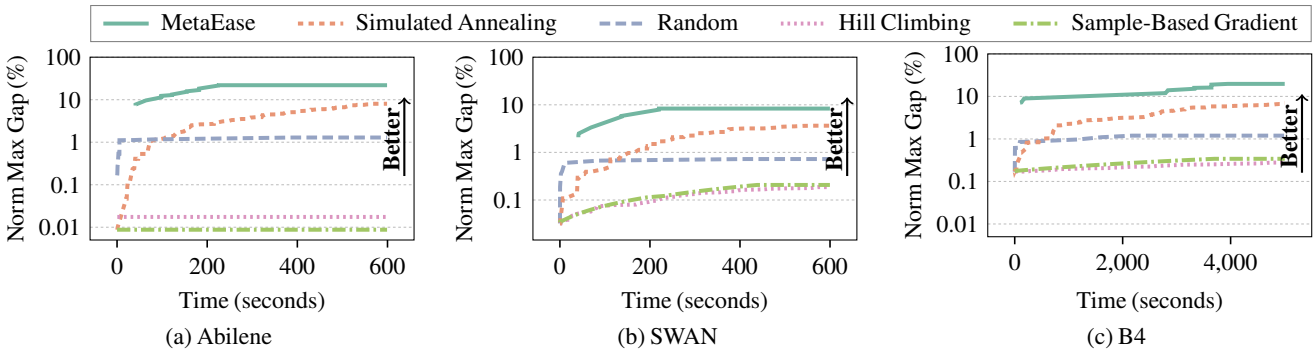


Figure 9: MetaEase finds larger performance gaps faster compared to black-box approaches. For a fair comparison, we allow all methods to fully utilize all threads available on the machine.

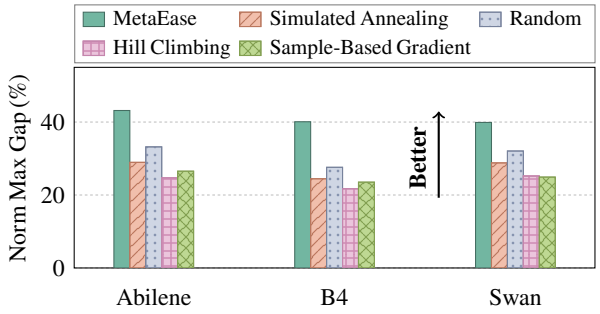


Figure 10: MetaEase finds a larger performance gap than black-box baselines for POP across production topologies under the same time budget.

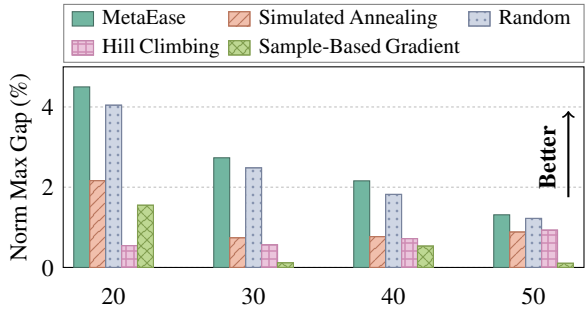


Figure 11: MetaEase outperforms all the baselines when it analyzes knapsack problems for 20, 30, 40, and 50 items. We normalized the gaps by number of items multiplied by the upper bound on each item's weight.

Method	Normalized Max Gap
MetaEase	73.13%
DNN Analyzer in [59]	71.84%
Random	63.02%
Simulated Annealing	61.39%
Hill Climbing	58.78%
Sample-based Gradient	58.78%

Table 6: For DOTE, MetaEase finds the largest performance gap on the Abilene topology.

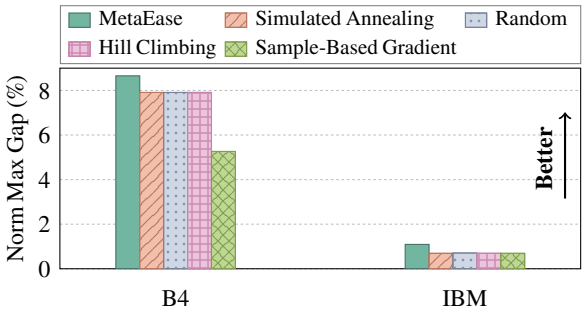


Figure 12: For Arrow, MetaEase finds larger gaps in the same time budget. We show the maximum gaps normalized by the total capacity of the healthy network.

mance gap compared to black-box baselines across production topologies (Abilene, B4, SWAN) under the same time budget. **For the example heuristic in §2**, MetaEase also outperforms black-box algorithms (Fig. 20).

7.2 MetaEase can analyze diverse problem domains

Domains where the optimal benchmark solution is convex, such as traffic engineering, are well-suited for MetaEase. We show MetaEase also performs well on other (non-convex) problems using four examples (Tab. 2):

For Knapsack, MetaEase again shows larger performance gaps relative to all black-box baselines (Fig. 11). Given a set of items with associated weights and values, the knapsack problem determines which subset of items maximizes the total value while ensuring the total weight does not exceed a specified limit. It is NP-complete [82] and is used to solve problems such as LTE downlink scheduling [31]. We analyze the commonly used greedy heuristic [31], which computes (1) each item's value-to-weight ratio, (2) sorts items by this ratio, and (3) picks items in order until capacity is exhausted.

Black-box approaches also perform well here: when item values and weights vary widely, the greedy heuristic is more likely to underperform, and random sampling is more likely to encounter these adverse cases.

For Arrow [89], MetaEase outperforms black-box baselines (Fig. 12). To our knowledge, no other heuristic analyzer,

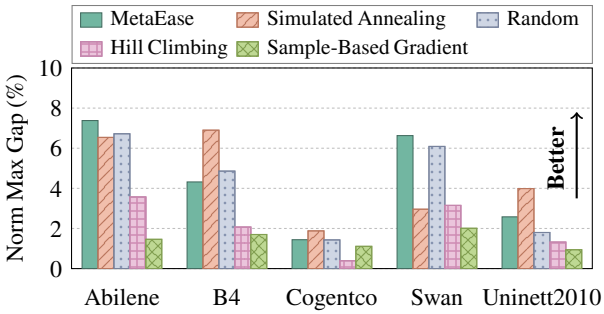


Figure 13: MetaEase finds the largest performance gap across most topologies when we use it to analyze a greedy heuristic that solves the maximum weight matching problem.

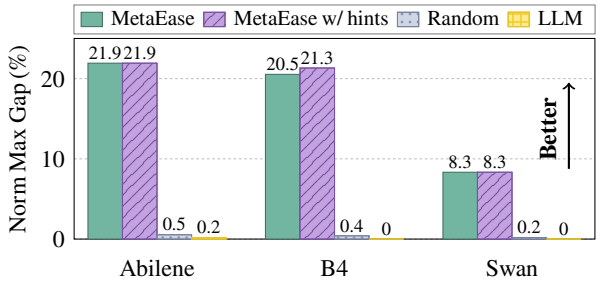


Figure 14: MetaEase with hints consistently produces larger performance gaps over the baselines, followed closely by vanilla MetaEase. All MetaEase variants outperform the baselines.

including MetaOpt, can analyze Arrow since it solves a non-convex optimization with an objective. Arrow jointly assigns optical resources (wavelengths and fibers) after a fiber cut. It introduces a randomized heuristic through its LotteryTickets concept and models the heuristics as a mixed-integer optimization. As a result, the heuristic has a single code path, and although we lose the advantages of using KLEE, MetaEase is still able to find competitive performance gaps.

Random search also performs well in this problem because it is very lightweight and explores more samples than other methods. In contrast, MetaEase incurs overhead to run KLEE and train a Gaussian Process; the former provides minimal benefit.

We use the B4 topology (12 nodes, 39 optical links) and the IBM topology (17 nodes, 154 optical links) in our experiments [89]. We run Arrow with 3 LotteryTickets and consider 3 single-fiber cut scenarios. We repeat each setting 10 times and verified that this sample size captures Arrow's variance.

For Maximum Weight Matching, MetaEase shows the largest performance gap on the Abilene, Cogentco, and Uninett2010 topologies and consistently ranks among the top two approaches (Fig. 13). Given a graph with weighted edges, maximum weight matching selects a set of non-overlapping edges to maximize the sum of their weights. Prior work used it to understand the throughput limits of datacenter topologies [41, 60]. We analyze a greedy heuristic from [82] and omit the details for brevity.

8 Evaluating MetaEase's Design Choices

We now demonstrate how each module in MetaEase contributes to its improved performance where we analyze the demand pinning heuristic using public topologies.

8.1 Seed generation module

MetaEase uses KLEE to partition inputs into equivalence classes and seeds the gradient-based search with samples from each class. Developers may optionally provide domain knowledge to select better starting points for each class (see §4.2).

We compare MetaEase w/ hints with three alternatives in Fig. 14: (i) MetaEase without hints; (ii) using 30 random starting points; and (iii) replacing KLEE with an LLM that analyzes the code and generates equivalence classes. For the latter, we prompted GPT-5 in reasoning mode to analyze the heuristic's code along with the topology and generate diverse demand matrices exposing different heuristic behaviors (see Fig. 21).

MetaEase with hints consistently finds gaps that are larger or comparable to vanilla MetaEase, and it is on average 1.6× faster. MetaEase also discovers substantially larger performance gaps than other methods. Surprisingly, the LLM-based equivalence classes did not improve the gradient-based search — this may mean we need to engineer a better prompt, but it may also indicate that LLMs are not well suited to this problem.

8.2 Gradient ascent module

Path-based gradients help with non-differentiability. We analyze the Demand Pinning heuristic on the Abilene topology. In this case, KLEE returned a starting point for an equivalence class that fell exactly at the demand pinning threshold. The gradient-based search then identified a larger performance gap than this initial point. Using a path-based gradient allowed us to find an even greater gap compared to a search that did not remain within the same equivalence class (Fig. 15–left).

Effectiveness of Gaussian Process-Based Search. We compared gradient ascent using Gaussian Process surrogates with two sample-based variants: (1) directly estimating heuristic's gradient and running gradient ascent for each code path, and (2) the traditional sample-based gradient ascent search. Both sample-based approaches were slower (Fig. 15–middle).

Sample-based gradient ascent is less efficient than the Gaussian Process approach (Fig. 15–right), requiring 4× more heuristic calls than MetaEase. For d variables, it requires $d+1$ heuristic calls per step, whereas MetaEase makes only N calls, regardless of the number of variables. This gap widens on larger topologies: MetaEase is 16.6× faster for Demand Pinning on B4 and 2.8× faster on Abilene (Fig. 15–middle).

8.3 The impact of hyperparameters

MetaEase has three main hyper-parameters: the number of dimensions for projected gradients (K), the block size for training the Gaussian Process (Δ), and the number of samples per block (N). K introduces a trade-off between input space exploration and the time required to run KLEE and train the

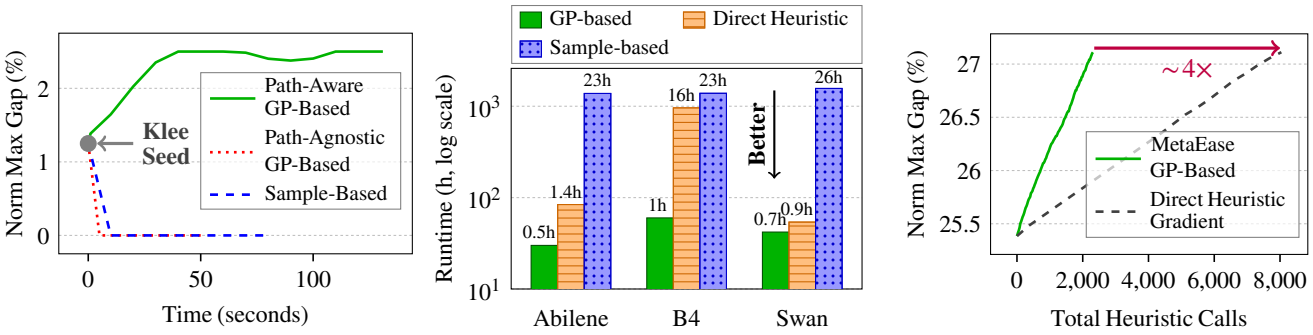


Figure 15: "Path-based" and "Surrogate-based" methods are effective: (left) MetaEase's gradient ascent improves upon the initial point and handles the discontinuities; (middle) surrogate-based is faster than sample-based variants, including vanilla sample-based and one where it only estimates heuristic's gradient; (right) MetaEase reduces the number of heuristic calls.

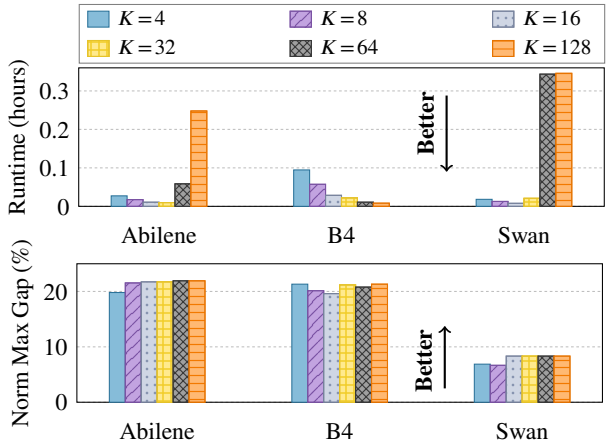


Figure 16: The number of dimensions in projected gradients (K) introduces a tradeoff between the quality of the discovered performance gap and the runtime.

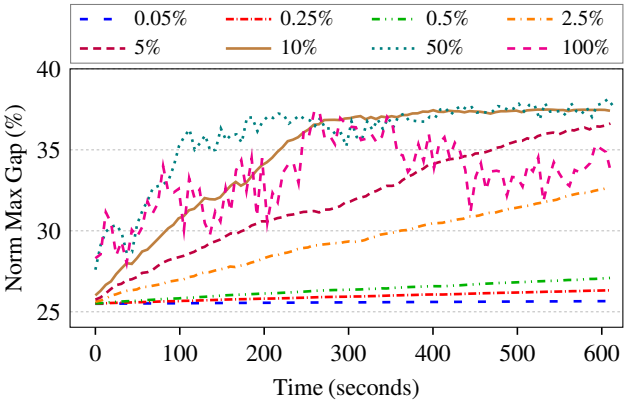


Figure 17: The block size Δ introduces a tradeoff: a larger Δ reduces the quality of the approximate gradient (which causes oscillations in MetaEase), while a smaller Δ increases runtime.

Gaussian Process (Fig. 16). Block size introduces another trade-off: larger blocks reduce the accuracy of the Gaussian Process surrogate and the gradient estimate, while very small blocks increase MetaEase's runtime (Fig. 17). The number of samples per block, N, directly affects the accuracy of the Gaussian Process (Fig. 18). We evaluate the Gaussian Process using 200 held-out samples. We observe that increasing N improves surrogate quality but also increases runtime.

9 Related Work

General domain heuristic analyzers. To our knowledge, MetaEase is the first general heuristic analyzer that operates without a mathematical model of the heuristic, instead exploiting its structure to guide search. Prior approaches either encode heuristics into analytic abstractions—such as XPlain's network-flow model [43] or MetaOpt's optimization formulation [57]—or treat them as black boxes, using simulated annealing, hill climbing [27, 46], sample-based gradient methods [71], Bayesian/surrogate optimization [18, 30, 32, 40, 42, 54, 70, 75], or derivative-free global search [24]. Symbolic execution tools likewise generate targeted adversarial test cases [19, 23, 62].

Domain-specific analysis and anomaly detection. Several systems focus on specific domains: Virelay verifies scheduling heuristics with SMT [33], and FPerf synthesizes adversarial packet schedules to expose throughput and fairness issues [9]. For congestion control, CCAC symbolically explores adversarial traces [12], constraint-guided templates yield provable guarantees [6], and prior work identifies starvation pathologies [11]. Analytic models reveal stability and fairness problems, from classic TCP fluid models [56] to modern analyses of high-bandwidth flow control [29] and CCAs like BBR [73]. HotCocoa adds NIC-oriented abstractions for implementing and analyzing CCAs [10]. Other works motivate solver- or test-based analyzers: Buffy offers a solver-agnostic language for performance analysis [76]; DNS studies find combined performance and security bugs [51]; Dote exposes adversarial samples in learning-enabled systems [59]; and Raha detects WAN degradation [13]. Control-plane verification has advanced significantly [4, 16, 67, 68, 74, 78, 80, 81]. Collie extends these efforts to RDMA subsystems [47]. A number of prior works have used symbolic execution to analyze performance in specific domains [22, 36–38]. None are as comprehensive nor as easy to use as MetaEase (or MetaOpt even). MetaEase uses symbolic execution as a guide not to directly solve the problem.

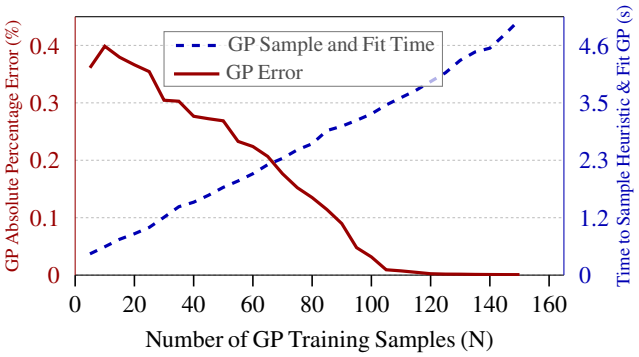


Figure 18: Training the Gaussian Process with more samples improves accuracy but increases runtime.

10 Limitations and Scope

Scope. MetaEase analyzes heuristics directly from their implementation. In principle, it applies to any heuristic that takes concrete inputs and ultimately produces a numeric performance output; if the outcome is purely categorical with no meaningful numeric interpretation, MetaEase does not apply.

MetaEase assumes piecewise-smooth functions in its path-aware gradients. We assume, within a fixed code-path, the heuristic is approximately smooth — this is when the local GP surrogate provides a useful direction. If the heuristic remains highly discontinuous even within a code-path, surrogate gradients can become noisy and search quality degrades.

Scalability, symbolic execution limits, and failure cases. Symbolic execution is hard to scale. Heuristics where the implementation creates extreme path explosion, deep input-independent loops, or complex stateful data structures are difficult for KLEE (or similar symbolic-execution tools) to cover exhaustively. In such cases, MetaEase may start from incomplete seed coverage and can fall back to random multi-start or user-guided seeding constraints. Likewise, as input dimensionality and benchmark complexity increase, runtime grows and convergence may slow. MetaEase is a heuristic search method and is *not* guaranteed to find the global maximum gap.

11 Discussion

MetaEase extends beyond heuristic analysis. Raha [13] shows worst-case network degradation can be found by searching over adversarial conditions. MetaEase can similarly treat failure scenarios as inputs I (e.g., link or node failures) and maximize the performance gap subject to plausibility constraints (e.g., at most k simultaneous failures). This yields the same Stackelberg-style formulation as in our main objective [57]. A full empirical study of this setting is left to future work.

Alternative definitions of equivalence classes may also work. Our goal was to find regions in the input space where the heuristic makes the same decision, which motivated our use of KLEE. However, other mechanisms may perform better for certain heuristics, and we leave this for future work.

We can find equivalence classes in the benchmark, too. Users can provide the benchmark implementation instead of the optimization model. In such cases, we can apply KLEE to the benchmark and define equivalence classes as the Cartesian product of the classes returned for the benchmark and the heuristic.

MetaEase and MetaOpt are complementary. MetaOpt is preferable when the heuristic can be cleanly modeled as a mathematical program and its solver scales. It provides stronger guarantees, but it is impractical for heuristics with complex control flow, sorting, randomness, or machine learning components that are hard to encode. In such cases, MetaEase is preferable because it analyzes the source code directly without re-modeling.

Random black-box methods may suffice in some cases. For some heuristics (e.g., Arrow), the performance curve is nearly flat. In such scenarios, random black-box techniques can explore the search space quickly and achieve a performance gap close (though still smaller) to that of MetaEase. Detecting when a heuristic falls into this category could enable switching to these faster methods when appropriate.

We can make MetaEase faster. Since we run MetaEase's gradient ascent from different seeds in parallel, we can improve speed without an inherent upper limit by using more threads.

MetaEase can help with heuristic design. MetaEase's output is not just a gap value; it is a set of concrete worst-case inputs together with the decision regimes that lead to large degradation. This enables an actionable workflow to improve heuristics: (i) root-cause analysis by inspecting which branches are triggered, (ii) safe deployment by converting recurring failure patterns into runtime checks and fallbacks, (iii) regression testing by adding discovered inputs to a stress-test suite, and (iv) heuristic improvement and robustness by using these hard cases for manual refinement or automated LLM-based tools [34, 45].

12 Conclusion

To our knowledge, MetaEase is the first general-domain heuristic analyzer that analyzes a heuristic's *implementation* and enables heuristic developers to quantify the performance risk it imposes on their networks. Unlike prior approaches, MetaEase requires no mathematical formulation of a heuristic, making it more accessible to developers. MetaEase combines symbolic execution and gradient-based search to find the maximum gap between heuristics and benchmarks. Our evaluation across diverse heuristics and domains demonstrates its effectiveness. We plan to open-source MetaEase to support developers in assessing heuristic risks.

Acknowledgements

This work was funded in part by Microsoft Research internship and the U.S. National Science Foundation under Award No. 2212102. We sincerely thank Amin Khodaverdian for his valuable support. We also thank Frank Wang for his feedback. Finally, we thank our shepherd, Chang Lou, and the NSDI committee for their valuable guidance and feedback.

References

[1] Internet topology zoo. http://www.topology-zoo.org/. Accessed: 2025-09-11.

[2] Klee documentation (v2.1). https://klee-se.org/releases/docs/v2.1/docs/. Accessed: 2025-09-18.

[3] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic meets modern: a pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 632–647, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and general network verification. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 201–219, 2020.

[5] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: Active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 36–52, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Towards provably performant congestion control. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 951–978, 2024.

[7] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. {SP-PIFO}: Approximating {Push-In}{First-Out} behaviors using {Strict-Priority} queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, 2020.

[8] Albert Gran Alcoz, Balázs Vass, Pooria Namyar, Behnaz Arzani, Gábor Rétvári, and Laurent Vanbever. Everything matters in programmable packet scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1467–1485, 2025.

[9] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[10] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets '17, page 108–114, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 177–192, 2022.

[12] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.

[13] Behnaz Arzani, Sina Taheri, Pooria Namyar, Ryan Beckett, Siva Kakarla, and Elnaz Jallilipour. Raha: A general tool to analyze wan degradation. In *Proceedings of the ACM SIGCOMM 2025 conference*, 2025.

[14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[15] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. Virtual machine allocation with lifetime predictions. *Proceedings of Machine Learning and Systems*, 5:232–253, 2023.

[16] Ryan Beckett, Ratul Mahajan, Jitendra Padhye, and David Walker. A general approach to network configuration verification (minesweeper). In *ACM SIGCOMM*, pages 321–334, 2017.

[17] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[18] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[19] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[20] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[21] Paolo H. Calamai and Jorge J. More. Projected gradient methods for linearly constrained problems. In *Nonlinear Programming 3*, pages 71–116. Springer, 1987.

[22] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI*

17), pages 347–362, Boston, MA, March 2017. USENIX Association.

[23] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[24] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM, 2009.

[25] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*. IEEE, 2012.

[26] John M. Danskin. The theory of max–min, with applications. *SIAM Journal on Control*, 5(1):64–104, 1967.

[27] Lawrence Davis. Bit-climbing, representational bias, and test suite design. In *Proc. 4th Int'l Conf. on Genetic Algorithms*, pages 18–23. Morgan Kaufmann, 1991.

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[29] Nandita Dukkipati and Nick McKeown. Flow control in fast long-distance networks. In *IEEE INFOCOM*, pages 1079–1088, 2006.

[30] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D. Turner, and Matthias Poloczek. Scalable global optimization via local bayesian optimization. In *NeurIPS*, 2019.

[31] Nasim Ferdosian, Mohamed Othman, Borhanuddin Mohd Ali, and Kweh Yeah Lun. Greedy–knapsack algorithm for optimal downlink resource allocation in lte networks. *Wireless Networks*, 22(5):1427–1440, 2016.

[32] Alexander Forrester, András Sobester, and Andy Keane. *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley, 2008.

[33] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. Quantitative verification of scheduling heuristics, 2023.

[34] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. Glia: A human-inspired ai for automated systems design and optimization, 2025.

[35] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[36] Yigong Hu, Gongqi Huang, and Peng Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 719–734. USENIX Association, November 2020.

[37] Rishabh Iyer, Katerina Argyraki, and George Candea. Automatically reasoning about how systems code uses the CPU cache. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 581–598, Santa Clara, CA, July 2024. USENIX Association.

[38] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 517–530, 2019.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[40] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, 1998.

[41] Sangeetha Abdu Jyothi, Ankit Singla, P. Brighten Godfrey, and Alexandra Kolla. Measuring and understanding throughput of network topologies. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 761–772, 2016.

[42] Motonobu Kanagawa, Philipp Hennig, Dino Sejdinovic, and Bharath K Sriperumbudur. Gaussian processes and kernel methods: A review on connections and equivalences. *arXiv preprint arXiv:1807.02582*, 2018. Posterior mean of GP regression equals kernel ridge regression.

[43] Pantea Karimi, Solal Pirelli, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Beibin Li, Pooria Namyar, and Behnaz Arzani. Towards safer heuristics with xplain. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 68–76, 2024.

[44] Pantea Karimi, Solal Pirelli, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Beibin Li, Pooria Namyar, and Behnaz Arzani. Towards safer heuristics with xplain, 2024.

[45] Pantea Karimi, Dany Rouhana, Pooria Namyar, Siva Kesava Reddy Kakarla, Venkat Arun, and Behnaz Arzani. Robust heuristic algorithm design with llms, 2025.

[46] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[47] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in {RDMA} subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022.

[48] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with {BLASTSHIELD}. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 325–338, 2022.

[49] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonnette, Nikolaj Bjørner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, et al. {OneWAN} is better than two: Unifying a split {WAN} architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 515–529, 2023.

[50] Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill Education, New York, 5 edition, 2015.

[51] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A formal framework for end-to-end dns resolution. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 932–949, 2023.

[52] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[53] Nick McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking*, 2002.

[54] Charles A Micchelli, Yuesheng Xu, and Haizhang Zhang. Universal kernels. *Journal of Machine Learning Research*, 7:2651–2667, 2006.

[55] Paul Milgrom and Ilya Segal. Envelope theorems for differential and nondifferentiable optimization. *Econometrica*, 70(2):583–601, 2002.

[56] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *ACM SIGCOMM*, pages 151–160, 2000.

[57] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. Finding adversarial inputs for heuristics using multi-level optimization. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 927–949, 2024.

[58] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. Solving {Max-Min} fair resource allocations quickly on large graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[59] Pooria Namyar, Michael Schapira, Ramesh Govindan, Santiago Segarra, Ryan Beckett, Siva Kesava Reddy Kakarla, and Behnaz Arzani. End-to-end performance analysis of learning-enabled systems. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 86–94, New York, NY, USA, 2024. Association for Computing Machinery.

[60] Pooria Namyar, Sucha Supittayapornpong, Mingyang Zhang, Minlan Yu, and Ramesh Govindan. A throughput-centric view of the performance of datacenter topologies. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 349–369, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[62] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.

[63] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. January 2011.

[64] Laurent Perron and Vincent Furnon. Or-tools.

[65] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. {DOTE}: Rethinking (predictive){WAN} traffic

engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581, 2023.

[66] PredWanTE. Predwante/dote: Implementations for dote traffic engineering. https://github.com/PredWanTE/DOTE, 2023. Accessed: 2025-09-03.

[67] Divya Raghunathan, Ryan Beckett, Aarti Gupta, and David Walker. Acorn: Network control plane abstraction using route nondeterminism. *arXiv preprint arXiv:2206.02100*, 2022.

[68] Divya Raghunathan et al. Abstract interpretation of distributed network control planes (shapeshifter). In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 213–227, 2020.

[69] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, 2024.

[70] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[71] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[72] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 404–417, 2017.

[73] Simon Scherrer, Markus Legner, Adrian Perrig, and Stefan Schmid. Model-based insights on the performance, fairness, and stability of bbr. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 519–537, 2022.

[74] Johann Schlamp, Matthias Wählisch, Thomas C. Schmidt, Georg Carle, and Ernst W. Biersack. Cair: Using formal languages to study routing, leaking, and interception in bgp. *arXiv preprint arXiv:1605.00618*, 2016.

[75] Matthias Seeger. Gaussian processes for machine learning. *International journal of neural systems*, 14(02):69–106, 2004.

[76] Amir Seyhani, Junyi Zhao, Aarti Gupta, David Walker, and Mina Tahmasbi Arashloo. Buffy: A formal language-based framework for network performance analysis. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 95–102, New York, NY, USA, 2024. Association for Computing Machinery.

[77] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. {TACCL}: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[78] Xiaozhe Shao, Zibin Chen, Daniel Holcomb, and Lixin Gao. Accelerating bgp configuration verification through reducing cycles in smt constraints (binode). *IEEE/ACM Transactions on Networking*, 2021.

[79] Stanford University IT. Abilene core topology. Technical web page, 2015. Accessed: 2025-09-11.

[80] S. Steffen et al. Probabilistic verification of network configurations (netdice). In *ACM SIGCOMM*, 2020.

[81] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 94–107, New York, NY, USA, 2023. Association for Computing Machinery.

[82] V.V. Vazirani. *Approximation Algorithms*. Springer Berlin Heidelberg, 2013.

[83] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-through: part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 327–338, New York, NY, USA, 2010. Association for Computing Machinery.

[84] Gerhard J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, 1997.

[85] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.

[86] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404, 2017.

[87] Yiwen Zhang, Xumiao Zhang, Ganesh Ananthanarayanan, Anand Iyer, Yuanchao Shu, Victor Bahl, Z Morley Mao, and Mosharaf Chowdhury. Vulcan: Automatic query planning for live {ML} analytics. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1385–1402, 2024.

[88] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Efficient {Direct-Connect} topologies for collective communications. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 705–737, 2025.

[89] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. Arrow: restoration-aware traffic engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 560–579, 2021.

A Encoding a New TE Heuristic in MetaOpt

We wanted to analyze a new sort-based heuristic in MetaOpt. This heuristic routes the demands and selects the top 20% as critical demands, route those demands on the network optimally in stage 1, and then route the rest of the non-critical demands on the residual capacities optimally in stage 2 (see Code 1).

Code 1: New Sort-based Traffic Engineering Heuristic

```
double SortBasedHeuristic(Graph *G) {
   // collect and sort demands
   Demands D = positive_demands(G);
   sort_desc(D);

   // split top 20% as critical
   int k = max(1, 0.2 * size(D));
   Demands Critical = top_k(D, k);
   Demands Remaining = rest(D, Critical);

   // solve optimal on critical demands first
   Routing
     criticalRoutes = SolveOptimal(G, Critical);
   // Adjust the capacities
   Graph G = AdjustCapacities(G, criticalRoutes);
   // solve optimal on non critical demands
   Routing
     nonCriticalRoutes = SolveOptimal(G, Remaining);

   return G->total_met_demands;
}
```

We formulated this heuristic in MetaOpt (see Fig. 19). This formulation is convoluted despite the helper functions MetaOpt provides. The user has to find a way to extract the non-convex parts of the optimization into the outer problem (Raha [13] describes a similar challenge).

To see why this is hard in MetaOpt, let us go through how we can sort the top 20% of demands as part of the optimization. First, we need to introduce binary variables c_i which indicate whether a demand is in the top 20% or not. We face two challenges: (1) since binary variables are non-convex we need to make sure that we do not use them directly to model the heuristic; (2) we need to set the values such that c_i is 1 if the demand falls in the top 20% and to 0 otherwise (since we do not know the values of the demands before we solve the optimization problem we cannot set these values ahead of time). Constraints 10-15 in Fig. 19 allows us to solve the second problem and we also have to apply tricks to move the values c_i out of the heuristic model to avoid the non-convexity they introduce.

Note that optimization formulation is error-prone and it requires careful considerations and testing. Consequently, it's not straightforward to just ask LLMs to write them. There needs to be human interventions to make sure the formulations are correct.

B Gaussian Process Surrogate Gradient

A Gaussian Process (GP) is a function of the form:

$$\text{Heuristic}_{GP}(I) = m(I) + Cov(I,\mathbf{X})\mathbf{K}^{-1}\big(\mathbf{y} - m(\mathbf{X})\big)$$

which we need to train in order to find the mean function $m(\cdot)$, and the covariance kernel $Cov(\cdot)$. To train it, we use data $(\mathbf{X},\mathbf{y})$, where $\mathbf{X} = \{x_1,\ldots,x_N\}$ are samples from the input region Δ^n and they all are in the same code-path, and $\mathbf{y} = \big(\text{Heuristic}(x_1),\ldots,\text{Heuristic}(x_N)\big)^\top$ are the outputs when we run the heuristic on the input samples.

The analytic gradient of the Gaussian process at I is

$$\nabla\text{Heuristic}_{GP}(I) = \nabla m(I) + \nabla Cov(I,\mathbf{X})\mathbf{K}^{-1}\big(\mathbf{y} - m(\mathbf{X})\big),$$

where

$$\nabla Cov(I,\mathbf{X}) = \big[\nabla_I Cov(I,x_1),\ldots,\nabla_I Cov(I,x_N)\big].$$

The gradient of the GP is closed-form and we leverage that in estimating the gradient of the heuristic in region Δ^n.

C How MetaEase Restricts the Search within One Code Path

For each seed I returned by KLEE [19], we run Heuristic(I) and record its code-path signature $h = \text{path}(I)$. This signature defines the equivalence class $Class_h$.

At iteration t of updates, given the current point I_t ($\text{path}(I_t) = h$), we draw a batch of samples in a small box around I_t (side length 2Δ). We evaluate the heuristic on these candidates and retain those with signature h, forming a training set $\mathbf{X}_t$. From $\mathbf{X}_t$, we select N points to fit a Gaussian Process surrogate, $\text{Heuristic}_{GP}(x)$. We then compute the objective gradient

$$g_t = \nabla\text{Benchmark}(I_t) - \nabla\text{Heuristic}_{GP}(I_t),$$

according to §2.4, and propose the next step

$$I_{t+1} = I_t + \eta g_t.$$

If $\text{path}(I_{t+1}) = h$, we accept the step. Otherwise, we project the move back into the same class: specifically, we choose the point $z \in \mathbf{X}_t$ whose direction from I_t has the smallest acute angle with g_t, i.e.,

$$z = \operatorname*{argmax}_{u\in\mathbf{X}_t} \frac{g_t^\top(u - I_t)}{\|u - I_t\|} \quad \text{s.t.} \quad \frac{g_t^\top(u - I_t)}{\|u - I_t\|} > 0.$$

We then set $I_{t+1} = z$. This ensures progress while staying inside $Class_h$.

The process repeats until convergence, budget exhaustion, or until all candidate angles are obtuse ($g_t^\top(u - I_t) < 0$), at which point the ascent terminates.

Tool	Heuristic format as input	Guarantee worst-case type	Heuristic Type
MetaEase	C implementation (Code)	Empirical best found (symbolic-guided)	Domain agnostic. Any heuristic (non-convex, DNN-based, randomized, etc)
MetaOpt [57]	Optimization form	Formal lower bound	Domain agnostic. Only convex or feasibility heuristic
Virelay [33]	Model + assumptions (SMT)	Formal (model-level)	General Domain (focuses on scheduling). Any heuristic expressible in SMT
XPlain [43]	Network-flow Abstraction	Formal lower bound	Domain agnostic. Only convex or feasibility heuristic
FPerf [9]	Queueing model + Query	Witness (existence)	Queue management. Discrete-event, rule-based, non-convex (e.g., FIFO)
Black-box Search	Executable	Empirical best found (blind search)	Domain agnostic. Any heuristic

Table 7: MetaEase compared to prior performance analyzers and black-box baselines. Unlike prior tools, MetaEase operates directly on heuristic code and leverages symbolic-guided search to uncover large performance gaps of heuristic compared to a benchmark.

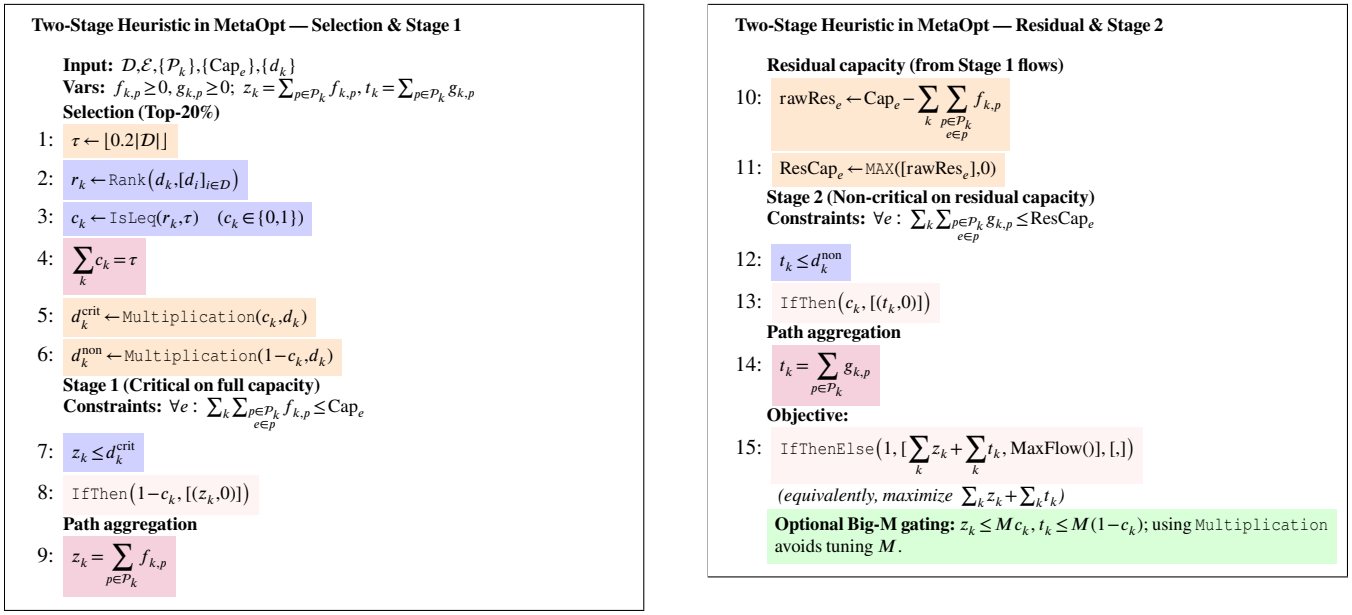


Figure 19: Encoding of Traffic Engineering heuristic in MetaOpt. This heuristic selects the top-20% of demands as critical (1) route critical set optimally, (2) then route remaining non-critical demands on residual capacities. This required a lot of attention and LLMs can't do it without human supervision to check for correctness.

Symbol	Meaning
Δ	Block size for GP sampling around a point §3.2
N	Number of samples for fitting the GP §3.2
η	Learning rate for gradient ascent §3.4
K	Number of variables in projected gradient technique §4.1

Table 8: Parameters in MetaEase's gradient ascent

Topology	#Nodes	#Edges
Cogentco	197	486
Uninett2010	74	202
Abilene	10	26
B4	12	38
Swan	8	24

Table 9: Detail of topologies used for evaluation. We used similar topologies to MetaOpt [57].

D How MetaEase Handles Randomness

In many cases, developers may be dealing with randomized heuristics, or they may want to measure the performance gap over different scenarios. For example, in studying Arrow's heuristic in §7.2, we look at the performance gap across different standards of fiber cut. In these cases, MetaEase maximizes the *expected* performance gap. Let $I \in \mathcal{I}$ denote the input, ξ the exogenous "scenario" randomness (different scenarios of fiber cut), and τ the heuristic's internal randomness (e.g., number of tickets in Arrow (§7.2), random partitions in POP (§6.1).

Objective. We denote the expected performance gap as

$$\max_{I \in \mathcal{I}} \mathbb{E}_{\tau,\xi}\left[\text{Benchmark}(I;\xi) - \text{Heuristic}(I;\tau,\xi)\right] \tag{4}$$

By linearity of expectation,

$$\max_{I \in \mathcal{I}} \mathbb{E}_{\xi}\left[\text{Benchmark}(I;\xi)\right] - \mathbb{E}_{\tau,\xi}\left[\text{Heuristic}(I;\tau,\xi)\right]. \tag{5}$$

Gradients. Under mild regularity (differentiation under the expectation),

$$\nabla_I \mathbb{E}_\xi\big[\text{Benchmark}(I;\xi)\big] = \mathbb{E}_\xi\big[\nabla_I \text{Benchmark}(I;\xi)\big], \quad (6)$$

$$\nabla_I \mathbb{E}_{\tau,\xi}\big[\text{Heuristic}(I;\tau,\xi)\big] = \mathbb{E}_{\tau,\xi}\big[\nabla_I \text{Heuristic}(I;\tau,\xi)\big]. \quad (7)$$

We keep the same decomposition used in the deterministic case:

$$\nabla \text{Gap}(I) = \underbrace{\nabla \text{Benchmark}(I)}_{\text{via dual/Lagrangian}} - \underbrace{\nabla \text{Heuristic}(I)}_{\text{via local GP surrogate}},$$

now interpreted in expectation per (6)–(7).

Averaging over Samples. We approximate these expectations by averaging over random samples, reusing the same random draws across both benchmark and heuristic to reduce variance [50]. At each ascent step, we approximate the expectations by averages over n_ξ scenarios and n_τ heuristic draws (often $n_\tau = 1$ if the heuristic's randomness is embedded in the scenario). Draw paired samples

$$\{\xi_i\}_{i=1}^{n_\xi}, \quad \{(\tau_{i,j},\xi_i)\}_{j=1}^{n_\tau} \text{ for each } i,$$

reusing the same ξ_i across the benchmark and heuristic to reduce variance [50].

Benchmark term. When the benchmark admits the Lagrangian in Eq. 3, for each scenario ξ_i we form

$$\phi_i(\lambda_i,\nu_i;I,\xi_i) \triangleq \sup_{x^o} \mathcal{L}\left(x^o,\lambda_i,\nu_i;I,\xi_i\right),$$

and compute its I-gradient (no solver call). The MC gradient estimator is

$$\widehat{\nabla}\text{Benchmark}(I) = \frac{1}{n_\xi}\sum_{i=1}^{n_\xi} \nabla_I \phi_i(\lambda_i,\nu_i;I,\xi_i). \quad (8)$$

Heuristic term. For stochastic heuristics, we directly roll out the C implementation $I \mapsto \text{Heuristic}(I;\tau,\xi)$ across n_τ random seeds τ and n_ξ random scenarios ξ. We report the *average performance* over these runs:

$$\overline{\text{Heuristic}}(I) = \frac{1}{n_\xi n_\tau}\sum_{i=1}^{n_\xi}\sum_{j=1}^{n_\tau} \text{Heuristic}(I;\tau_{i,j},\xi_i).$$

We then fit a local Gaussian Process surrogate to $\overline{\text{Heuristic}}(I)$ in the current neighborhood, and take its analytic gradient:

$$\widehat{\nabla}\text{Heuristic}(I) = \nabla_I \text{Heuristic}^{\text{GP}}(I). \quad (9)$$

Notes. We use this rolled out C implementation to find KLEE points as well. This way, the randomness also takes effect on the seed generation. If only the heuristic is randomized, we set n_ξ=1 for the benchmark and average over τ (POP in §6.1); if both the scenarios and heuristic are randomized(Arrow in §7.2), we couple the same ξ_i across both terms.

E Example of *Hint* in MetaEase

Hints in MetaEase are optional, domain-specific assumptions that guide symbolic execution toward more informative regions of the input space. They do not affect correctness; rather, they improve scalability by pruning inputs that are trivial or unlikely to produce meaningful performance gaps.

In traffic engineering with the Demand Pinning heuristic, we provide two types of hints:

(1) Range constraints. We restrict each demand to lie within a reasonable interval. This reduces the search space explored by KLEE, allowing it to focus on practically relevant demand magnitudes.

(2) Hardness predicates. If the total demand traversing every edge is strictly below that edge's capacity, the instance is trivial: both the heuristic and the benchmark can satisfy all demands, resulting in no performance gap. Therefore, we guide KLEE to generate only inputs that place at least one edge under pressure, i.e., where aggregate demand exceeds capacity on some edge. This increases the likelihood of exposing worst-case behavior.

Code 2: Example of domain-specific hint for Traffic Engineering

```
for (int k = 0; k < K; ++k) {
  // Restrict
      each demand to a bounded interval.
  klee_assume(0 <= demand[k] && demand[k] <= D_MAX);
}

bool any_edge_overloaded = false;

for (int e = 0; e < num_edges; ++e) {
  // Compute the total
      load induced on edge e by all demands.
  int edge_load
      = sum_demands_on_edge(demand, edges[e]);

  // Track whether
      this input stresses at least one edge.
  // We want: exists
      e such that edge_load(e) > capacity(e).
  any_edge_overloaded = any_edge_overloaded
      || (edge_load > edges[e].Capacity);
}

// Hardness predicate:
// If no edge is overloaded
    , the instance is typically "easy",
// and is unlikely to expose a gap
      between the heuristic and the benchmark.
klee_assume(any_edge_overloaded);
```

F Other MetaEase Runtime Optimizations

Developers may want to do quick initial tests to make sure the heuristic they designed performs well in the majority of the input space. MetaEase provides options that allow

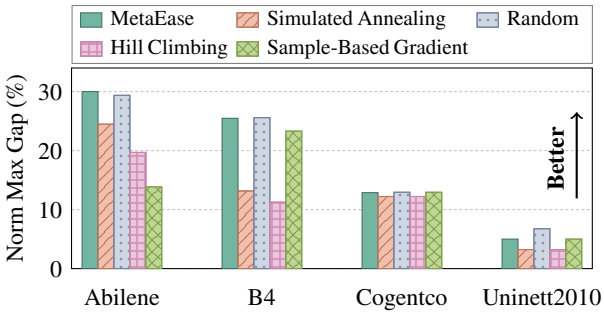


Figure 20: MetaEase delivers top-2 gaps compared to black-box searches when we analyze the example heuristic in §2.

users to control whether it should analyze the heuristic comprehensively. These options:

Prune equivalence classes. MetaEase can first evaluate the representative points KLEE returns for each equivalence class and remove those with no performance gap (it assumes those regions are probably areas where the heuristic performs well).

Prioritize equivalence classes. MetaEase normally runs gradient ascent in each equivalence class in parallel; but it also has an option where it orders the equivalence classes before it runs gradient ascent based on the performance gap of the representative point for that class — if we assume the representative point is a good estimate of how hard that region is for the heuristic, then this would allow MetaEase to prioritize search over regions that are likely to produce bad outcomes.

Early stop (timeouts). MetaEase also has a timeout feature that limits how long it searches each equivalence class.

G Evaluation Details

Fig. 20 shows the details of comparison of MetaEase with black-box approaches for the example heuristic in §2. MetaEase delivers top-2 gaps.

We also report the raw end-to-end runtime for MetaEase in Tab. 11 across all the problems and all the heuristics in the paper.

G.1 Extended MetaOpt Comparisons

For Demand Pinning [48], MetaOpt has longer runtimes than MetaEase. We run MetaOpt until convergence for Demand Pinning. MetaEase discovers performance gaps within 90% to 100% of what MetaOpt discovers but does so in significantly less time (up to 8 hours faster) across most topologies (Tab. 10). The only exception is on the Uninett2010 topology where MetaEase's gap is 65% of MetaOpt's.

H Seed Generation using LLM

To complement symbolic execution seeds, we also explored using large language models (LLMs) to automatically generate diverse seed inputs. The idea is to frame seed generation as a prompt-driven task: given the heuristic code and network topology, the LLM is asked to propose synthetic demand samples that exercise qualitatively different behaviors of the heuristic. Fig. 21 shows the prompt template we use for the `Demand Pinning` heuristic. Based on the results, naïvely prompting the LLM did not work. The LLM often produced trivial or redundant traffic demands. More work is needed to make this approach work.

Topology	**Δ Time (h)** (MetaOpt – MetaEase)	**Gap Ratio (%)** (MetaEase / MetaOpt)
Abilene	0.60	91.23%
B4	0.61	100.0%
Swan	0.46	100.0%
Uninett2010	2.23	63.4%
Cogentco	8.00	90.8%

Table 10: For Demand Pinning, MetaEase finds performance gaps comparable to MetaOpt, but in less time in most cases.

Prompt Template for Seed Generation

You are an expert in analyzing heuristics. You are given a WAN Traffic Engineering heuristic named `Demand Pinning`, which operates on the given topology (`TOPOLOGY`). This heuristic has a pinning threshold: for each demand, if the demand value is less than the threshold, it is routed through its first shortest path. For the rest of the demands, they are routed on residual capacities optimally.
Your task is to generate a diverse set of synthetic demand samples that explore the range of behaviors of this heuristic.

```
Demand Pinning Code:

// Insert Demand Pinning implementation here

TOPOLOGY:

// Insert JSON topology description here
```

Requirements:

1. **Diversity of Behaviors:** Each sample must be designed so that it triggers a different decision path or behavior. For example:
   - Highly skewed demand (one dominant demand).
   - Uniform low demands across all pairs.
   - Bottleneck saturation forcing rerouting/load balancing.
   - Edge cases: minimal vs. extremely high demand.
   - Sensitivity cases: adding/removing one demand flips the solution.
   - ...
2. **Number of Samples:** Generate at least 8–10 samples, each highlighting a distinct heuristic behavior.
3. **Topology Constraint:** Ensure that demands only use valid source/destination pairs from `TOPOLOGY`.

Output Formatting: Follow exactly the format below. Each sample should be a JSON object containing a list of source-destination pairs with their demand values.

```
// Example output format:
[
 "Sample_1": {
   {"src": "A", "dst": "B", "demand": 12},
   {"src": "B", "dst": "C", "demand": 5},
   ...
},
 ...
]
```

Figure 21: Prompt given to the LLM to generate seed demands for Demand Pinning.

Problem	Instance	Time (s)
TE: Demand Pinning §6.1	Abilene	230
	B4	5002
	Swan	599
	Uninett2010	7714
	Cogentco	1558
TE: POP §6.1	Abilene	4883
	B4	24528
	Swan	898
	Uninett2010	202320
	Cogentco	51480
TE: Heuristic in §2	Abilene	100
	B4	250
	Swan	500
	Uninett2010	1640
	Cogentco	5971
DOTE §6.1	Abilene	2478
VBP:FFD §6.2 (# items, # Dim)	10-1	2839
	10-2	215066
	15-1	5857
	15-2	8449
	20-1	71277
	20-2	280748
Knapsack §7.2 (# items)	20	8673
	30	10676
	40	12931
	50	15518
Arrow §7.2	B4	143098
	IBM	6907
Maximum Weight Matching §7.2	Abilene	108
	B4	1628
	Swan	200
	Uninett2010	1751
	Cogentco	2548

Table 11: MetaEase end-to-end execution times for different problems and heuristics.