Finding Adversarial Inputs for Heuristics using Multi-level Optimization

Pooria Namyar, *Microsoft and University of Southern California;* Behnaz Arzani and Ryan Beckett, *Microsoft;* Santiago Segarra, *Microsoft and Rice University;* Himanshu Raj and Umesh Krishnaswamy, *Microsoft;* Ramesh Govindan, *University of Southern California;* Srikanth Kandula, *Microsoft*

https://www.usenix.org/conference/nsdi24/presentation/namyar-finding

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Finding Adversarial Inputs for Heuristics using Multi-level Optimization

Pooria Namyar[†‡], Behnaz Arzani[†], Ryan Beckett[†], Santiago Segarra[†⋆],
Himanshu Raj[†], Umesh Krishnaswamy[†], Ramesh Govindan[‡], Srikanth Kandula[†]
[†]*Microsoft,* [‡]*University of Southern California,* [⋆]*Rice University*

Abstract– Production systems use heuristics because they are faster or scale better than their optimal counterparts. Yet, practitioners are often unaware of the performance gap between a heuristic and the optimum or between two heuristics in realistic scenarios. MetaOpt is a system that helps analyze these heuristics. Users specify the heuristic and the optimal (or another heuristic) as input, and MetaOpt encodes these efficiently for a solver to find performance gaps and their corresponding adversarial inputs. Its suite of built-in optimizations helps it scale to practical problem sizes. We used MetaOpt to analyze heuristics from three domains (traffic engineering, vector bin packing, and packet scheduling). We found a production traffic engineering heuristic can require 30% more capacity than the optimal in realistic cases. We modified the heuristic based on the patterns in the adversarial inputs MetaOpt discovered and reduced the performance gap by 12.5×. We examined adversarial inputs to a vector bin packing heuristic and proved a new lower bound on its performance.

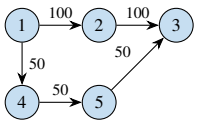
1 Introduction

Many solutions to network and systems problems are heuristic approximations to potentially intractable optimal algorithms [3, 20, 35, 38, 39, 53, 57, 58, 76]. These heuristics are often faster or scale better than their optimal counterparts. However, operators often do not fully understand how these heuristics will behave with new and untested inputs or how far from the optimal their results may drift in realistic use.

For example, Microsoft uses a heuristic, demand pinning (DP), on its wide-area network [46, 56]. It routes small demands (*i.e.,* demands $\leq$ a threshold) through their shortest path and uses a more computationally complex optimization to route the rest [38] (DP reduces the number of demands the optimization routes and completes faster). In Fig. 1, we show an example where DP allocates 40% less demand than the optimal routing. With this gap, Microsoft may have to either over-provision its network by 40%, delay 40%, or drop 40% of its customers' demands! Note that the 40% gap is a lower bound, and the worst-case gap can be higher.

We often do not understand the potential impact of such heuristics at scale: Does their gap depend on the problem size? Which inputs make them perform poorly? Are there realistic scenarios that they struggle with? We ask if we can develop techniques to *analyze heuristics* and answer such questions.

As a first step towards this goal, we have developed



Demand		DP (thresh = 50)		OPT	
src-dest	value	path	value	path	value
1-3	50	1-2-3	50	1-4-5-3	50
1-2	100	1-2	50	1-2	100
2-3	100	2-3	50	2-3	100
		Total DP	**150**	Total OPT	**250**

FIGURE 1: Suboptimal performance of DP. (left) Topology with unidirectional links. (right) A set of demands and their flow allocations using the DP heuristic and the optimal (H′) solution. DP first sends the demands at or below the threshold (50) over their shortest paths and then optimally routes the remaining demands.

MetaOpt, a system that allows users to *automatically discover the performance gap* between a heuristic H and any other function H′ for much larger problem sizes than in Fig. 1. MetaOpt also returns the *adversarial inputs* to these functions that cause large performance gaps. Users can use MetaOpt to (a) understand the performance gap of H relative to the optimal or to another heuristic; and (b) examine adversarial inputs to provide performance bounds on H or to modify H to improve its performance gap (§2).

In many problem domains, such as traffic engineering [38, 39, 46, 58], vector bin packing [35, 60], and packet scheduling [5, 64, 74], we can specify both H′ and H either as an optimization (with an objective and several constraints) or as a feasibility problem (with a set of constraints). Then, we can model the problem of finding large performance gaps between H and H′ in the language of optimization theory:

$$\underset{\text{s.t. input } \mathcal{I} \in \text{ConstrainedSet}}{\arg\max} \; \mathsf{H}'(\mathcal{I}) - \mathsf{H}(\mathcal{I}), \tag{1}$$

where H() and H′() take $\mathcal{I}$ as input and solve the corresponding algorithms. ConstrainedSet specifies a set of constraints that limit the set of values $\mathcal{I}$ can take.

In theory, we could throw this model at a solver [16, 36] and find performance gaps, but existing solvers do not support these optimizations. This model is an instance of a bi-level optimization [15] (with connections to Stackelberg games [23], see §6), and practitioners have to rewrite them into single-level optimizations before using a solver [15]. Rewriting a bi-level optimization into a single-level one by hand is tedious and can lead to poor performance if done incorrectly.

MetaOpt abstracts away this complexity and only asks users to specify H′ and H (§3). It also provides helper functions to make it easier to specify H′ and H. These functions are especially useful for constructs (*e.g.,* conditionals, randomization) that are harder to express as optimization constraints.

Under the hood, MetaOpt performs rewrites automatically and supports multiple solvers (Gurobi [36] and Zen [16]). We add three techniques to scale MetaOpt to large problems (*e.g.,* large topologies and demands for traffic engineering). First, since many rewrites can introduce non-linearities, we carefully select which part of the input to rewrite. Second, we introduce a new rewriting technique (Quantized Primal-Dual) that allows MetaOpt to trade-off between scale and optimality. Third, we design a new partitioning technique for graph-based problems to improve MetaOpts scalability further.

We show the versatility of MetaOpt[1] by using it to study several heuristics in traffic engineering [38, 39, 46, 58], vector bin packing [35, 60], and packet scheduling [5, 64, 74]. We have applied MetaOpt to (a) study performance gaps of these heuristics, (b) analyze adversarial inputs to prove properties, and (c) devise and evaluate new heuristics (see Table 1):

- DP can allocate 33% less demand compared to the optimal in large topologies. We analyzed adversarial inputs MetaOpt found and designed modified-DP which reduced this gap by an order of magnitude.
- We show for the first time that a two-dimensional vector bin packing heuristic FFDSum can require at least twice as many bins as the optimal *across all problem sizes*.
- A recently proposed programmable packet scheduler, SP-PIFO [64], an approximation of PIFO [64], can delay the highest priority packet by at least 3× relative to PIFO.

2 Heuristic Analysis at a Glance

Network and systems designers use heuristics when the optimum is too expensive to compute at relevant problem scales. We use examples of heuristics to motivate the types of analyses designers might wish to perform and describe how MetaOpt can aid these analyses.

2.1 Heuristics and their Importance

We describe heuristics from traffic engineering, cluster resource allocation, and switch packet scheduling.

Traffic engineering (TE). There are many techniques to scale TE solutions to large networks/demands (see §A.2 for details):

Demand Pinning (DP) [46, 56] is a heuristic that Microsoft uses in production. It pre-allocates flows along the shortest path for any node pair with demand smaller than a threshold T_d and uses the SWAN [38] optimizer on the rest. When many demands are small, this can result in substantial speedup.

Partitioned Optimization Problems (POP) [58] divides node pairs (and their demands) uniformly at random into partitions. It then assigns each partition an equal share of the edge capacities and solves the original problem (*e.g.,* the SWAN LP optimization [38]) once per partition. POP is faster than SWAN because it can solve each LP sub-problem much faster than the original [22] and can do so in parallel.

Vector bin packing (VBP). Production deployments use VBP to allocate resources in clusters efficiently [2, 35, 37, 53, 68]. One version of VBP takes a set of balls and bins with specific sizes and multiple dimensions (*e.g.,* memory, CPU, GPU [32, 48, 69]) and tries to pack the balls into the fewest number of bins. The optimal algorithm for this version is APX-hard [71]. Instead, many practitioners use a heuristic, first fit decreasing (FFD), which is greedy and iterative. At each step, FFD picks the unassigned ball with the *largest weight* and places it in the *first* bin that *fits* (has enough capacity). Prior works propose different ways to weigh the balls [35, 37, 53, 60, 66, 67, 72]. One variant, FFDSum, uses the sum across all dimensions as the weight of a ball.

Packet scheduling [5, 74]. Push-In-First-Out (PIFO [64]) queuing is a scheduling primitive that enables various packet scheduling algorithms for programmable switches. SP-PIFO [5] uses n priority FIFO queues to approximate PIFO and presents a heuristic we can implement at line rate. In SP-PIFO, each queue has a priority (*usually* equal to the priority of the last packet in it). A queue only admits a new packet if its priority is higher than the packet's priority. The algorithm scans queues from lowest to highest priority and places the new packet in the first queue that accepts it. SP-PIFO updates all queue priorities if no queue admits the packet.

Performance Analyses. Often, heuristic designers wish to answer questions such as:

- How far is my heuristic from the optimum?
- What inputs cause my heuristic's performance to degrade at practical problem instances?
- How can I redesign my heuristic to improve its performance?
- How can I compare the performance of two heuristics?
- Can I prove lower bounds on a heuristic's performance relative to another standard algorithm?

2.2 MetaOpt, a Heuristic Analyzer

MetaOpt is a heuristic analyzer that can help designers answer these questions. It finds (a) the *performance gap* between a given $\mathsf{H}()$ and an alternative $\mathsf{H}'()$ (where $\mathsf{H}'()$ can be the optimal solution) and (b) the adversarial inputs to $\mathsf{H}()$ that cause the performance gap. The performance of a heuristic measures its solution quality. Users decide what performance metric to use; for example, they can define the performance as the total flow that the heuristic admits in TE or the average packet delay in packet scheduling. The performance gap is the difference between the performance of $\mathsf{H}()$ and $\mathsf{H}'()$. MetaOpt can analyze a broad set of well-defined heuristics (see §3). We discuss its limitations in §5.

2.3 Using MetaOpt to Analyze Heuristics

Heuristic designers can use MetaOpt to answer the questions we presented in §2.1. They can use MetaOpt in two ways:

[1] Our code is available at https://github.com/microsoft/MetaOpt.

	MetaOpt finds performance gaps	MetaOpt helps prove properties	MetaOpt helps modify heuristics
TE	DP and POP can be 33.9% and 20% less efficient than optimal. MetaOpt finds realistic demands with strong locality that produce the same gap.	–	Modified-DP (which we designed with MetaOpt's help) has an order of magnitude lower gap than DP. We cannot improve the gap by running DP and POP in parallel.
VBP	MetaOpt finds tighter performance gaps under realistic constraints. It also finds the same examples which took theoreticians decades to find and prove a tight bound for 1d-FFD.	MetaOpt helped prove a tighter approximation ratio for 2-d FFDSum than previously known [60].	–
PIFO	MetaOpt finds SP-PIFO can delay the highest priority packet by 3×, and also finds inputs where AIFO incurs 6× more priority inversion than SP-PIFO.	MetaOpt helped prove a new bound on the weighted average delay of SP-PIFO.	Modified-SP-PIFO (which MetaOpt helped design) has 2.5× lower weighted average packet delay.

TABLE 1: MetaOpt (1) finds the performance gap between the heuristic and optimal; (2) helps prove various properties about the heuristic; and (3) helps modify them to improve their performance.

(a) to find performance gaps and (b) to prove properties or improve heuristics based on the adversarial inputs it finds. We next describe the results we obtained by applying MetaOpt to the heuristics in §2.1. We present more details and other results in §4. These use cases are not the only way operators can use MetaOpt, but they demonstrate its versatility (see §8).

Finding performance gaps. We show how MetaOpt helps find performance gaps in TE and packet scheduling.

Performance gaps in traffic engineering. We use MetaOpt to find the performance gaps for DP and POP, where $H'()$ is the optimal multi-commodity flow algorithm (§A.1). We measure the performance gap as the difference between the heuristic's and the optimal's total flow, normalized by the total network capacity. The performance gap is a lower bound on the *optimality gap*, the worst-case gap between the two.

We find DP and POP incur 33.9% and 20% relative performance gaps on a large topology (Cogentco, §4). This means there exists (and we can find) adversarial traffic demands that cause DP to use at least 33.9% more capacity than optimal. Network operators that use DP may need to over-provision the network by that much to satisfy this demand.

MetaOpt, by default, searches for adversarial inputs among all possible demands. We can constrain MetaOpt to search over *realistic* demands. These are sparse and exhibit strong locality, which means few node pairs that are close to each other exchange most of the traffic [3]. When we run MetaOpt with these constraints, the gap for POP and DP remains almost the same as before, but the discovered adversarial demands exhibit stronger locality and are sparser (*i.e.,* the adversarial demands that MetaOpt finds becomes more realistic).

Performance gaps in packet scheduling. We compare SP-PIFO to PIFO. We compute and compare the priority-weighted average packet delay (§4) between the two algorithms, which penalizes them if they increase the delay of high-priority packets. MetaOpt shows there exists an input packet sequence where SP-PIFO is 3× worse than PIFO.

We also use MetaOpt to compare SP-PIFO and AIFO [74] (two heuristics). AIFO emulates PIFO through a single FIFO queue and replaces $H()$ in this scenario. MetaOpt finds inputs for which AIFO incurs 6× more priority inversions than SP-PIFO. Such analyses can help designers weigh performance trade-offs against switch resource usage.

Proving properties of and improving heuristics. MetaOpt discovers performance gaps and the corresponding adversarial inputs. We show how to prove performance bounds for these heuristics or to improve them in several cases[2]. These require the user to *analyze the inputs further* to see if they share a common pattern (see §B.2 for an example).

A new bound for vector bin-packing. VBP heuristics try to minimize the number of bins they use. Theoreticians prove bounds on their approximation ratio: the worst-case ratio of the number of bins the heuristic uses compared to the optimal over any input. Recent work [60] showed 2-dimensional FFDSum asymptotically approaches an approximation ratio of 2 (where the optimal uses nearly infinite bins). We prove (§4) that the approximation ratio is always at least 2 — even when the optimal requires a finite number of bins!

A new bound and a better heuristic for packet scheduling. We analyzed adversarial inputs MetaOpt found for SP-PIFO and proved a lower bound on its priority-weighted average delay relative to PIFO. The bound is a function of the priority range and the number of packets.

Adversarial inputs to SP-PIFO trigger priority inversions, which means they cause SP-PIFO to enqueue high-priority packets behind low-priority ones. We tested a Modified-SP-PIFO that splits queues into groups; it assigns each group a priority range and runs SP-PIFO on each group independently. This modification reduces the possibility of packets with vastly different priorities interfering with each other and causing priority inversions. Modified-SP-PIFO reduces the performance gap of SP-PIFO by 2.5×. Users should weigh the trade-off between the improved performance gap, the

[2]One may have to sacrifice other metrics such as run-time to improve the gap. Users have to weigh this trade-off and decide if it is acceptable.

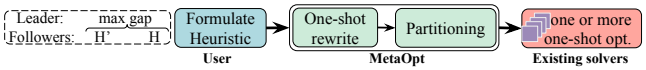


FIGURE 2: MetaOpt's workflow involves four steps; (1) User encodes H and H′ §3.2, (2) MetaOpt automatically applies rewrites to obtain a single-level optimization §3.3, (3) it partitions the problem into smaller subproblems to achieve scalability §3.5, (4) MetaOpt feeds the resulting optimizations into existing solvers [36, 55] and finds large performance gaps.

Optimal	Heur.	Formul.	Rewrite
OptMaxFlow	POP	Convex(Random) §A.3	KKT/PD
	DP	Convex(Conditional) §A.3	KKT/PD
VBP MILP	FFD	NonConv.(Greedy) §B.1	Feasibility
PIFO MILP	SP-PIFO	NonConv.(Priority) §C.1	Feasibility
	AIFO	NonConv.(Admission) §C.2	Feasibility

TABLE 2: Overview of the five heuristics we explored in this paper. We cover how to formulate the heuristics as optimizations in the appendix and discuss their rewrites as constraints in §3.3.

memory allocation, and the impact on ongoing traffic to decide if they should deploy Modified-SP-PIFO.

Improving traffic engineering heuristics. We found that DP performs poorly when small demands traverse long paths. We used MetaOpt to analyze a Modified-DP, which routes demands on their shortest paths if the shortest path is less than k hops and the demand is less than T_d. This simple change reduced the performance gap by an order of magnitude. Modified-DP presents a trade-off between the speed and the performance gap of the heuristic. Users can control this trade-off through k and T_d. MetaOpt can guide users in choosing these values by quantifying each choice's performance gap.

3 MetaOpt Design

Our goal is to build a *widely applicable* system that finds *large performance gaps* between H and H′ *quickly and at scale*.

3.1 MetaOpt Approach

This is a hard problem when H and H′ are arbitrary algorithms, but we observe that we can formulate many heuristics in networks and systems as:

Convex optimization problems in which the heuristic seeks to optimize an objective subject to a collection of constraints. DP and POP fall into this category: DP solves optimization for large demands, and POP solves optimization for each partition.

Feasibility problems in which the heuristic searches for a solution that satisfies a collection of constraints. FFDSum and SP-PIFO fall into this category: FFDSum packs balls into bins subject to weight and capacity constraints, and SP-PIFO places packets into the queues based on their priorities.

We can find the performance gap between any H′ and H through a bi-level or meta optimization (hence MetaOpt) as long as H falls in one of these two classes. We do not need convexity for H′ — we need it to be either an optimization or

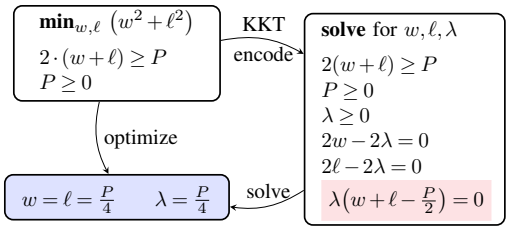


FIGURE 3: Rewrite using KKT in an example where we find a rectangle's optimal width w and length ℓ such that its perimeter is $\geq P$. The inner variables are w and ℓ. The right panel shows the feasibility problem using the KKT theorem. Equations with λ variables correspond to first-order derivatives of inequality constraints in the original problem. P is a variable of the outer optimization but is treated as a constant in the inner problem.

a feasibility problem. We model the problem as[3]:

$$\begin{aligned}
&\arg\max_{\mathcal{I}}\ \mathsf{H}'(\mathcal{I})-\mathsf{H}(\mathcal{I}) && \text{(leader problem)}\\
&\text{s.t.}\quad \mathcal{I}\in \mathsf{ConstrainedSet} && \text{(input constraints)}\\
&\mathsf{H}'(\mathcal{I})=\max_{\mathbf{f}'\in\mathcal{F}'}\mathsf{H}'_\mathsf{Objective}(\mathbf{f}',\mathcal{I}) && \text{(optimal)}\\
&\mathsf{H}(\mathcal{I})=\max_{\mathbf{f}\in\mathcal{F}}\mathsf{H}_\mathsf{Objective}(\mathbf{f},\mathcal{I}) && \text{(heuristic)}
\end{aligned}\qquad (2)$$

where the *leader* or *outer* optimization maximizes the difference between the two functions (*i.e.,* the performance gap) over a space of possible inputs $\mathcal{I}$. This leader problem is subject to *follower* or *inner* problems (H′ and H). We model the performance of H' on input $\mathcal{I}$ through H'_Objective. This function decides the values for the variables $\mathbf{f}'$, internally encodes problem constraints, and computes the overall performance of H′. H'_Objective treats the outer problem's variables, $\mathcal{I}$, as input and constant. We define H_Objective the same way.

Bi-level Optimization: A Brief Primer. Modern solvers do not directly support the style of bi-level optimizations we described in Equation 2 [36]. To solve these, users need to rewrite the bi-level optimization as a single-level optimization [15]. These rewrites convert an optimization problem into a set of feasibility constraints: if the inner problems are both optimizations, the rewrites will replace both H′() and H() with a set of feasibility constraints in the outer optimization. The resulting formulation is a single-level optimization that modern solvers [16, 36] can attempt to solve.

Fig. 3 shows an example of an inner problem and the corresponding rewrite. This somewhat contrived example finds a rectangle's optimal length and width subject to some constraints (the outer problem may want to optimize P). The rewrite uses the Karush–Kuhn–Tucker (KKT) theorem [22] and converts convex optimizations with at least one strictly feasible point [22] into feasibility problems. The theorem states any point that solves the new problem matches the solution of the original. We describe another technique in

[3] Note that we can transform minimization optimizations to maximization by negating their objective.

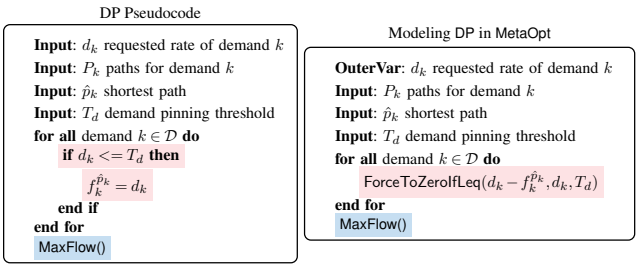


FIGURE 4: Modeling DP in MetaOpt. (see Table A.1 for notation)

§3.4, which exploits the Primal-Dual theorem [22]. Both of these rewriting techniques produce a single-level optimization equivalent to the bi-level formulation if the inner problems are convex [15]. The problem MetaOpt solves has properties that allow us to reduce the overhead of these rewrites (see §3.3).

3.2 MetaOpt: The User View

Inputs. MetaOpt could have asked the user to input the single-level formulation, which can be hard and error-prone. The rewritten single-level formulation can have an *order of magnitude* more constraints than the original bi-level formulation (§4) and is hard to optimize for practitioners who do not understand bi-level optimizations.

MetaOpt simply lets the user input H and H′ (Fig. 2). It then automatically produces a single-level formulation, optimizes the rewrites, feeds these into a solver of the user's choice (MetaOpt currently supports Gurobi [36] and Zen [16]), and produces performance gaps and adversarial inputs.

How to specify H or H′. It can be hard to describe H or H′ in the optimization language. When we modeled problems in MetaOpt, we observed certain constructs are common across many heuristics. We encode these as helper functions to make it easier for users less familiar with optimization theory to model their heuristics. The set of helper functions is not complete (there may be constraints for which we have not devised a corresponding helper function) but the interface is extensible and we can add new functions as needed.

For instance, a user who wants to compare the performance of DP and the optimal TE multi-commodity flow would have to specify the formulation of both of these algorithms. Standard textbooks describe the former [17], so we focus on the latter. DP involves a conditional (an `if` statement on the left of Fig. 4) where the outcome is determined based on the demands. These demand values are *variables* of the outer problem and unknown apriori, which means we need to determine the outcome of these conditionals as part of the optimization and model them as convex constraints.

We can use the big-M method [22] (§A.3) to convert this `if` statement into constraints optimization solvers support. MetaOpt provides a helper function, ForceToZeroIfLeq (see Fig. 4), to help users do this conversion. This level of indirection makes it easier for the user to specify DP and also

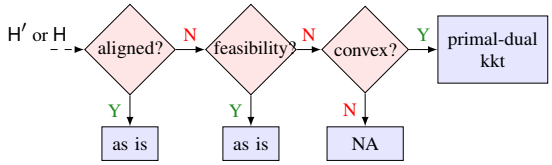


FIGURE 5: MetaOpt automatically converts the bi-level problem to a single-level optimization. It supports any follower, which either (1) is a convex optimization; (2) is a feasibility problem; or (3) has an objective that aligns with the outer problem.

gives MetaOpt the flexibility to optimize or change the formulations when needed. For example, the big-M method causes numerical instability in larger problems, and MetaOpt uses an alternate method to convert it to constraints (see §A.3).

Our helper functions (§D, Table A.8) codify common design patterns and help specify constraints across a diverse set of problems. We show how to use and combine them to model other heuristics, such as, FFD, which involves greedy decisions, and SP-PIFO, which involves dynamic priority updates. These helper functions encode succinct and readable models.

3.3 Automatic Rewrites

MetaOpt produces a bi-level optimization from H′ and H (Equation 2) and then automatically rewrites it. While the underlying theory behind rewrites is well-known [15], to our knowledge, there are no automated rewriters, and the process has required human intervention until now. We need to be careful when we automate rewrites. For example, one challenge is modeling non-linear constraints that involve the multiplication of variables. In the Primal-Dual rewrite (§3.4), the constraints in the dual depend on the type of optimization (maximization or minimization) and whether the corresponding primal variable is unconstrained, positive, or negative [22].

We have developed automatic rewrite techniques for KKT, Primal-Dual, and a new variant of the latter (*i.e.,* Quantized Primal-Dual §3.4). Users can choose which rewrite they use.

MetaOpt does not naïvely rewrite the bi-level formulation and only rewrites the inner problems if necessary. We call this technique *selective rewriting*. It avoids rewriting in two cases (Fig. 5): when the inner problem is a feasibility problem or when it is "aligned". In these cases, we can directly merge the inner problem's constraints into the outer problem and remove its objective if they have one.

An *aligned inner problem* is one where optimizing the outer problem's objective directly or indirectly optimizes the inner problem as well. We observe that the objective of MetaOpt is such that one of H′ or H is always aligned with the outer problem. The outer problem maximizes H′ and minimizes H to maximize the gap. This aligns with H′ if H′ is a maximization problem and with H when H is a minimization (both H and H′ solve the same problem, so they are either both maximizations or both minimizations).

For all other instances, we need to rewrite the inner problem. MetaOpt currently rewrites an inner problem only if it is an

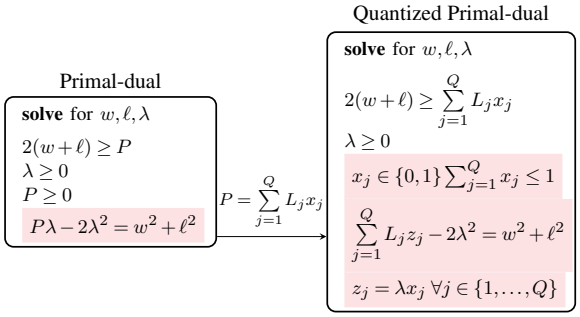


FIGURE 6: Left shows rewrite using primal-dual theorem. It has fewer constraints and different multiplicative terms ($P\lambda$ versus $\lambda(w+\ell-\frac{P}{2})$ in the KKT rewrite). On the right, we show how to quantize the parameters of the outer problem (P). The QPD rewrite no longer has any multiplicative terms since we can linearize the multiplication of binary (x_j) and continuous variables (λ). The quadratic terms (*e.g.,* w^2) is due to the quadratic objective of the original problem. L_js are constants.

unaligned convex optimization. KKT and Primal-Dual apply to these cases. MetaOpt also supports *unaligned non-convex* inner problems that can be written as a feasibility problem (*e.g.,*SP-PIFO and FFD). See §5 for other extensions.

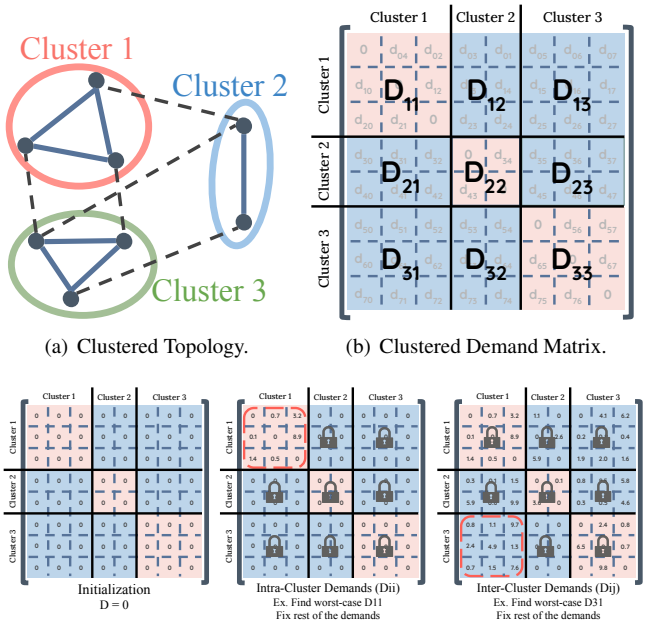
With all this, MetaOpt generates a single-level formulation that is equivalent to the bi-level optimization, preserves the theoretical properties, and scales well.

3.4 The Quantized Primal-Dual Rewrite

The KKT rewrite does not scale beyond small problems. It introduces multiplicative terms (pink highlighted constraint in Fig. 3) that commodity solvers support (special ordered sets in Gurobi [36] and disjunctions in Z3 [55]). However, these constraints with multiplicative terms slow down the solvers and dictate their latency.

A similar observation holds for the Primal-Dual rewrite [15]. It uses the strong duality theorem [22] to convert an optimization into a feasibility problem. According to this theorem, any feasible point of a convex problem is optimal iff the primal objective at that point is the same as the dual. Therefore, the Primal-Dual rewrite contains a constraint that ensures the primal and dual objectives are equal in addition to the primal and the dual constraints. Fig. 6 (left) shows the Primal-Dual rewrite for the optimization in Fig. 3. This rewrite can generally result in non-linear constraints that impact the scalability.

To scale, we have developed a technique called *Quantized Primal-Dual* (QPD) that converts the Primal-Dual rewrite into a simpler problem (see Fig. 6). In QPD, we replace the input P with $\sum_{j=1}^{Q} L_j x_j$ where L_js are constants we choose a priori and x_js are binary variables. We require $\sum_j x_j \leq 1$, which means the outer problem has to pick one of the $Q+1$ values $(0, L_1, \ldots, L_Q)$ for P. We only need to quantize the leader's variables that appear in the multiplicative terms of



(a) Clustered Topology. (b) Clustered Demand Matrix.

(c) MetaOpt's Clustering Method.

FIGURE 7: Partitioning in MetaOpt. We first find the demands that maximize the gap between H′ and H in each cluster. We then fix the demands within each cluster and, one by one, find the demands between pairs of clusters that increase the gap.

the Primal-Dual rewrite. The inner problem is still optimal under this rewrite, but we trade off the optimality of the outer problem for speed by quantizing the input space.

Two challenges remain: (a) determining the number of quanta and (b) picking the values (L_j). Using more quanta leads to more integer variables, slowing down the solver. Using fewer quanta results in lower-quality adversaries as we will limit the input to only a few pre-selected values.

Since QPD rewrites are much faster to solve than the other rewrites, we can sweep multiple quanta choices and pick the best. We use the exact KKT rewrite on smaller problems to find good candidates. We observe empirically that adversarial inputs occur at so-called extreme points. For example, the worst-case demands have either 0 or the maximum possible value in POP[4]. Although we do not have a formal proof, we conjecture that the intuition behind these observations is similar to the intuition behind the simplex theorem [18].

We evaluate how these rewrites impact the scalability and investigate the approximation gap of QPD in §4.

3.5 Partitioning to Scale MetaOpt

We have found it necessary to use more aggressive scaling techniques to analyze problem instances at practical scales, such as TE heuristics for realistic topologies and demands. One such technique is *partitioning*. We can partition any

[4]For DP, the worst-case demands take values of 0, the demand pinning threshold T_d, or the maximum possible demand.

Topology	#Nodes	#Edges	#Part.	DP	POP
Cogentco	197	486	10	33.9%	20.76%
Uninett2010	74	202	8	28.4%	20.15%
Abilene [65]	10	26	–	12.69%	17.31%
B4 [39]	12	38	–	13.16%	17.89%
SWAN [38]	8	24	–	2.29%	22.08%

TABLE 3: Details of the topologies used in §4.1 and discovered gap.

problem, but we show the key steps in Fig. 7 for the TE heuristics where the problem has an intrinsic graph structure.

First, we partition nodes in the underlying network graph into clusters and solve the rewritten single-level optimization on each cluster in parallel. In this step, we only consider the intra-cluster demands (the diagonal pink blocks in Fig. 7).

We then freeze the demands from the last step and find the demands between pairs of clusters that worsen the gap by solving the rewritten problem on each pair. This step iteratively fills the blue blocks of the demand matrix in Fig. 7.

We can parallelize the second step between cluster pairs with little overlap and produce an overall demand by adding the values MetaOpt discovers after invoking each optimization. This method speeds up MetaOpt because each individual optimization, whether per cluster or per cluster pair, is much smaller than the overall problem.

We empirically find this partitioning approach consistently discovers inputs with large performance gaps. This is because more than one adversarial input exists, and our partitioning method does not bias against them. For example, the adversarial inputs for DP follow a common pattern where demands between distant nodes are just below the threshold. For such inputs, the heuristic wastes the capacity of many links by routing the demands on their shortest paths. In contrast, the optimal routing allocates those link capacities to multiple demands between nearby nodes. Our partitioning method still allows MetaOpt to find many inputs with this pattern.

4 Evaluation

We apply MetaOpt to traffic engineering, vector bin packing, and packet scheduling heuristics to show its generality. MetaOpt helped us quantify and understand the performance gaps of heuristics, prove theoretical properties, and design heuristics with lower performance gaps. Table 1 summarizes our findings. We also show the importance of our optimizations in MetaOpt and quantify its speed and scalability.

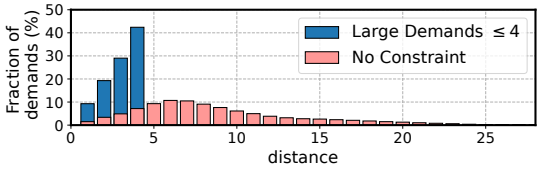
Implementation. Our prototype is in C# and uses Gurobi v9.5.2 [36]. We also have a port that uses Z3 [55]. To partition the graph, we adapt the previous code [3, 25] for spectral clustering [59] and FM partitioning [19, 24] and report results for different cluster numbers and clustering techniques.

4.1 Heuristics for WAN Traffic Engineering

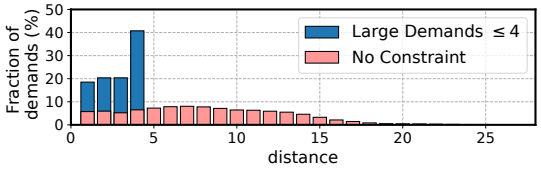
In this section, we (a) obtain performance gaps for DP and POP with respect to the optimal max-flow algorithm and (b) devise modified versions of these heuristics based on our analysis of their adversarial patterns.

Heu	Additional Constraints on $\mathcal{I}$	Density	Gap
DP	–	54.06%	33.9%
	locality (distance of large demands ≤ 4)	12.03%	33.4%
POP	–	16.14%	20.76%
	locality (distance of large demands ≤ 4)	4.74%	20.70%

(a) Impact of adding locality constraints on gap and density.



(b) Impact of adding locality constraints on DP



(c) Impact of adding locality constraints on POP

FIGURE 8: Using MetaOpt to find practical adversarial inputs on Cogentco. We can find more local and sparser adversarial inputs by constraining the input space.

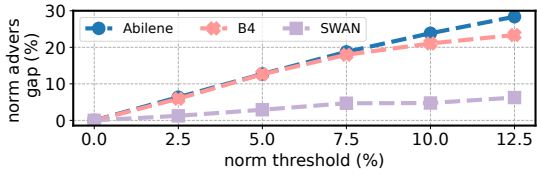
Experiment Setup. We use K-shortest paths [73] to find the paths between node pairs ($= 4$ if unspecified). We constrain the demands to be below a maximum value (half the average link capacity if unspecified) to ensure they are realistic and a single demand does not create a bottleneck. For DP, we vary the demand pinning threshold (=5% of average link capacity if unspecified). For POP, we vary the number of partitions (=2 if unspecified) and report the average gap over 5 random trials (see §A.3). We report runtimes on an AMD Opteron 2.4GHz CPU (6234) with 24 cores and 64GB of memory and use all available threads (unless mentioned otherwise). We timeout each optimization after 20 minutes.

Topologies. We use two large topologies from [1] and three public production topologies [38, 39, 65] (Table 3).

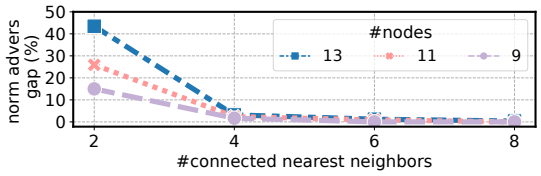
Metrics. We normalize the performance gap by the sum of the link capacities so we can compare across different scales.

Finding performance gaps. We compare DP and POP to the optimal max-flow on topologies that range from 8 to nearly 200 nodes (Table 3). We use the QPD rewrite (§3.4) and partitioning (§3.5) for most experiments but do not need the partitioning technique for small topologies (SWAN, B4, and Abilene). DP's performance gap ranges from 2% to over 33%. POP also exhibits a large performance gap (up to 22%).

These performance gaps are over any possible input demand. We can also use MetaOpt to obtain performance gaps on realistic inputs. Production demands are sparse and exhibit strong locality [3]. We can express these properties in MetaOpt through constraints on the input space ($\mathcal{I}$ in Equa-

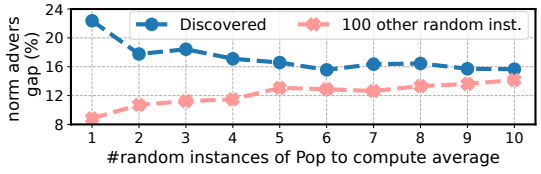


(a) Gap vs. the threshold value for DP.

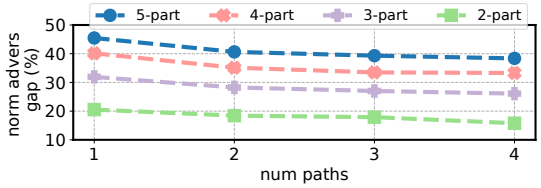


(b) Gap vs. connectivity.

FIGURE 9: DP's performance gap increases with the threshold and decreases with the connectivity.



(a) Gap vs. instances to approximate the expected value.

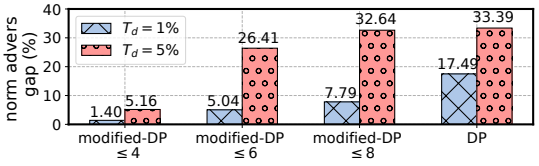


(b) Gap vs. #paths and #partitions for avg POP.

FIGURE 10: POP's performance gap when varying (a) #instances to approximate average and (b) #paths and #partitions.

Heuristic	Distance	Threshold wrt avg link cap
DP	–	0.1%
modified-DP	≤ 6	1% (10x)
	≤ 4	5% (50x)

(a) Maximum threshold such that discovered gap $\sim 5\%$.



(b) DP vs. modified-DP

FIGURE 11: We propose modified-DP based on insights from MetaOpt. It only pins small demands between near nodes, is more resilient, and pins more demands with the same gap.

tion 2). The gaps for DP and POP on those inputs are only slighly lower than in the unconstrained case, but we find adversarial demands that are sparser and more local (Fig. 8).

Designing heuristics with lower performance gap. We can use the adversarial inputs from MetaOpt to design new heuristics or explore whether we can combine heuristics to improve their gap. We first describe how we identified patterns in the adversarial inputs of DP and POP and then show how we used these patterns to improve these heuristics' performance gap.

Adversarial input patterns for DP. Intuitively, DP's performance gap increases as we increase its threshold since the heuristic forces more demands on their shortest path. Yet, the gap grows faster on some topologies even though they have roughly the same #nodes, #edges, and diameter.

We used synthetic topologies to study DP. To create each topology, we start with a ring graph and then connect each node to a varying number of its nearest neighbors. The results (Fig. 9(b)) indicate that the performance gap grows with the (average) shortest path length (fewer connections across nearest neighbors = longer shortest paths). Intuitively, if the shortest path lengths are longer on average, DP will use the capacity on more edges to route the small demands. This reduces the available capacity to route the rest of the demands.

Adversarial input patterns for POP. Since POP is a random heuristic, we search for inputs that maximize the expected gap. We approximate this expectation by an empirical average over n random partition samples. Then, we check whether the adversarial inputs can generalize by testing them on 100 other random instances. When MetaOpt uses a small number of samples to estimate the expected gap, the adversarial inputs overfit. We can improve its generalization by increasing the number of samples but at the cost of scalability. We find $n = 5$ permits scaling without overfitting (Fig. 10(a)).

POP's performance gap increases as we increase the number of paritions because each partition (1) gets a smaller fraction of each edge's capacity, and (2) has less information about the global state. We can reduce this gap by increasing the number of paths as it helps each partition better allocate the fragmented capacity.

Modified-DP. DP has a higher performance gap when nodes have a larger average distance. We use this insight to modify DP and design a heuristic with a lower gap. Modified-DP only pins demands that are (a) smaller than a threshold and (b) *between nodes less than k hops apart* (user specifies k). This heuristic routes small demands between distant nodes optimally, leaving more capacity for other demands. Its gap is $12.5\times$ smaller than DP for $T_d = 1\%$ and $k = 4$.

Increasing the distance threshold in modified-DP allows for better scalability by pinning more demands but at the cost of a higher performance gap. MetaOpt can help users adjust the parameter k based on their needs.

Modified-DP has another benefit. We can use a higher demand threshold (10 - $50\times$) and maintain the same gap as the original DP. We show this by using MetaOpt to compute the maximum threshold each method can admit while having a

max #balls	ball size granularity	FFD($\mathcal{I}_{\text{MetaOpt}}$)
20	0.01×	8
20	0.05×	7
14	0.01×	7

TABLE 4: MetaOpt finds slightly tighter bounds when constraining the number and size of balls. For OPT($\mathcal{I}$)=**6**, the tightest known theoretical bound for FFD [30] is **8**. This assumes the input can have unlimited balls of any size.

OPT($\mathcal{I}$)	MetaOpt		theoretical bound [60]	
	#balls	approx ratio	#balls	approx ratio
2	6	2.0	4	1.0
3	9	2.0	12	1.33
4	12	2.0	24	1.5
5	15	2.0	40	1.6

TABLE 5: MetaOpt finds adversarial examples with tighter approx. ratio for 2d-FFD than the best known theoretical bound [60].

gap $\leq 5\%$ (Fig. 11(a)). Operators can leverage this to pin more demands when small demands exhibit strong locality [3].

Meta-POP-DP. This meta-heuristic runs POP and DP in parallel and selects the best solution for each input. The two heuristics appear to have distinct adversarial inputs: DP underperforms when distant pairs have small demands, and POP when large demands that go through the same link end up in the same partition. We expected combining them would reduce the performance gap significantly compared to each one. But MetaOpt shows the new heuristic only improves the performance gap by 6% on the Cogentco topology. It finds inputs where small demands are between distant pairs (adversarial to DP) and large demands are between nearby nodes that end up in the same partition (adversarial to POP).

4.2 Heuristics for Vector Bin Packing

We use MetaOpt to (a) derive performance gaps that verify known results for FFD in one dimension and (b) prove a new property for FFD in 2 dimensions.

Finding performance gaps. Performance in FFD is measured by the number of bins required to fit a given number of balls. After decades of theoretical studies on 1d-FFD [14, 30, 43, 52], the work in [30] established the tight bound $\text{FFD}(\mathcal{I}) \leq \frac{11}{9}\text{H}'(\mathcal{I}) + \frac{6}{9}$ for any $\mathcal{I}$ (H′ is the optimal). To prove tightness, the authors craft a careful example where $\text{H}'(\mathcal{I}) = 6$ bins and $\text{FFD}(\mathcal{I}) = 8$. MetaOpt found the same example when we constrained its inputs to $\text{H}'(\mathcal{I}) = 6$ and proved FFD needs 8 bins in the worst-case.

[30] assumes (1) $\mathcal{I}$ can have an unlimited number of balls and (2) the balls in $\mathcal{I}$ can have any size (even $0.00001cm^3$!). However, when packing jobs (balls) in machines (bins) [68], we often know *a priori* an upper bound on the number of jobs or the quantization levels for resource requirements. We can incorporate such constraints and ensure MetaOpt finds practical performance gaps. As Table 4 shows, when we constrain the number of balls and the ball sizes, MetaOpt finds adversarial inputs that produce a tighter bound compared to [30].

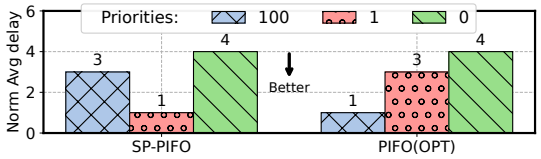


FIGURE 12: SP-PIFO can delay the highest priority packet (rank = 0) by 3×. We show the average delay of packets with the same priority normalized by the average delay of the highest-priority packets in PIFO. We assume packets have priorities between 0 - 100, and the queues can admit all the packets (similar to SP-PIFO). When 10K packets arrive at the same time, and the queues drain at 40 Gbps, the average delay for the highest priority packets in PIFO is $0.74ms$ (the performance gap in this figure is independent of the number of packets).

MetaOpt max objective	#priority inversions	
	SP-PIFO [5]	AIFO [74]
AIFO() − SP-PIFO()	6	37
SP-PIFO() − AIFO()	24	11

TABLE 6: Using MetaOpt to compare two heuristics. We show the number of priority inversions on an 18-packet trace from MetaOpt. The total queue size is 12 and SP-PIFO has 4 queues.

Proving properties. While multi-dimensional FFD is widely used in practice [35, 37, 53], its theoretical guarantees are less well understood. Recently, [60] crafted an example where 2-dimensional FFDSum uses α times more bins than the optimal. $\alpha \in [1, 2)$ and $\alpha \to 2$ as the optimal tends to infinity. In other words, α is strictly less than 2 for a finite size problem.

MetaOpt found adversarial inputs with $\alpha = 2$ *for every problem size we considered* (Table 5). For example, when the optimal uses 4 bins, MetaOpt finds an adversarial input with 12 balls causing FFDSum to use 8 bins. In constrast, [60] uses 24 balls and only achieves approximation ratio of 1.5.

We studied the adversarial inputs from MetaOpt and proved the following theorem, establishing an approximation ratio of at least 2 for FFDSum. §B.2 contains the detailed proof.

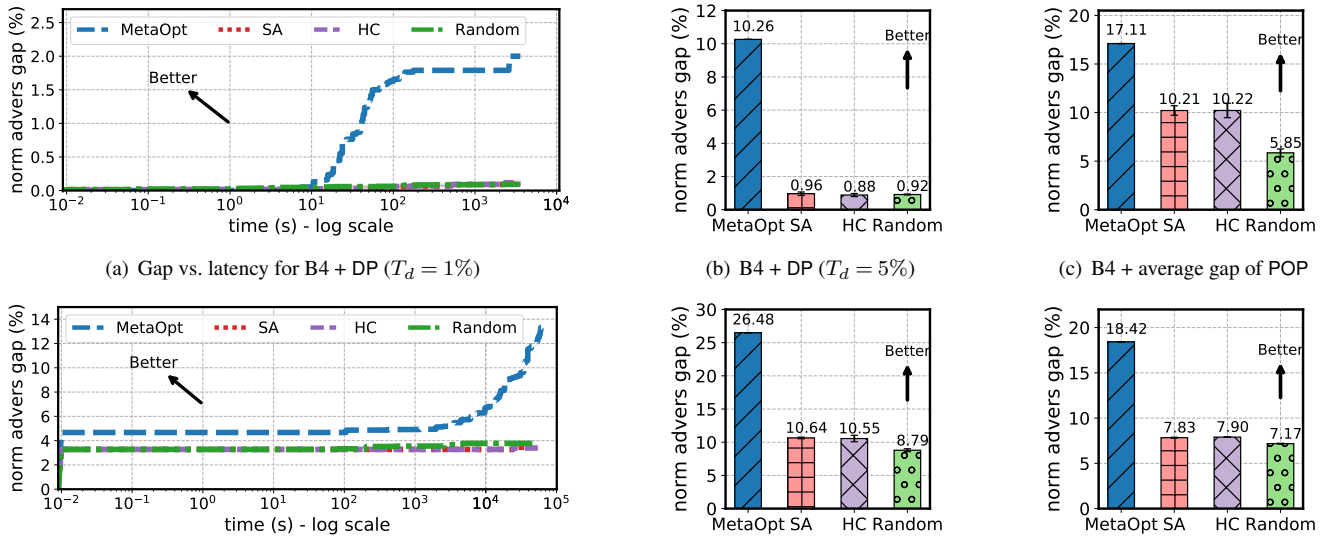
Theorem 1. *In 2-dimensional VBP, for any $k > 1$, there exists an input $\mathcal{I}$ with $OPT(\mathcal{I}) = k$ and $FFDSum(\mathcal{I}) \geq 2k$.*

4.3 Heuristics for Packet Scheduling

We use MetaOpt to compare the performance of SP-PIFO with both optimal (PIFO [64]) and another heuristic (AIFO [74]). Unlike SP-PIFO, AIFO uses a single queue but adds admission control based on packet priorities to approximate PIFO.

Finding performance gaps. We use the average delay of packets weighted by their priorities to compare PIFO and SP-PIFO. MetaOpt discovers packet traces where SP-PIFO fails to prioritize packets correctly and incurs 3× higher delays for high-priority packets than PIFO (Fig. 12).

We also use MetaOpt to compare SP-PIFO and AIFO. Unlike SP-PIFO, AIFO is designed for shallow-buffered switches, and its admission control can drop packets. For a fair compar-



(a) Gap vs. latency for B4 + DP ($T_d = 1\%$)

(b) B4 + DP ($T_d = 5\%$)

(c) B4 + average gap of POP

(d) Gap vs. latency for Cogentco + DP ($T_d = 1\%$)

(e) Cogentco + DP ($T_d = 5\%$)

(f) Cogentco + average gap of POP

FIGURE 13: MetaOpt is faster and finds larger gaps between OPT and POP or DP on Cogentco and B4. We report the gap relative to the total capacity and use only one thread to run each method for fair comparison (SA = Simulated Annealing, HC = Hill Climbing).

ison, we assume both heuristics use the same switch buffer size, and we split the buffer evenly across SP-PIFO queues. With limited buffers, these algorithms may drop packets, so we need to consider the impact of their respective drop rates when comparing their performance. We borrow a metric from SP-PIFO: we count k priority inversions when a packet is inserted in a queue after k lower priority packets (even if the queue is full and the packet would have been dropped). We found (Table 6):

AIFO sometimes outperforms SP-PIFO because (1) it has one large queue instead of n smaller ones – SP-PIFO drops many packets when faced with a burst of packets with the same priority since it assigns them to a single smaller queue; and (2) SP-PIFO lacks admission control – we can create an adversarial pattern where lower-priority packets arrive right before a group of high-priority ones to make SP-PIFO admit the lower-priority packets and drop the higher-priority ones.

But SP-PIFO also sometimes outperforms AIFO because (1) AIFO lacks a sorting mechanism, which can cause high-priority packets to get delayed behind lower-priority ones, and (2) AIFO depends on an estimate of the distribution based on the most recent window of packets. MetaOpt found traces in which a few packets with entirely different priorities compared to others can disrupt AIFO's distribution estimate!

Proving properties. MetaOpt shows that SP-PIFO's adversarial inputs exhibit significant priority inversions. We used these inputs to prove a lower bound on the worst-case performance gap, in terms of priority-weighted average delay, between SP-PIFO and PIFO (see §C.3):

Theorem 2. *For any number of packets $N \geq 1$, integer priorities between $0 - R_{max}$ and $q \geq 2$ queues, there exists a sequence of packets $\mathcal{I}$ where the difference between the weighted average packet delay that results from SP-PIFO is*

$$(R_{max} - 1)(N - 1 - p)p \quad \textit{where} \quad p = \lceil (N-1)/2 \rceil \qquad (3)$$

worse compared to PIFO.

Designing better heuristics. SP-PIFO uses the same set of queues to schedule packets with a wide range of priorities. We found it underperforms when the difference between packet priorities is large. Theorem 2 also confirms this, as the gap is proportional to R_{max}.

We evaluated a Modified-SP-PIFO, in which we limited the range of packet priorities that can compete with each other: we formed m queue groups where each group served a fixed priority range. SP-PIFO runs on queues within a group. This modified version can reduce the gap of SP-PIFO by $2.5\times$.

4.4 Evaluating MetaOpt

We show MetaOpt can find solutions faster than other baseline search methods and helps users describe H′ and H in a compact way. We also show how our various design choices help.

How fast MetaOpt discovers a performance gap. Alternatives to MetaOpt include random search, hill climbing [29], and simulated annealing [45]. Random search repeatedly picks new random inputs and returns the one with a maximum gap. Hill climbing (HC) and simulated annealing (SA) use information from past observed inputs to guide the search (more details in §E).

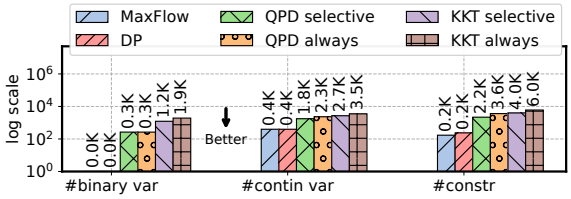


FIGURE 14: Users specify DP and OPT in MetaOpt. We show the complexity of these specifications and the rewrites in terms of the number of variables and constraints (see Fig. A.2 for POP).

MetaOpt finds 1.7× - 17× larger gaps than the next best (Fig. 13). The baselines fail since they ignore the heuristic's details and treat it as a black box. MetaOpt uses its knowledge of the heuristic and the topology to guide the search.

MetaOpt is the only method that consistently discovers substantially larger gaps over time, while other techniques get stuck in local optima and improve only slightly even after many hours (Fig. 13(a), 13(d)).

Input and rewrite complexity. Users specify H′ and H, and MetaOpt automatically applies selective rewrites (§3.3) to scale better. We evaluate how complex these specifications and rewrites are regarding the number of binary variables, continuous variables, and constraints (Fig. 14). In general, solvers perform better if these quantities are lower.

We use DP as an example to highlight three features in MetaOpt's design (Fig. 14). The three metrics show the user's inputs are more compact than the rewritten optimization; they have a fifth of the constraints and half the number of continuous variables. This quantifies how MetaOpt's automatic rewrites can reduce the user's burden. Selective rewrites are important: we can use them to reduce the number of constraints (2.2K vs. 3.6K for QPD) and continuous variables (1.8K vs. 2.3K for QPD) compared to when we always rewrite the bi-level optimization. We can produce more compact specifications (with fewer variables and constraints) through QPD compared to KKT, even with selective rewrites. This explains why it helps MetaOpt scale.

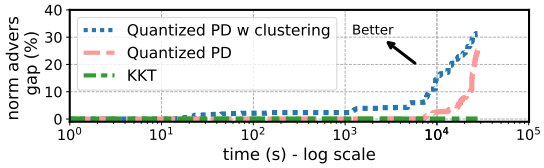
The impact of partitioning. MetaOpt partitions the problem to find larger gaps faster than both (non-partitioned) quantized primal-dual and KKT, even on medium-sized topologies (Fig. 15(a)). For larger topologies, KKT and primal-dual cannot find large gaps without partitioning.

As we increase the number of partitions, MetaOpt scales better and finds larger gaps until it eventually plateaus (10 for Cogentco in Fig. 15(b)). We can slightly improve the gap if we double the solver timeout.

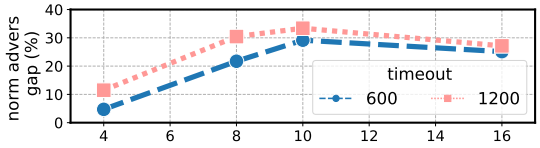
The inter-cluster step in partitioning is important, especially for heuristics that underperform when demands are between distant nodes (DP in Fig. 15(c)). The partitioning algorithm also impacts the discovered gap (Fig. 15(d)).

The impact of quantization. To quantify, we compare the relative difference between the gap from quantized primal-dual and KKT (which does not use quantization). We found quantization has little impact on solution quality: 4% for DP and 0 for POP on B4 (we can not scale KKT to larger topologies). For POP, we use two quantiles: 0 and the max demand. If a demand d in an uncongested partition falls in between these values, forcing the demand to zero cannot decrease the gap: the rewrite's throughput would drop by d, and the optimal throughput by some value between 0 and d. A similar argument applies to the congested case.

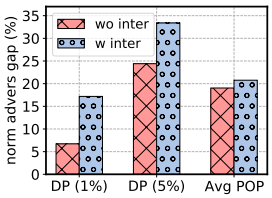
We use three quantiles for DP: 0, the threshold, and the max demand. For a high enough threshold, quantized primal-dual may avoid assigning the threshold value to demands between distant nodes to not violate capacity constraints, whereas KKT can assign any value, which causes the relative difference in solution quality.



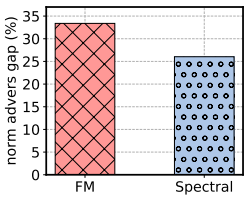
(a) KKT vs Quantized primal-dual vs w partitioning on Uninett2010



(b) Number of partitions and solver timeout on Cogentco



(c) Inter-cluster on Cogentco



(d) Graph partitioning on Cogentco

FIGURE 15: Partitioning helps MetaOpt find larger gaps faster.

5 Discussion

We have shown MetaOpt applies to various heuristics in multiple domains. We have developed a simple partitioning technique to scale MetaOpt to large problem instances (with partitioning, it focuses on finding large performance gaps instead of the worst-case). We defer the following to future work:

Application to a broader set of heuristics. MetaOpt only applies to heuristics, which we can model as convex or feasibility problems. A number of heuristics in systems and networking do not fit in either of these two categories (e.g., mixed integer *maximizations or minimizations* [76] that we cannot cast as a feasibility problem). Our approach in MetaOpt has roots in the theory of Stackelberg games and goes beyond bi-level

optimization. We discuss how the theory in this space can help extend MetaOpt's scope in §6.

Finding adversarial settings. The set up (*e.g.,* topology in DP) influences the performance gap as well. Users can model this aspect as part of MetaOpt and find adversarial problem settings. This may impact MetaOpt's scalability.

Ease of use. In MetaOpt, users have to write their heuristics in the optimization language, which requires expertise. MetaOpt provides a set of helper functions to simplify the process. However, there is room for improvement in enabling users to model their heuristics and extending MetaOpt to *explain why* a heuristic underperforms automatically.

6 Extending MetaOpt's scope

We observe the dynamics of the problem MetaOpt addresses resembles leader-follower games (Stackelberg equilibria [49]). In such games, a leader maximizes their payoff and controls the inputs to one or more followers. With this *fixed* input from the leader, the followers have to respond and optimize for their own payoff and decide outputs, which the leader does not control but influences their payoff.

These games apply to a variety of leader-follower combinations (*e.g.,* optimization-based and Bayesian) and there are various techniques to find the equilibrium in such games [21, 44, 70, 75]. Hence, in theory, we can use these techniques to analyze the performance gaps of a broader class of heuristics than MetaOpt currently supports, as long as we have a leader-follower combination where we know how to compute the equilibrium. This is future work.

7 Related Work

To the best of our knowledge, no prior work finds *provable* adversarial inputs for heuristics that approximate optimal problems. Our techniques (e.g., big-M, convex rewrites, and generally translating the problem to one that is amenable to off-the-shelf solvers) are not per-se novel [8, 26, 31], and prior work in networking have used some of these theories [6, 11, 12, 42]. However, no other work has combined them in this way. We also extend them to randomized, conditional, and sequential non-convex heuristics. Without our changes, we could not apply existing solvers directly or quickly find large gaps.

Our qualitative results – the optimality gap and hard examples for POP, DP, FFD, SP-PIFO, and AIFO – are novel. We also find and prove tighter bounds for the optimality gap of FFD and SP-PIFO.

Our work is different from most prior techniques. Traditional algorithmic worst- or average-case analyses [27, 54] are specific to an individual heuristic and must be applied case by case. We cannot do such analyses for some heuristics as they only find loose bounds or do not account for realistic input constraints (none exist for DP, POP, or [63, 76]).

Verification methods seek inputs that violate an *statically-specified safety or correctness* invariant on a given function [41]. In contrast, we look for inputs that maximize the *performance* gap.

Model checking approaches based on SMT solvers [4, 7, 9, 10, 34] can search for adversarial inputs that result in performance gaps greater than a fixed bound when users can encode both the optimal and heuristic as pure feasibility problems. However, these approaches cannot handle bi-level optimization, where the optimal or heuristic must be formulated as optimizations (*e.g.,* traffic engineering).

Local search algorithms [29, 45] apply to any (potentially black-box) heuristic or optimal algorithm. However, the flip side of such generality is that they are slow on large input spaces, get stuck in local optima, and fail to find practical inputs because they ignore the inner workings of the heuristic.

Recent work finds malicious inputs to learned techniques [33, 51]. However, none of these find provably large gaps or even consider the optimal algorithm. Other broadly related work include [13, 28, 50, 61].

Our partitioning approach is different from [3, 58]. NCFlow and POP need to return a *feasible solution* given an input (*e.g.,* one that respects capacity and path constraints). As a result, they have to combine the solutions from all partitions to ensure their feasibility. This makes them complex when they have to enforce global constraints. For example, POP [58] sacrifices quality and partitions demands separately to ensure the sub-problems enforce rigorous constraints. MetaOpt does not need such constraints because it generates an input for the problem, not the solution. Notice a certain 'coming full circle' aspect here: we use a similar (but not the same) partitioning to analyze the optimality gap of POP quickly.

This paper is an extended version of [56]. Compared to this workshop paper, we changed the methodology to improve generality and scalability, added helper functions for ease of use, added support for heuristics from VBP and packet scheduling, and did a more extensive evaluation.

8 Conclusion

MetaOpt is a heuristic analyzer for heuristics that can be posed as an optimization or a feasibility problem. It can be used to find performance gaps at scale, prove lower bounds on worst-case gaps, and devise improvements to heuristics. At its core, MetaOpt solves a bi-level optimization problem. To do this efficiently, it selectively rewrites heuristic specifications as a single-level optimization, and incorporates several scaling techniques. Future work can include using it to evaluate and improve other heuristics, increasing its expressivity (§6), and identifying infeasible inputs instead of adversarial ones.

Acknowledgments. We thank our shepherd, Marco Chiesa, and the anonymous reviewers for their insightful comments. We also thank Siva Kakarla, Rodrigo Fonseca, Jeff Mogul, Jay Lorch, and Daniel Berger for their helpful feedback. This material is based upon work supported in part by the U.S. National Science Foundation under grant No. CNS-1901523.

References

[1] Internet Topology Zoo. http://www.topology-zoo.org/.

[2] Yarn resource allocation of multiple resource-types. https://bit.ly/3YMDL2Z.

[3] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *NSDI*, 2021.

[4] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Automating network heuristic design and analysis. In *HotNets*, 2022.

[5] 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.

[6] David Applegate and Edith Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 313–324, 2003.

[7] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *NSDI*, 2023.

[8] Bryan Arguello, Richard L. Chen, William E. Hart, John D. Siirola, and Jean-Paul Watson. Modeling bilevel program in pyomo. https://www.osti.gov/servlets/purl/1526125.

[9] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *SIGCOMM*, 2022.

[10] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *SIGCOMM*, 2021.

[11] Behnaz Arzani, Alexander Gurney, Bo Li, Xianglong Han, Roch Guerin, and Boon Thau Loo. Fixroute: A unified logic and numerical tool for provably safe internet traffic engineering. *arXiv preprint arXiv:1511.08791*, 2015.

[12] Behnaz Arzani, Nicholas Iodice, Steven Hwang, Prahalad Venkataramanan, Roch Geurin, and Boon Thau Loo. Sunstar: A cost-effective multi-server solution for reliable video delivery. *arXiv preprint arXiv:1812.00109*, 2018.

[13] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. Surgeprotector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *SIGCOMM*, 2022.

[14] Brenda S Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 1985.

[15] Y. Beck and M. Schmidt. A Gentle and Incomplete Introduction to Bilevel Optimization. https://optimization-online.org/?p=17182, July 2023.

[16] Ryan Beckett and Ratul Mahajan. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 8–15, 2020.

[17] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Englewood Cliffs, 1992.

[18] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

[19] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks, 2008.

[20] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *SIGCOMM*, 2019.

[21] Branislav Bosansky and Jiri Cermak. Sequence-form algorithm for computing stackelberg equilibria in extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

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

[23] Wu chang Feng, Francis Chang, Wu chi Feng, and Jonathan Walpole. Provisioning on-line games: A traffic analysis of a busy counter-strike server.

[24] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev.*, 2004.

[25] Aaron Clauset. Fast Modularity Community Structure Inference Algorithm. https://bit.ly/3aAVGQH.

[26] Benoît Colson, Patrice Marcotte, and Gilles Savard. Bilevel programming: A survey. *4OR*, 2005.

[27] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[28] Anthony Corso, Robert Moss, Mark Koren, Ritchie Lee, and Mykel Kochenderfer. A survey of algorithms for black-box safety validation of cyber-physical systems. *Journal of AI Research*, 2021.

[29] L. Davis. Bit-climbing, representational bias, and test suit design. *Proc. Intl. Conf. Genetic Algorithm*, pages 18–23, 1991.

[30] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is $ffd(i) \leq 11/9opt(i) + 6/9$. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies: First International Symposium, ESCAPE*. Springer, 2007.

[31] Pablo Garcia-Herreros, Lei Zhang, Pratik Misra, Erdem Arslan, Sanjay Mehta, and Ignacio E Grossmann. Mixed-integer bilevel optimization for capacity planning with rational markets. *Computers & Chemical Engineering*, 86:33–47, 2016.

[32] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[33] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *HotNets*. ACM, 2019.

[34] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. Quantitative verification of scheduling heuristics. *arXiv preprint arXiv:2301.04205*, 2023.

[35] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[36] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[37] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: Vm allocation service at scale. In *OSDI*, 2020.

[38] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, and Min Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[40] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *SIGCOMM*, 2016.

[41] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, 2009.

[42] Chuan Jiang, Sanjay Rao, and Mohit Tawarmalani. Pcf: provably resilient flexible routing. 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*, pages 139–153, 2020.

[43] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[44] Jan Karwowski, Jacek Mańdziuk, and Adam Żychowski. Sequential stackelberg games with bounded rationality. *Applied Soft Computing*, 132:109846, 2023.

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

[46] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *NSDI*, 2022.

[47] Alok Kumar et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM*, 2015.

[48] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms, 2022.

[49] Tao Li and Suresh P Sethi. A review of dynamic stackelberg game models. *Discrete & Continuous Dynamical Systems-B*, 22(1):125, 2017.

[50] Zinan Lin, Hao Liang, Giulia Fanti, and Vyas Sekar. Raregan: Generating samples for rare classes. *arXiv preprint arXiv:2203.10674*, 2022.

[51] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. (self) driving under the influence: Intoxicating adversarial network inputs. In *HotNets*. ACM, 2019.

[52] Yue Minyi. A simple proof of the inequality $ffd(l) \leq 11/9opt(l) + 1$, $\forall l$, for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinca*, 1991.

[53] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *OSDI*, 2022.

[54] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[55] 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.

[56] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. Minding the gap between fast heuristics and their optimal counterparts. In *HotNets*, 2022.

[57] 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 *NSDI*, 2024.

[58] 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 *SOSP*, 2021.

[59] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2002.

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

[61] Pedro Reviriego and Daniel Ting. Breaking cuckoo hash: Black box attacks. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[62] Thomas Sauerwald. Sorting networks. https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/advalg.pdf.

[63] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with Shoofly. In *SIGCOMM*, 2021.

[64] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Stanford University IT. Abilene core topology, 2015.

[66] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and Distributed Computing*, 2010.

[67] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *WWW*, 2007.

[68] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014.

[69] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, 2015.

[70] Jiali Wang, He Chen, Rujun Jiang, Xudong Li, and Zihao Li. Fast algorithms for stackelberg prediction game with least squares loss. In *International Conference on Machine Learning*, pages 10708–10716. PMLR, 2021.

[71] Gerhard J. Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Inf. Process. Lett.*, 1998.

[72] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[73] Jin Y. Yen. Finding the K Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

[74] 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*, SIGCOMM '21, page 179–193, New York, NY, USA, 2021. Association for Computing Machinery.

[75] Yunxiao Zhang and Pasquale Malacaria. Bayesian stackelberg games for cyber-security decision support. *Decision Support Systems*, 148:113599, 2021.

[76] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. Arrow: Restoration-aware traffic engineering. In *SIGCOMM*, 2021.

Term	Meaning
$\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}$	Sets of nodes, edges, demands, and paths
N, M, K	Number of nodes, edges, and demands, i.e., $N = \lvert\mathcal{V}\rvert, M = \lvert\mathcal{E}\rvert, K = \lvert\mathcal{D}\rvert$
c_e, p	c_e: capacity of edge $e \in \mathcal{E}$ path p: set of connected edges
(s_k, t_k, d_k)	The kth element in $\mathcal{D}$ has source and target nodes ($s_k, t_k \in \mathcal{V}$) and a non-negative volume (d_k)
$\mathbf{f}, f_k^p$	$\mathbf{f}$: flow assignment vector with elements f_k f_k^p: flow for demand k on path p

TABLE A.1: Multi-commodity flow problems' notation.

A Details of Traffic Engineering

Table A.1 summarizes our notation.

A.1 Multi-commodity flow problem

The optimal form of WAN-TE typically involves solving a multi-commodity flow problem. Given a set of nodes, capacitated edges, demands, and pre-chosen paths per demand, a flow allocation is feasible if it satisfies demand and capacity constraints. The goal is to find a feasible flow to optimize a given objective (e.g., total flow [3], max-min fairness [38,39,57], or utility curves [47]). We define the feasible flow over a pre-configured set of paths as (see Table A.1)

$$\begin{aligned}
&\mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \{\mathbf{f} \mid && && && (4)\\
&f_k = \sum_{p\in\mathcal{P}_k} f_k^p, && \forall k \in \mathcal{D} && \text{(flow for demand } k\text{)}\\
&f_k \le d_k, && \forall k \in \mathcal{D} && \text{(flow below volume)}\\
&\sum_{k,p\mid p\in\mathcal{P}_k, e\in p} f_k^p \le c_e, && \forall e \in \mathcal{E} && \text{(flow below capacity)}\\
&f_k^p \ge 0 && \forall p \in \mathcal{P}, k \in \mathcal{D} && \text{(non-negative flow)}\}
\end{aligned}$$

Among all the feasible flows, the optimal solution seeks to maximize the total flow across the network:

$$\begin{aligned}
\mathsf{OptMaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq &\arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k && (5)\\
&\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}).
\end{aligned}$$

A.2 More details on DP and POP heuristics

Demand Pinning (DP) [46]. First, it routes all demands at or below a predefined threshold T_d through their shortest path. It then jointly routes the rest of the demands optimally over multiple paths:

$$\mathsf{DemandPinning}(\mathcal{D},\mathcal{P}) \triangleq \{\mathbf{f} \mid f_k^p = \begin{cases} d_k & \text{if } p \text{ is shortest path in } P_k \\ 0 & \text{otherwise} \end{cases}, \forall k \in \mathcal{D} : d_k \le T_d\}, \quad (6)$$

We can pose DP as an optimization with constraints that route demands below the threshold on the shortest paths:

$$\begin{aligned}
\mathsf{DemPinMaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq &\arg\max_{\mathbf{f}} \sum_{k\in\mathcal{D}} f_k && (7)\\
&\text{s.t.} \quad \mathbf{f} \in \mathsf{FeasibleFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P})\\
&\quad\ \ \mathbf{f} \in \mathsf{DemandPinning}(\mathcal{D},\mathcal{P})
\end{aligned}$$

Partitioned Optimization Problems (POP). [58] POP divides node pairs (and their demands) uniformly at random into partitions, assigns each partition an even share of edge capacities, and solves the original problem (*e.g.,* the SWAN optimization [38]) in parallel, once per partition.

$$\mathsf{POPMaxFlow}(\mathcal{V},\mathcal{E},\mathcal{D},\mathcal{P}) \triangleq \bigcup_{\text{part. c}} \mathsf{OptMaxFlow}(\mathcal{V},\mathcal{E}_c,\mathcal{D}_c,\mathcal{P}), \quad (8)$$

where $\cup$ is the vector union, the per-partition demands $\mathcal{D}_c$ are disjoint subsets of the actual demands drawn uniformly at random, and the per-partition edge list $\mathcal{E}_c$ matches the original edges but with proportionally smaller capacities.

A.3 Formulation of DP and POP

Demand Pinning formulation for quantized demands. DP has conditional or *if* clauses: if a demand is smaller than the threshold T_d, then route it over its shortest path; otherwise, use the optimal algorithm to route it.

We describe the shortest path for demand k using $\widehat{p}_k$ and write the *if* clause as:

$$\widehat{f_k^{p_k}} \ge \sum_{q=1}^{Q} \mathbb{1}\left[L_q \le T_d\right] L_q x_q^k, \ \forall k \in \mathcal{D}. \quad (9)$$

The $\{0, L_1, \ldots, L_Q\}$ are the quantas and x_q^ks are binary variables that pick which quanta is active for demand k. The term $\mathbb{1}[L_q \le T_d]$ is constant at runtime and only shows which terms exist in the sum.

Note that $d_k = \sum_q L_q x_q^k$ by the definition of quantization. Thus, if the demand d_k is smaller than the threshold T_d, Equation 9 will ensure that the allocation on the shortest path is equal to d_k.[5]

Demand Pinning big-M formulation. We can also encode DP using the standard big-M approach from optimization theory. This formulation does not require quantized demands and is useful for the KKT rewrite. However, big-M can cause numerical instability at scale.

[5]When $d_k \le T_d$, Equation 9 effectively becomes $\widehat{f_k^{p_k}} \ge d_k$. In the converse case ($d_k > T_d$), Equation 9 becomes $\widehat{f_k^{p_k}} \ge 0$ (a no-op).

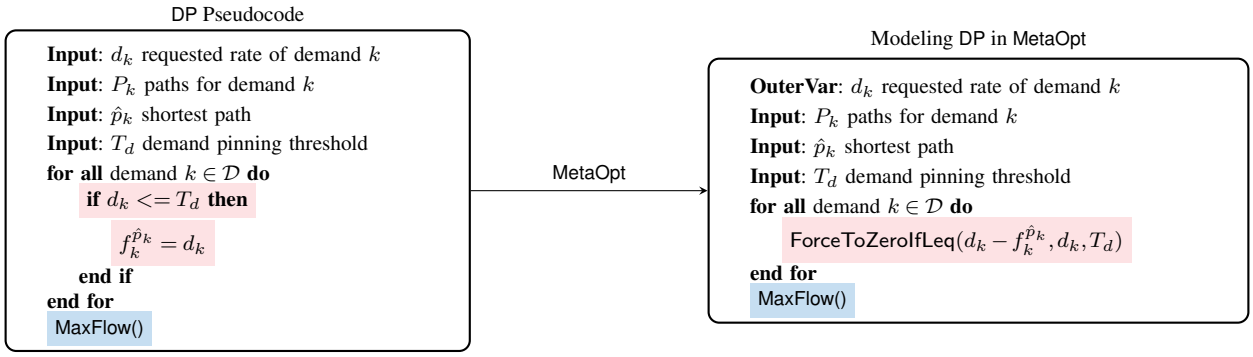


FIGURE A.1: The pseudocode for DP and how users can model it in MetaOpt using the helper functions.

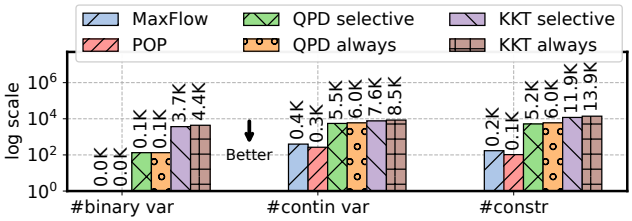


FIGURE A.2: The complexity of user's input for POP and the subsequent rewrites in terms of the size of the optimization.

$$\sum_{p \in P_k,\, p \neq \widehat{p_k}} f_k^p \leq \max\left(M(d_k - T_d), 0\right),\ \forall k \in \mathcal{D},$$

$$d_k - f_k^{\widehat{p_k}} \leq \max\left(M(d_k - T_d), 0\right),\ \forall k \in \mathcal{D},$$

where M is a large pre-specified constant. Notice that whenever demand d_k is below the threshold T_d, the constraints allocate zero flow on all but the (default) shortest path — DP routes the *full demand* on the default path in such cases. We can use standard optimization theory to convert the max in these constraints into a set of linear constraints [22] (this requires us modifying the objective but does not impact the final solution's quality).

Partitioned Optimization Problems Formulation. POP is convex as it is the union of solutions to disjoint linear optimizations (Equation 8). It is hard to encode POP as it uses random partitions, which makes POP ($\mathcal{I}$) a random variable in the leader problem, but MetaOpt needs a deterministic representation of the heuristic. We can consider a specific instance of each random variable (*e.g.,* one random assignment of demands to partitions). However, it will overfit to that instance and not reflect POP's true behavior.

We use the *expected value* or *tail percentile* of the gap from multiple random trials. To compute the average, we replace H in Equation 2 with its expected value and approximate the expectation through empirical averages over a few randomly generated partitions (see §4.1). To find the tail, we use a sorting network [40, 62] to compute the desired percentile across multiple random trials.

In addition, we encode an advanced version of POP in §A.4 that splits large demands across multiple partitions instead of assigning each demand to one partition.

A.4 POP Client Splitting

In §2, we introduce the (basic) POP heuristic [58], which incorporates *resource splitting* for our WAN TE problem, and in §A.3, we present POP as a convex optimization. The work in [58] also specifies an extended full-fleshed version of POP that incorporates "*client splitting*". We next show how to express this variant as a convex optimization problem.

We can think of POP client splitting as an operation that takes in a set of demands $\mathcal{D}$ and returns a modified set $\mathcal{D}_{\text{cs}} = \text{ClientSplit}(\mathcal{D})$ that can then be input into POP as in (8). The function ClientSplit() generates several duplicates of the existing demands and reduces their volume in proportion. It performs several operations where it replaces $(s_k, t_k, d_k) \in \mathcal{D}$ with two elements of the form $(s_k, t_k, d_k/2)$. It iterates and repeats this operation until it terminates (see [58]).

We encode a version of client splitting where we split an element in $\mathcal{D}$ if its demand value d_k is larger than or equal to a threshold d_{th}, and we keep splitting it until either we get to a predefined number of maximum splits (of the original demand[6]) or the split demand is lower than d_{th}.

Without loss of generality, we describe this idea for a single demand d_1: we can replicate this process for all demands in $\mathcal{D}$. Take for example the scenario where we split the demand at most twice (which creates at most 4 virtual clients). We a-priori encode all the flow variables for all possible splits of d_1: we use seven variables of the form $f_{i,j}$ instead of the single f_i where f_{1_1} is the flow if we do not split the client; $f_{1,2}$ and $f_{1,3}$ are the flows if we split the client once and $f_{1,4}$ through $f_{1,7}$ are the flows if we split the client twice. These

[6]Notice that [58] pre-specifies a total aggregated number of splits across all clients whereas we set the a maximum for per-client splits. This slight modification facilitates the convex representation of the heuristic.

Term	Meaning
i, j, d	indexes for ball, bin and dimension
$\mathbf{Y}_i, \mathbf{C}_j$	Multi-dim vectors of ball and bin sizes
W_i	Weight of ball i
α_{ij}	$= 1$ if ball i is assigned to bin j and 0 otherwise
$\mathbf{x}_{ij}$	Vector of resources allocated to ball i in bin j
f_{ij}	$= 1$ if ball i can fit in bin j and 0 otherwise

TABLE A.2: Our notation to formulate FFD as a feasibility problem. We use bold to indicate multi-dimensional vectors and capitalize variables, which are typically constants for FFD, but can be variables of the outer problem in MetaOpt.

flows have to satisfy:

$$0 \le f_{1,1} \le d_1,$$
$$0 \le f_{1,i} \le \frac{d_1}{2}, \text{ for } i \in \{2,3\}$$
$$0 \le f_{1,i} \le \frac{d_1}{4}, \text{ for } i \in \{4,5,6,7\}.$$

The variable $f_{1,1}$ should be zero unless $d_1 < d_{\text{th}}$ (when we do not split clients), which we can achieve using big-M constraints (§A.3):

$$f_{1,1} \le \max(M(d_{\text{th}} - d_1), 0).$$

We want $f_{1,2}$ and $f_{1,3}$ to be exactly zero unless $d_1 \ge d_{\text{th}}$ and $d_1/2 < d_{\text{th}}$, which we can achieve by doing

$$f_{1,i} \le \max(M(d_1 - d_{\text{th}} + \epsilon), 0), \quad \text{for } i \in \{2,3\},$$
$$f_{1,i} \le \max(M(d_{\text{th}} - d_1/2), 0), \quad \text{for } i \in \{2,3\},$$

where we added the small pre-specified $\epsilon > 0$ to allow for the case where $d_1 = d_{\text{th}}$. Lastly, we want $f_{1,4}$ through $f_{1,7}$ to be exactly zero unless $d_1 \ge 2d_{\text{th}}$. We encode this as:

$$f_{1,i} \le \max(M(d_1 - 2d_{\text{th}} + \epsilon), 0), \quad \text{for } i \in \{4,5,6,7\}.$$

We can replicate this procedure for all d_k and encode POP with client splitting as a convex optimization problem. Once this is done, the techniques in §3.4 apply.

B Details of Vector Bin Packing

B.1 Formulation of FFD (First-Fit-Decreasing)

We formulate the first-fit-decreasing heuristic as a feasibility problem (a set of constraints and no objective). Such a formulation allows the bi-level optimization in MetaOpt to become a single-level optimization without a rewrite (§3.3). To our knowledge, this formulation of FFD is novel.

Modeling FFD using MetaOpt's helper functions. Fig. A.3 (right) shows how users can easily model FFD without having to go through the mathematical details. It also shows the mapping between the helper functions and the pseudocode using different colors.

Details of how we model FFD as an optimization. Table A.2 lists our notation. The model uses binary variables. It is not a scalable method to solve FFD in practical systems, and we propose it only as an effective method to find adversarial inputs for the FFD heuristic.

	Bin index $j \longrightarrow$					
Fit f_{ij}	0	0	1	0	1	1
RHS of Equation 11	0	$\frac{1}{2}$	1	$\frac{2}{4}$	$\frac{4}{5}$	$\frac{4}{6}$
α_{ij}	0	0	1	0	0	0

TABLE A.3: Illustrating how we model *first-fit* in Equation 11.

Modeling *decreasing* ball weights: Recall that FFD places the unassigned ball with the largest weight in each iteration. We can use a sorting network [62] to ensure we pick balls in decreasing order. Instead, we propose a simpler alternative:

We observe the ball-weighting functions are a fixed function of the ball size. Let $\mathbf{Y}$ be a multi-dimension vector that captures the size of the i'th ball on each dimension. Then, the weight of the i'th ball, W_i, is $\sum_d Y_i^d$ in FFDSum [66], $\prod_d Y_i^d$ in FFDProd [72] and Y_i^1/Y_i^2 in FFDDiv [67] respectively.[7]

We constrain the input space (ConstrainedSet in Equation 2) to ensure that we assign balls in decreasing order of their weight if we assign them based on their index:

$$W_i \ge W_{i+1} \ \forall \text{item } i \tag{10}$$

Modeling *first-fit*: FFD assigns each ball to the first bin that has enough capacity. Let bins be ordered by their index and α_{ij} be a binary variable with value of 1 iff bin j is the first bin (i.e., the one with the smallest index) that has enough capacity for ball i. We model the first-fit constraint as:

$$\alpha_{ij} \le \frac{f_{ij} + \sum_{\text{bin } k<j}(1 - f_{ik})}{j} \quad \forall \text{item} i, \forall \text{bin } j \tag{11}$$

$$\sum_{\text{bin } j} \alpha_{ij} = 1 \quad \forall \text{item } i \tag{12}$$

Table A.3 shows an example that illustrates this constraint in action. It is easy to prove that the right-hand-side of Eqn. 11 is 1 for the first bin where the ball can fit (i.e., smallest index in set of bins $\{j \| \text{ fit } f_{ij} = 1\}$) and less than 1 for all other bins. The second constraint is necessary to ensure that $\alpha_{ij} = 1$ for exactly the first-fitting bin for each ball.

Modeling resource allocation and capacity constraints: We first ensure our allocation is consistent with the ball assignment: we allocate sufficient resources only from the assigned bin (not any other bin). We can do this simply by: $\mathbf{x}_{ij} \triangleq \mathbf{Y}_i \alpha_{ij}$. Here, the resource assigned to a ball i at bin j ($\mathbf{x}_{ij}$) is simply the product of the ball size vector $\mathbf{Y}$ with the assignment indicator variable α. But $\mathbf{Y}$ is a variable of the outer (or leader) problem, so such equations are non-linear. To linearize, we use a technique similar to what is colloquially known as big-M in optimization literature. Let Z be an

[7] The FFDDiv function applies only to two dimensions.

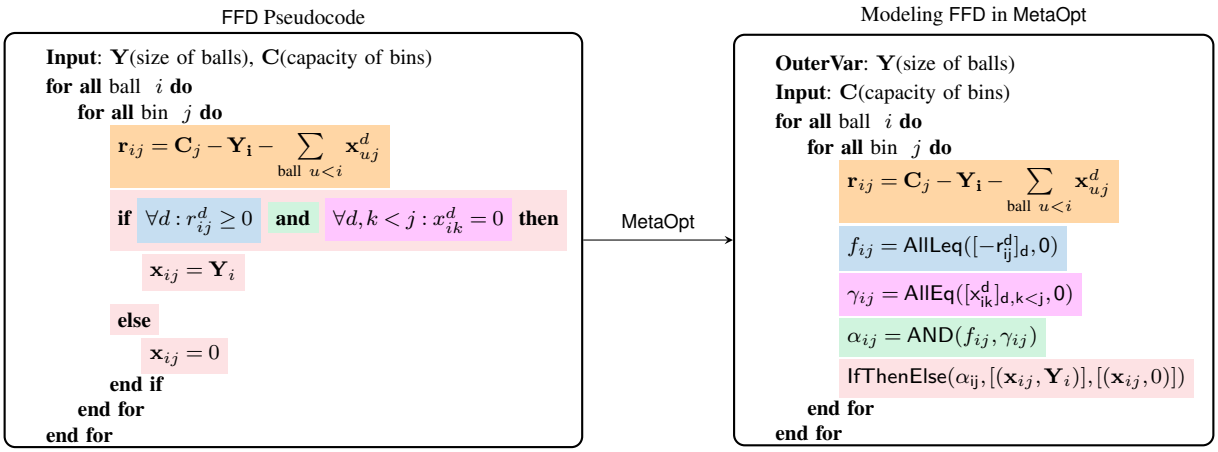


FIGURE A.3: The pseudocode for FFD and how users can model it in MetaOpt using the helper functions.

appropriately large postive constant, then:

$$x^d_{ij} \leq Z\alpha_{ij} \quad \forall \text{item } i \; \forall \text{bin } j \; \forall \text{dim } d \tag{13}$$

$$\sum_{\text{bin } j} x^d_{ij} = Y^d_i \quad \forall \text{item } i \; \forall \text{dim } d \tag{14}$$

We define the residual capacity of bin j after placing ball i in it as (remember we allocate ball i only after we place all the balls with a lower index):

$$\mathbf{r}_{ij} \triangleq \mathbf{C}_j - \mathbf{Y}_i - \sum_{\text{balls } u<i} \mathbf{x}_{uj} \quad \forall \text{item } i \; \forall \text{bin } j \tag{15}$$

The sum on the right captures how much resources we have already allocated to other balls from this bin. We next ensure f_{ij} is 1 iff the bin j has adequate resources to fit ball i. Let M be some appropriately large positive constant:

$$\min_d r^d_{ij} \leq Mf_{ij} \leq M + \min_d r^d_{ij}, \quad \forall \text{item } i \; \forall \text{bin } j \; \forall \text{dim } d \tag{16}$$

Here, if ball i fits in bin j, the residual capacity r^d_{ij} should be greater than 0 across all dimensions d. Therefore, Equation 16 clamps Mf between a positive number and M plus that positive number (remember r_{ij} is the remaining capacity of bin j *after* placing ball i in it).[8] Since f is a binary variable, the only feasible assignment in this case is 1. Conversely, if the ball does not fit in a bin, the residual capacity r^d_{ij} is below 0 on at least one dimension d and the constraint in Eqn. 16 clamps Mf to be between a negative number and M plus that negative number which forces f to be 0. In practice we set the value of M to be larger than the largest single-dimension bin capacity (*i.e.*, $\max_{j,d} C^d_j$).

[8] Corner case: when the residual capacity is precisely 0 on all dimensions, we want f to still be 1, but these constraints will allow f to be 0 as well. This is a rare case, but it can occur in practice. We can solve this corner case in a few different ways, including adding a small value ϵ to the left-most term in Equation 16.

Unique solution for FFD: The Equations 11–16 uniquely specify a solution to the iterative first-fit-decreasing heuristic. These constraints are linear even if the ball and bin sizes are variables in the outer problem. This is key because MetaOpt can apply without having to rewrite the heuristic follower.

Counting the number of bins: In this case, MetaOpt seeks inputs that cause the heuristic to use more bins than the optimal. To find such adversarial inputs, the outer (leader) problem needs the number of bins used by FFD:

$$\text{Num. bins used by FFD} \triangleq \sum_{\text{bin } j} \max_{\text{ball } i} \alpha_{ij}. \tag{17}$$

The term simply counts bins that have at least one ball. This is linear (max has a linear rewrite) and does not give rise to any additional concerns.

B.2 Proof of Theorem 1

Our goal is to show that for any $k > 1$, an input $\mathcal{I}$ exists where $\mathsf{FFDSum}(\mathcal{I})$ needs at least $2k$ bins while $\mathsf{OPT}(\mathcal{I})$ only needs k. Since we are proving a lower bound on the approximation ratio of FFDSum, it suffices to show an example for each k. We do this by the following; for every value of $k > 1$, we can find m and p such that $k = 2m + 3p$ and $p \in \{0, 1\}$. Then, we create an example consisting of $6m + 9p$ balls where $\mathsf{FFDSum}(\mathcal{I}) = 2\mathsf{OPT}(\mathcal{I}) = 2k$. We show the constructed example in Table A.4 along with the allocation from OPT and FFDSum.

C Details of Packet Scheduling Heuristics

We describe how we model SP-PIFO [5] and AIFO [74] as feasibility problems. These formulations are to the best of our knowledge novel. Table A.5 lists our general notations, and Table A.6 and Table A.7 show our specific notations for SP-PIFO and AIFO respectively.

Definition (Ranks and Priorities). Packet scheduling papers [5, 74] use both ranks and priorities: a packet with a

Ball ID	Ball Size	Wei ght	Num	Bin ID (OPT)	Bin ID (FFD)
1	[0.92, 0.00]	0.92	m	B1	B1
2	[0.91, 0.01]	0.92	m	B2	B2
3	[0.48, 0.20]	0.68	$\times p$	C1	C1
4	[0.68, 0.00]	0.68		C2	C2
5	[0.52, 0.12]	0.64		C3	C1
6	[0.32, 0.32]	0.64		C3	C2
7	[0.19, 0.45]	0.64		C2	C3
8	[0.42, 0.22]	0.64		C1	C3
9	[0.10, 0.54]	0.64		C1	C4
10	[0.10, 0.54]	0.64		C2	C5
11	[0.10, 0.53]	0.63		C3	C6
12	[0.06, 0.48]	0.54	m	B2	B1
13	[0.07, 0.47]	0.54	m	B1	B2
14	[0.01, 0.53]	0.54	m	B1	B3
15	[0.03, 0.51]	0.54	m	B2	B4

TABLE A.4: Constructed example to prove the approximation ratio of 2d-FFDSum is always lower bounded by 2 for any value of $\mathsf{OPT}(\mathcal{I}) > 1$.

higher rank has a lower priority and vice-versa. If a packet has rank R_p, and R_{max} is the maximum possible rank, we can compute the the packet's priority by $R_{max} - R_p$. This ensures that all the packets with a rank lower than R_p have higher priority values, and all those with a rank higher than R_p have lower priority values.

C.1 Formulation of SP-PIFO

SP-PIFO approximates PIFO [64] using n strict priority FIFO queues. It keeps a packet rank for each queue (*i.e.,* queue rank) that shows the lower bound on packet ranks the queue admits. For each packet, it starts from the lowest priority queue until it finds the first queue that can admit the packet (packet rank $\geq$ queue rank). If a queue admits the packet, SP-PIFO adds the packet to the queue and updates the queue rank to the recently admitted packet's rank (*i.e.,* push up). If none of the queues admit the packet (packet rank $<$ highest-priority queue rank), it reduces the rank of all the queues such that the highest-priority queue can admit the packet (*i.e.,* push down). Fig. A.4 (left) shows the pseudocode for SP-PIFO.

Modeling SP-PIFO using MetaOpt's helper functions. Fig. A.4 (right) shows how users can easily model SP-PIFO without having to go through the mathematical details. It also shows the mapping between the helper functions and the pseudocode using different colors.

Modeling push down. SP-PIFO reduces the rank of all the queues if none of the queues can admit the packet. This happens when the rank of the highest priority queue is higher than the packet rank (R_p). We model this as:

$$\hat{l}_q^p = l_q^{p-1} + max(0, l_N^{p-1} - R_p) \tag{18}$$

This constraint keeps the queue ranks the same if the packet

Term	Meaning
P, p	Number of packets and index for packet
R_{max}, R_p	Maximum rank and Rank of packet p
w_p	Weight of packet p
d_{pj}	$= 1$ if packet p dequeued after j, o.w. $= 0$

TABLE A.5: Our notation for formulating packet scheduling heuristics as feasibility problems. We capitalize variables which are typically constants for heuristic but can be variables of the outer problem in MetaOpt.

Term	Meaning
N, q	number of queues, and index for queue
$\mathbf{l}^{p-1}$	vector of queue ranks when deciding for packet p
$\hat{\mathbf{l}}^p$	vector of queue ranks after push down for packet p
x_{pq}	$= 1$ if packet p is in queue q, o.w. =0

TABLE A.6: Additional Notations for SP-PIFO.

rank R_p is greater than the highest priority queue l_N^{p-1}. Otherwise, it applies push down and reduces the rank of all the queues so that the highest priority queue can admit the packet (after the update, the rank of highest priority queue N is the same as packet rank R_p).

Deciding on the proper queue. Recall SP-PIFO adds a packet to the queue with the lowest priority among the ones that can admit the packet; that is the q that admits the packet ($R_p \geq \hat{l}_q^p$) but the one lower priority queue $q-1$ does not admit the packet ($R_p < \hat{l}_{q-1}^p$). We model this as following:

$$M x_{pq} \leq M + R_p - \hat{l}_q^p \qquad \forall \text{packet}\, p\, \forall \text{queue}\, q \tag{19}$$

$$M x_{pq} \leq M + \hat{l}_{q-1}^p - R_p - \epsilon \qquad \forall \text{packet}\, p\, \forall \text{queue}\, q \tag{20}$$

$$\sum_q x_{pq} = 1 \qquad \forall \text{packet}\, p \tag{21}$$

where M is a large constant ($\geq R_{max}$) and ϵ is a small constant (< 1). The first constraint ensures a queue with rank greater than the rank of the packet does not admit the packet (if $R_p < \hat{l}_q^p$, the constraint forces x_{pq} to 0). The second constraint ensures a queue does not admit the packet if a lower priority queue admits the packet (if $R_p \geq \hat{l}_{q-1}^p$, the constraint forces x_{pq} to). The last constraint forces the optimization to place the packet in one of the queues.

Modeling push up. Recall SP-PIFO updates the rank of the queue to the most recently admitted packet's rank. We model this as:

$$l_q^p = \hat{l}_q^p + x_{pq}(R_p - \hat{l}_q^p)\ \forall \text{packet}\, p\ \forall \text{queue}\, q \tag{22}$$

This constraint only updates the rank of queue q to R_p if the packet is placed in the queue ($x_{pq} = 1$). We can linearize this constraint [22].

Unique solution for SP-PIFO. We can combine these con-

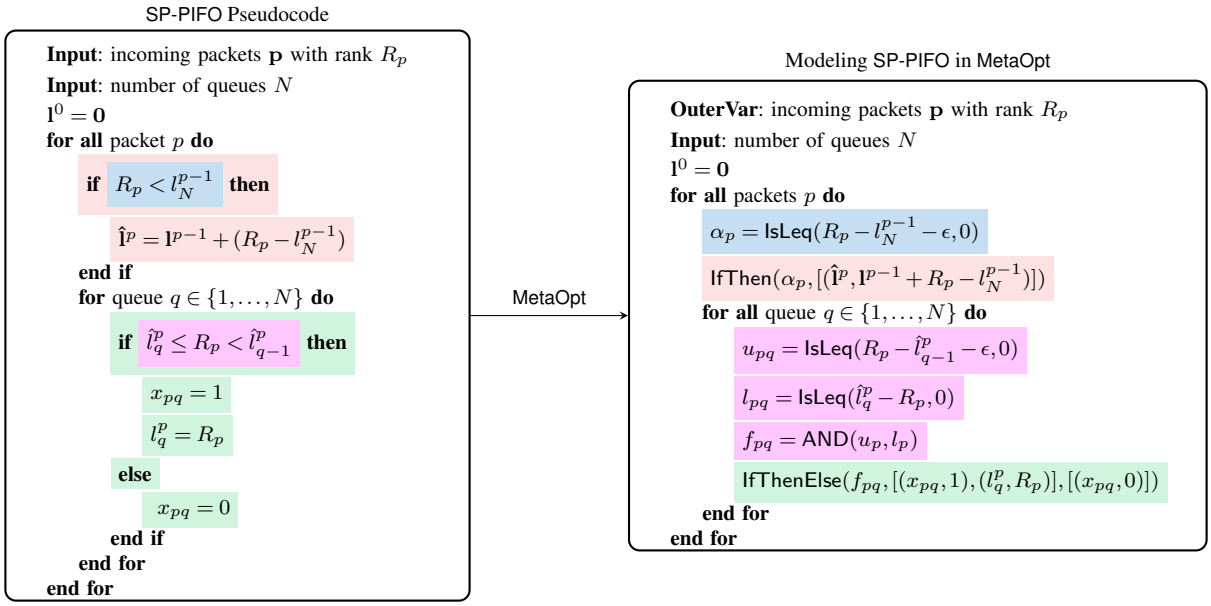


FIGURE A.4: The pseudocode for SP-PIFO and how users can model it in MetaOpt using the helper functions.

straints to uniquely specify SP-PIFO's decisions on a sequence of incoming packets. All these constraints are linear or linearizable using standard techniques even though packet ranks are variables in the outer problem.

Computing weighted average delay. We measure the gap in terms of the average delay of forwarding packets weighted by their priority. To measure delay of a packet, we count how many packets SP-PIFO decides to dequeue before it. Let d_{pj} indicate whether packet p is dequeued after packet j. We model the weighted average delay as:

$$\text{Weighted avg delay} = \frac{1}{P} \sum_{\text{pkt } p, j \neq p} (R_{max} - R_p) d_{pj} \tag{23}$$

Next, we define d_{pj}. We first assign weights to the packets such that the weights respect the order in which the packets should be dequeued (a packet p has a higher weight than j if it should be dequeued before j). We assign weights w_p as:

$$w_p = -p + \sum_{\text{queue } q} qP x_{pq} \tag{24}$$

This weighting guarantees that (1) a packet from a higher priority queue always has a higher weight than a packet from a lower priority queue, and (2) among the packets in the same priority queue, the one arrived earlier has a higher weight. Packet p is dequeued after packet j if the weight of packet j is higher.

$$w_j - w_p \leq M d_{pj} \leq M + w_j - w_p \;\; \forall \text{packets} p, j \tag{25}$$

Note that weights are unique.

C.2 Formulation of AIFO

AIFO [74] is an admission control on top of a FIFO queue that tries to approximate the same set of packets a PIFO queue would admit and is specifically designed for shallow buffers. AIFO keeps a window of recently seen packet ranks and computes the relative rank of the new packet with respect to this window. Then, it compares this quantile estimate with the fraction of available space in the queue (multiplied by some constant burst factor). If the quantile is lower or equal, AIFO admits the packet. Otherwise, it drops the packet.

Finding quantile estimate. Recall that AIFO computes how many packets in its recent window have lower ranks than an incomming packet p. We model this as:

$$R_p - R_j \leq M g_{pj} \leq M + R_p - R_j - \epsilon \tag{26}$$
$$\forall \text{packet} p \forall \text{packet} j: \; p - K \leq j \leq p - 1$$

$$g_p = \sum_{\text{pkt } j: p-K \leq j \leq p} g_{pj} \tag{27}$$

where M is a large constant ($M \geq R_{max}$) and ϵ is a small constant ($\epsilon < 1$). Observe that the first constraint compares the current packet with the last K packets (the ones in the window). For every pair, if $R_p > R_j$, the left constraint in Equation 26 forces $M g_{pj}$ to be positive and consequently $g_{pj} = 1$ (packet j in the window has a lower rank). If $R_p \leq R_j$, the right constraint forces g_{pj} to be 0. Equation 27 keeps track of the number of packets in the window with rank less than the rank of packet p. For packets $p < K$, we add some additional variables that represent the rank of packets arrived

Term	Meaning
B	burst factor
C	queue size in the number of packets
a_p	1 if packet p is admitted, $=0$ if dropped
K	number of samples in the window to estimate the quantile
g_p	number of packet ranks in the window smaller than rank of packet p
g_{pj}	1 if rank of packet j in the window is smaller than rank of packet p

TABLE A.7: Additional Notations for AIFO.

and departed before this sequence.

Deciding to admit or drop. Recall that AIFO admits the packet if the quantile estimate of the current packet rank is less than some factor of the available capacity of the queue. We model this as:

$$\hat{c}_p = B\frac{C - \sum_{\text{pkt } i<p} a_i}{C} \quad \forall \text{packet } p \tag{28}$$

$$\hat{c}_p - g_p + \epsilon \leq Z a_p \leq Z + \hat{c}_p - g_p \quad \forall \text{packet } p \tag{29}$$

where Z is a large constant ($\geq$ maximum of window size and queue size). The first constraint computes the fraction of available capacity multipled by a constant (burst factor in [74]) and the second constraint ensures $a_p = 0$ (*i.e.,* we drop the packet) if the quantile estimate is higher than the available capacity metric $\hat{c}_p$ and $a_p = 1$ otherwise.

Computing the final ordering of packet is similar to SP-PIFO. These constraints in combination find AIFO's unique solution. All the constraints are also linear.

C.3 Proof of Theorem 2

We use the same approach as our proof for FFDSum. We show a constructive example that matches the gap. In our example, first p packets with the lowest rank (=0) arrive, then 1 packet with the highest possible rank (=R_{max}), and finally, p^* packets with the second highest possible rank (=$R_{max}-1$).

Given this sequence of packets; SP-PIFO first adds all the p packets with the lowest rank to the lowest priority queue and then, updates the queue rank to 0 (*i.e.,* push up). Then, it adds the highest rank packet to the lowest priority queue and updates the queue rank to R_{max}. Lowest priority queue can not admit the packets with rank=$R_{max}-1$ anymore because the condition to admit a packet is that the queue rank should be lower than the packet rank. So, all the p^* packets are enqueued in a higher priority queue. As a result, all these p^* packets are going to be forwarded before the p packets with highest priority (Fig. A.5 shows this using an example).

We can compute the weighted sum of packet delays for SP-PIFO and PIFO as:

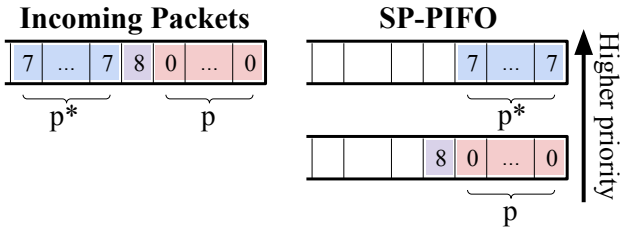


FIGURE A.5: Example of the input trace for SP-PIFO when $R_{max} = 8$. In this case, SP-PIFO dequeues all the packets with the lowest rank (highest priority) after all the second lowest priority packets ($r = R_{max} - 1$). Packets arrived earlier are on the right side of the queue.

$$\text{Wdelay}_{\text{PIFO}} = \frac{R_{max}p(p-1)}{2} + pp^* + \frac{p^*(p^*-1)}{2} \tag{30}$$

$$\text{Wdelay}_{\text{SP-PIFO}} = \frac{p^*(p^*-1)}{2} + R_{max}pp^* + \frac{R_{max}p(p-1)}{2} \tag{31}$$

We can compute the difference in the weighted sum of delays as:

$$\begin{aligned}\text{Wdelay}_{\text{SP-PIFO}} - \text{Wdelay}_{\text{PIFO}} &= (R_{max}-1)pp^* \\ &= (R_{max}-1)(N-1-p)p\end{aligned} \tag{32}$$

Note that $p + p^* = N - 1$. We can derive Theorem 2 by finding the maximum of Equation 32.

D List of MetaOpt Helper Function

Table A.8 lists the helper functions in MetaOpt. MetaOpt internally and automatically translates these into constraints. For specific use cases, please refer to Fig. A.1 for DP, Fig. A.3 for FFD, and Fig. A.4 for SP-PIFO.

E Black-box search methods

We next describe our baselines in more detail. We compared MetaOpt to these baselines in §4.

Random search. This strawman solution picks random inputs, computes the gap, and returns the input that resulted in the maximum gap after .

Hill climbing is a simple local search algorithm. It first randomly chooses an arbitrary input $\mathbf{d}_0$ and then generates its neighbors ($\mathbf{d}_{aux}$): it adds to $\mathbf{d}_0$ a value, which it draws from a zero-mean σ^2-variance Gaussian distribution. If this neighboring input increases the gap the hill climber moves to it. Otherwise it draws another neighbor.

The hill climber repeats these steps until it fails to make progress and terminates. This happens when it fails to increase the gap for K consecutive iterations. The hill climber outputs its current solution as a local maximum once it terminates (Algorithm 1).

Helper Function	Description
$\mathsf{IfThen}(\mathsf{b}, [(\mathsf{x_i}, \mathsf{F_i}())])$	if binary variable $b = 1$ then $x_i = F_i()$ for all i.
$\mathsf{IfThenElse}(\mathsf{b}, [\mathsf{x_i}, \mathsf{F_i}()], [(\mathsf{y_j}, \mathsf{G_j}())])$	if binary variable $b = 1$ then $x_i = F_i()$ for all i, otherwise $y_j = G_j()$ for all j.
$\mathsf{b} = \mathsf{AllLeq}([\mathsf{x_i}], \mathsf{A})$	$b = 1$ if all x_is are $\leq$ a constant A, otherwise $b = 0$.
$\mathsf{b} = \mathsf{IsLeq}(\mathsf{x}, \mathsf{y})$	$b = 1$ if $x \leq y$, otherwise $b = 0$.
$\mathsf{b} = \mathsf{AllEq}([\mathsf{x_i}], \mathsf{A})$	$b = 1$ if all x_is are $=$ a constant A, otherwise $b = 0$.
$\mathsf{b} = \mathsf{AND}([\mathsf{u_i}])$	$b = 1$ if all u_is are $= 1$, otherwise $b = 0$.
$\mathsf{b} = \mathsf{OR}([\mathsf{u_i}])$	$b = 1$ if at least one $u_i = 1$, otherwise $b = 0$.
$\mathsf{y} = \mathsf{Multiplication}(\mathsf{u}, \mathsf{x})$	Linearizes multiplication of a binary variable u and a continuous variable x. (Internally, we choose a simpler encoding if x is non-negative)
$\mathsf{y} = \mathsf{MAX}([\mathsf{x_i}], \mathsf{A})$	$y =$ maximum of x_is and a constant A.
$\mathsf{y} = \mathsf{MIN}([\mathsf{x_i}], \mathsf{A})$	$y =$ minimum of x_is and a constant A.
$[\mathsf{b_i}] = \mathsf{FindLargestValue}([\mathsf{x_i}], [\mathsf{u_i}])$	$b_i = 1$ if x_i is the largest among the group of variables x_j with corresponding $u_j = 1$, otherwise $b_i = 0$. At least one $b_i = 1$.
$[\mathsf{b_i}] = \mathsf{FindSmallestValue}([\mathsf{x_i}], [\mathsf{u_i}])$	$b_i = 1$ if x_i is the smallest among the group of variables x_j with corresponding $u_j = 1$, otherwise $b_i = 0$. At least one $b_i = 1$.
$\mathsf{r} = \mathsf{Rank}(\mathsf{y}, [\mathsf{x_i}])$	$r =$ rank of variable y among the group of variables $[x_i]$ (quantile).
$\mathsf{ForceToZeroIfLeq}(\mathsf{v}, \mathsf{x}, \mathsf{y})$	Forces $v = 0$ if $x \leq y$ (users can model this with IfThen, but this one is customized and faster). Internally, we choose a simpler encoding if v is binary.

TABLE A.8: MetaOpt's helper functions. (b and u are binary variables, and x and y are continuous variables)

Algorithm 1 Hill climbing

Input: $\mathbf{d}_0, \sigma^2, K$
$\mathbf{d} \leftarrow \mathbf{d}_0,\ k \leftarrow 0$
while $k < K$ **do**
  $\mathbf{d}_{\text{aux}} \leftarrow \max(\mathbf{d} + \mathbf{z}, \mathbf{0})$ where $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$
  if $\text{gap}(\mathbf{d}_{\text{aux}}) > \text{gap}(\mathbf{d})$ **then** $\mathbf{d} \leftarrow \mathbf{d}_{\text{aux}},\ k \leftarrow -1$ **end if**
  $k \leftarrow k + 1$
end while
Output: $\mathbf{d}$

We re-run the hill climber M_{hc} times with different initial inputs and return the solution that produces the maximum gap to minimize the impact of the starting point.

Simulated annealing refines hill-climbing and seeks to avoid getting trapped in a local maxima [45]. The difference between the two algorithms is simulated annealing may still (with some probability) move to a neighboring input even if that input does not improve the gap.

Simulated annealing gradually decreases the probability of moving to inputs that do not change the gap: it defines a temperature term, t_p, which it decreases every K_p iterations to $t_{p+1} = \gamma t_p$. Here, $0 < \gamma < 1$ which ensures $t_p \to 0$. If $\text{gap}(\mathbf{d}_{\text{aux}}) \leq \text{gap}(\mathbf{d})$, we have $\mathbf{d} \leftarrow \mathbf{d}_{\text{aux}}$ with probability $\exp(\frac{\text{gap}(\mathbf{d}_{\text{aux}}) - \text{gap}(\mathbf{d})}{t_p})$. We repeat the process M_{sa} times and return the best solution.

Hill climbing vs simulated annealing. Hill climbing has less parameters and is better suited for smooth optimizations where there are few local-optima. But simulated annealing is better suited for intricate non-convex optimizations with many local-optima because its exploration phase, although slower, allows it to avoid local optima and works better in the long run.

Both of these algorithms have a number of hyperparameters: we run grid-search to find the ones that produce the highest gap.