Minding the gap between Fast Heuristics and their Optimal Counterparts: Extended Version

Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Srikanth Kandula
Microsoft

ABSTRACT

Production systems use heuristics because they are faster or scale better than the corresponding optimal algorithms. Yet, practitioners are often unaware of how worse off a heuristic's solution may be with respect to the optimum in realistic scenarios. Leveraging two-stage games and convex optimization, we present a provable framework that unveils settings where a given heuristic underperforms.

CCS CONCEPTS

• **Networks** → **Network performance analysis**; **Network reliability**; **Network management**;

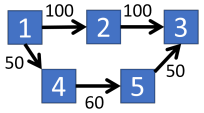
KEYWORDS

Network management, Heuristics, Adversarial Inputs

1 INTRODUCTION

Several recent solutions to networking problems are heuristic approximations to potentially intractable optimal algorithms [1, 15, 16, 21, 29, 37]. These heuristics are often faster or scale better, yet there is no clear understanding of how they behave with different inputs or how far from the optimal their outputs may drift. Many (e.g., [1, 5, 37]) lack an optimality-gap proof or an understanding of when they may underperform. Our goal is to provide practitioners with a means of determining this gap and finding inputs that lead to poor performance in practice, so that they can deploy workarounds or otherwise ameliorate the negative impact.

Consider a production WAN traffic controller at Microsoft whose heuristic [21] has two phases: First, it routes all demands with value at or below a threshold through their shortest path. It then jointly routes the remaining demands over multiple paths, leading to substantial speed-up since the latter has fewer demands. We refer to this heuristic as demand pinning (DP). In contrast, the optimal scheme (OPT) jointly considers all demands. Figure 1 shows a case where DP is sub-optimal: the demand 1 → 3 is at the threshold (= 50), so DP consumes capacity on its shortest path, which reduces the available capacity for flows 1 → 2 and 2 → 3. The gap in flow carried between the heuristic and optimal is 100 units (over 38%). While this example is illustrative, the largest possible gap for this topology and heuristic is not clear. Similarly, it is unclear what happens if the topology or heuristic changes.



Demand		DP (thresh=50)		OPT	
src-dest	value	path	value	path	value
1-3	50	1-2-3	50	1-4-5-3	50
4-5	10	4-5	10	4-5	10
1-2	100	1-2	50	1-2	100
2-3	100	2-3	50	2-3	100
		Total DP	**160**	Total OPT	**260**

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 (OPT) solution. DP first sends the demands at or below the threshold (50) over their shortest paths and then routes the remaining optimally.

A few questions are worth answering for heuristics. What is the worst-case outcome? That is, the input which maximizes the gap between the optimal and heuristic. Worst-case examples are not always realistic, thus raising the question: given some constraints (e.g., demands following the hose model [3, 28]), are there realistic inputs that trigger poor outcomes? Conversely, are there conditions which guarantee that the heuristic will perform well? Answers to these questions can guide practitioners in choosing between heuristics, devising alternatives for hard-to-solve inputs, or in combining heuristics with complementary strengths.

Answering these questions is hard. Traditional algorithmic worst- or average-case analyses [9, 26] are specific to individual heuristics and must be applied case-by-case. We are unaware of any such analyses for DP, POP [29], or [33, 37]. Also, for some heuristics, such analysis may not be possible, may only identify loose bounds or may not account for additional realistic input constraints. Verification methods also seek inputs that violate an invariant on a given function [18]; however, broadly speaking, these methods support *statically-specified* invariants on safety or correctness. In contrast, we seek inputs that maximize the gap between the optimal and

HotNets '22, November 14–15, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9899-2/22/11…$15.00
https://doi.org/10.1145/3563766.3564102

heuristic algorithms. To find such inputs, one can employ local search algorithms [11, 20], which iteratively pick an input, evaluate its gap, and then switch to a neighboring input with a larger gap. While local search algorithms apply to any (potentially black-box) heuristic or optimal algorithm, the flip side of such generality is that they ignore specific knowledge of the inner workings of the heuristic. Consequently, for large input spaces, we find that local search algorithms are often slow, get stuck in local optima, and fail to find desirable (practical) inputs.

Our main contribution is *a method that can provably find the input leading to the largest gap between a heuristic and an optimal solution while also supporting a rich class of additional constraints on the inputs that practitioners can specify to define realistic scenarios*. The method applies whenever users can pose the heuristic and the optimal method as convex optimizations. The idea is rather simple: we formulate searching for adversarial inputs with the largest gap as:

$$\arg\max_{\text{s.t. input } \mathcal{I} \in \text{ConstrainedSet}} \text{OPT}(\mathcal{I}) - \text{Heuristic}(\mathcal{I}). \tag{1}$$

OPT() and Heuristic() are solutions to convex optimizations with input specified by $\mathcal{I}$, and ConstrainedSet contains additional constraints on inputs. We can interpret the solution to (1) through the lens of game theory as a Stackelberg equilibrium [23], where the *outer* problem (or leader) picks an input, and the *inner* followers maximize OPT() and Heuristic() individually in response to the leader's choice. Such games are not convex, and existing solvers do not support multi-level optimization. In §3.1, we show how to, using optimization theory [8], translate the two-level optimization into a single-shot optimization that off-the-shelf solvers can solve. We discuss support for heuristics that take random decisions (e.g., POP [29]) and heuristics that take conditional actions and may not appear at first blush to be convex (e.g., DP [21]) so they fit within this framework. Compared to local search methods, we find significantly larger gaps ($\geq 20\%$) while reducing the search time by as much as three orders of magnitude. We are unaware of any prior work that finds provably strong adversarial examples for heuristics. We believe our techniques are a promising first step, but much work remains (see §5), and we invite the community to help.

2 WHY HEURISTICS MATTER

As a running example, we focus on heuristics for flow allocation in WAN traffic engineering (TE). The optimal form 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 optimizing a given objective (e.g., total flow [1], max-min fairness [15, 16], or utility curves [22]). We define the feasible flow over a pre-configured set of paths as:

Term	Meaning
$\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}$	Sets of nodes, edges, demands, and paths
c_e, p	c_e: capacity of edge $e \in \mathcal{E}$ path p: set of 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 volume ($d_k \geq 0$)
$\mathbf{f}, f_k^p$	$\mathbf{f}$: flow assignment vector with elements f_k f_k^p: flow for demand k on path p

Table 1: Multi-commodity flow problems' notation.

$$\begin{aligned}
&\text{FeasibleFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq \{\mathbf{f} \mid \\
&f_k = \sum_{p \in \mathcal{P}_k} f_k^p, && \forall k \in \mathcal{D} \quad \text{(flow for demand } k\text{)} \\
&f_k \leq d_k, && \forall k \in \mathcal{D} \quad \text{(flow below volume)} \\
&\sum_{k,p \mid p \in \mathcal{P}_k, e \in p} f_k^p \leq c_e, && \forall e \in \mathcal{E} \quad \text{(flow below capacity)} \\
&f_k^p \geq 0 && \forall p \in \mathcal{P}, k \in \mathcal{D} \quad \text{(non-negative flow)}\}
\end{aligned} \tag{2}$$

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

$$\begin{aligned}
\text{OptMaxFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq \arg\max_{\mathbf{f}} & \sum_{k \in \mathcal{D}} f_k \\
\text{s.t.} \quad & \mathbf{f} \in \text{FeasibleFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}).
\end{aligned} \tag{3}$$

Variables and constraints of optimization (3) increase with $|\mathcal{D}| + |\mathcal{E}|$ where $|\mathcal{D}|$ is typically quadratic in $|\mathcal{V}|$. Prior work shows this does not scale: at large network sizes, the solver can take tens of minutes to finish, hindering its use in practice for managing networks with dynamic demands [1].

We describe two heuristics that improve scalability by reducing the size of the TE problem in different ways.

Demand Pinning (DP), which is in production use [21], pre-allocates flow via shortest paths for all node pairs whose demand is below a configuration threshold T_d:

$$\begin{aligned}
&\text{DemandPinning}(\mathcal{D}, \mathcal{P}, T_d) \triangleq \{\mathbf{f} \mid \forall k \in \mathcal{D}, \\
&d_k > T_d \text{ or } f_k^p = \begin{cases} d_k & \text{if } p \text{ is shortest path in } P_k \\ 0 & \text{otherwise} \end{cases} \}
\end{aligned} \tag{4}$$

We can write DP as an optimization with added constraints that route demands below the threshold on shortest paths.

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

Partitioned Optimization Problems (POP) [29]. POP divides node pairs (and their demands) uniformly at random into a number of partitions and solves the original problem in parallel, once per partition, with edge capacities also uniformly divided across the problems. If $|\mathcal{D}|$ is much larger than $|\mathcal{E}|$ and there are c partitions, each of the per-partition problems is roughly $1/c$ the size of the original leading to

a substantial speedup – LP solver times are typically super-linear in problem sizes [6]. We describe POP as:

$$\text{POPMaxFlow}(\mathcal{V}, \mathcal{E}, \mathcal{D}, \mathcal{P}) \triangleq \bigcup_{\text{part. c}} \text{OptMaxFlow}(\mathcal{V}, \mathcal{E}_c, \mathcal{D}_c, \mathcal{P}), \quad (6)$$

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 capacity.

We seek demands that maximize the gap between (3) and the two heuristics (5) and (6) using (1); (3) is the inner problem OPT(), and (5) or (6) are the inner problem Heuristic().

3 FINDING ADVERSARIAL INPUTS

A direct approach to find adversarial inputs is *black-box* search algorithms such as simulated annealing [20]. They iteratively select a new input, execute optimal and heuristic solvers on that, and then use the gap to guide future iterations. Black-box solutions are, however, slow in practice §4. We present a novel *white-box* approach applicable whenever OPT() and Heuristic() are representable as convex optimizations. Our method uses the KKT theorem [6] to rewrite the two-stage game as a single-shot optimization, can be implemented in existing solvers, is generally faster, and finds higher quality (as well as optimal) gaps than black-box approaches.

Scope of applicability: We show that a broad class of heuristics (randomized and those that take conditional actions – POP and DP) can be specified as convex optimizations. Finding adversarial gaps as in (1) is more complex than the underlying optimal algorithms and heuristics; thus, scalability and timeliness are valid concerns. However, in practice, we only invoke gap finding on each new heuristic to better understand its limitations and so timeliness is a less stringent constraint. In fact, all the results in this paper finish within an hour.

Although the overall problem in (1) is larger (in terms of constraints and variables) than the underlying problems, we show the key computational challenge stems from the non-linear (convex) constraints introduced by the KKT rewrite. Thus, we report results on small albeit practical topologies. Scaling to larger topologies remains an open problem; see §5.

3.1 Single-shot Optimization

We are unaware of any commodity solver directly supporting two-stage optimization (1). When inner problems are convex, we can reformulate them as a single-shot optimization using KKT and replace them with sets of feasibility constraints. Take the example in Figure 2. The optimization on the right finds a feasible point satisfying the new constraints – any such feasible solution is also optimal for the problem on the left. This property holds for any convex problem with at least one

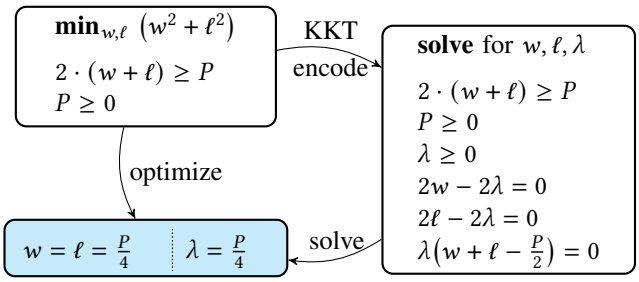


Figure 2: Example optimization: minimize the diameter of a rectangle with width w and length ℓ whose perimeter $\geq P$. We convert the inner optimization (top left) into a feasibility problem using the KKT theorem. The equations with λ variables correspond to first order derivatives of inequality constraints in the original problem. The perimeter P is a variable to an outer optimization but is treated as a constant in the inner problem.

strictly feasible point (Slater's condition [6]). Variables for the outer problem, which are constants in the inner problems (P in Figure 2 and $\mathcal{I}$ in (1)), play no role in the KKT rewrite.

An astute reader may observe potential issues with this translation: First, the inequality constraints in the optimization result in non-convex multiplicative constraints. However, we can encode multiplicative constraints using features available in commodity solvers – special ordered sets in Gurobi [14] and disjunctions in Z3 [27]. Second (see Figure 2), the KKT conversion increases the problem size (1 constraint and 2 variables on the top left increase to 4 and 3 respectively on the right) by a constant factor since the number of new variables and constraints is proportional to the size of the original problem. We empirically find the number of multiplicative constraints (not the overall size of the problem) determines the latency of the solver.

3.2 Codifying Heuristics as Convex Problems

The proposed method requires OPT() and Heuristic() in (1) to be representable as convex optimizations. From (2) and (3), it is easy to see that OPT() is a linear (hence, convex) optimization problem. We discuss next how to represent the two heuristics as convex optimizations.

Supporting DP. We can encode the *or constraint* in (4) via a big-M approach as follows. Denoting by $\widehat{p}_k$ the shortest path for demand k, we postulate the following constraints:

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

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

where M is a large pre-specified constant. Notice that, whenever the demand d_k is below the threshold T_d, the above constraints ensure the allocated flow will be zero on all paths but the shortest and the flow allocated on the shortest path will match the demand. We convert these max functions into a convex form using standard techniques [6].

Supporting POP. POP is convex as it is the union of solutions to several disjoint linear optimizations (6). Nonetheless, an additional complexity in POP is the random partitions causing $\text{POP}(\mathcal{I})$ in (1) to be a random variable. So, we must look for inputs that maximize the gap for a deterministic descriptor of the random variable. We can do that by finding adversarial inputs that are worse in expectation by replacing $\text{Heuristic}(\mathcal{I})$ in (1) with its expected value $\mathbb{E}\left(\text{Heuristic}(\mathcal{I})\right)$. We approximate this expectation with empirical averages on a few different randomly generated partitions of POP. Alternatively, we can look for the gap between the optimal solution and a pre-specified tail percentile of the random heuristic (using multiple random partition instantiations and a sorting network [17, 32] to bubble up the worst outcomes). We discuss details of a more complex POP in appendix A.

3.3 Additional Details

Gap search. To improve the scalability, we exploit two observations on how commodity solvers operate. First, the solvers use good guesses (for the multiplicative constraints) and usually find a reasonable (but not optimal) solution quickly. Next, branch-and-bound techniques incur a significant amount of time to find only marginally better inputs or to prove that no such input exists. Thus, for solvers which show incremental progress (e.g., Gurobi), we timeout the solver when the incremental progress in a given time window is smaller than 0.5% (the primal-dual gap — which solvers report — bounds how far our solution is from optimal when stopped [6]); for solvers which do not show progress (e.g., Z3), we iteratively ask for any input with a gap that is at least as large as a specified value and binary sweep the value with a fixed timeout.

Realistic constraints on inputs. Our proposed method can search for inputs that maximize the gap within a constrained subspace specified using ConstrainedSet in (1). Here, we present two classes of realistic constraints.

- *Bounded distance from a goalpost:* We constrain the input to be no more than a certain *distance* from a *goalpost*. An example is for the demands of POP or DP to be close to historically observed demands; we can specify the distance in absolute or relative terms and use multiple goalposts with different distance ranges. The goalpost may be partially specified; i.e., some demands can be unconstrained.
- *Intra-input constraints* are of the form $g(\mathcal{I}) \geq f(\mathcal{I})$ or $g(\mathcal{I}) = f(\mathcal{I})$, where f and g are functions of the input $\mathcal{I}$. An example of such a constraint is that all demands are within a specified distance from the average demand.

3.4 Black-box Search Techniques

We discuss hill climbing [11] and simulated annealing [20], which can find adversarial gaps for any heuristic.

Algorithm 1 Hill climbing

Input: $\mathbf{d}_0$, σ^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: $\mathbf{d}$

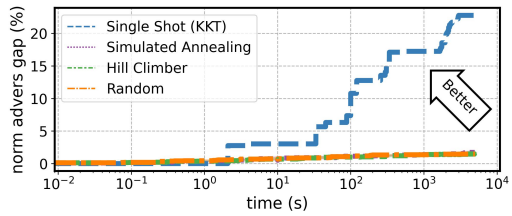
Hill climbing is arguably the simplest local search algorithm. We start at a randomly chosen arbitrary demand $\mathbf{d}_0$ and generate neighbors by adding a value drawn from a zero-mean σ^2-variance Gaussian distribution to every element of the current demand independently. If the neighboring demand increases the gap, we move to that demand. Otherwise, we draw another neighbor and repeat the evaluation. If we do not find any neighboring demand that increases the gap after K evaluations, we output the current solution as a local maximum; see Algorithm 1. We then repeat this process M_{hc} times starting from different random initial demands and return the one with the maximum gap. We set $\sigma = 10\%$ of link capacity, $K = 100$ and M_{hc} based on the latency budget.

Simulated annealing is a refinement of hill climbing seeking to avoid getting trapped in local maxima [20]. A key distinction is that even if a neighboring demand does not improve the gap, we still move to such a demand with some probability. Specifically, if $\text{gap}(\mathbf{d}_{\text{aux}}) \leq \text{gap}(\mathbf{d})$, we have that $\mathbf{d} \leftarrow \mathbf{d}_{\text{aux}}$ with probability $\exp(\frac{\text{gap}(\mathbf{d}_{\text{aux}}) - \text{gap}(\mathbf{d})}{t_p})$, where t_p is called the temperature. t_p is initialized at t_0, and in every K_p iterations, it is decreased as $t_{p+1} = \gamma t_p$ for $0 < \gamma < 1$. Notice that $t_p \to 0$, i.e., the probability of moving to a demand that does not improve the gap decreases with iterations. Thus, simulated annealing mimics hill climbing more closely as the number of iterations increases. We repeat the process M_{sa} times and return the best solution. We set $t_0 = 500$, $\gamma = 0.1$, $K_p = 100$, and M_{sa} based on the desired latency.

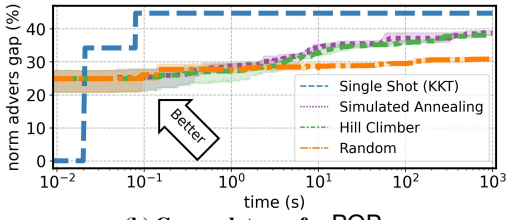
Hill climbing requires less hyperparameter tuning than simulated annealing and is well-suited for smooth optimizations. Simulated annealing is better suited for intricate non-convex optimizations because its exploration phase, although initially slow, tends to work better in the long run.

4 EARLY RESULTS

Our goal with an initial evaluation is to: (i) Verify that our method finds worst-case gaps between practical heuristics and their optimal counterparts. We show that our white-box technique significantly outperforms black-box alternatives, which cannot find examples with large gaps and have orders of magnitude higher latency. (ii) Verify that the discovered inputs and gaps are qualitatively useful and shed light on what causes the optimality gap for heuristics in realistic conditions.

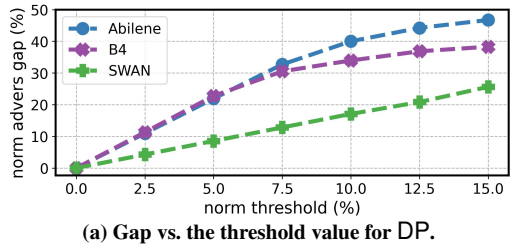


(a) Gap vs. latency for DP.

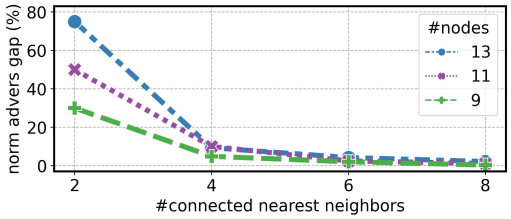


(b) Gap vs. latency for POP.

Figure 3: Gap between OPT and heuristics vs. execution time on B4. Our technique finds larger gaps faster.



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



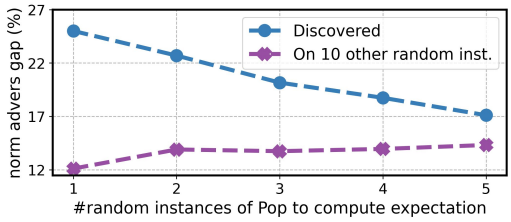
(b) Gap vs. average path length.

Figure 4: Gap between OPT and DP in different scenarios.

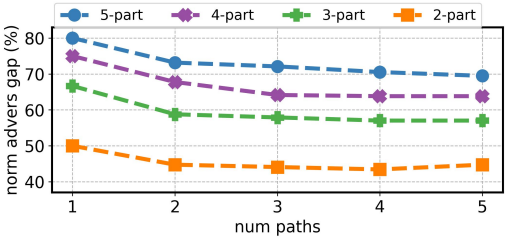
Methodology. We report the discovered worst-case gaps between OptMaxFlow and the POP and DP heuristics (see §2) on three production topologies (SWAN [15], B4 [16], Abilene [34]) and on synthetic topologies. For POP, we vary the number of partitions (=2 if unspecified). For DP, we vary the threshold that controls which demands are pinned to their shortest paths (=5% of link capacity if unspecified). We also vary the number of paths available for each node pair (=2 if unspecified) and use Gurobi [14] as the solver for our method.[1]

Maximum discovered gap vs. latency. Figure 3 shows both heuristics have sizable optimality gaps: 20%–45%. To obtain a metric comparable across topologies, this figure plots the difference in the carried demand divided by the sum of edge capacities. Our white-box technique outperforms the black-box solutions in both finding examples with larger gaps and the required amount of time to find them. For DP, black-box techniques have a harder time finding large gaps since the space of inputs with a notable gap is much smaller than POP.[2] All methods ran on one thread on a desktop.[3]

[1] Our SMT-based implementation by Z3 [27] is omitted due to a slow runtime.



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



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

Figure 5: Gap between OPT and POP on B4.

Qualitative findings. Figure 4a shows that higher thresholds cause a larger gap in DP because the heuristic will force more demands onto their shortest paths. The gap increases faster for some topologies, although all three topologies have roughly the same number of nodes and edges. To understand why, Figure 4b shows results on synthetic topologies: circles with n nodes where each node connects to a varying number of its nearest neighbors. The optimality gap grows with the average (shortest) path length. Intuitively, this is because pinning demands on longer paths uses up capacity on more edges and has a greater reduction in the total flow.

Using a single random POP partition in (1) finds inputs with a large gap for that partition but a much smaller gap when tested on 10 other random partitions (Figure 5a). As discussed in §3.2, we resolve this by using multiple random partitions and seeking inputs with a large average gap; 5 random instances suffice to find consistently bad inputs. A larger number of partitions leads to higher optimality gaps in POP (Figure 5b), perhaps because edge capacity is divided between more partitions. When more paths are available between node pairs, the gap reduces somewhat because the additional paths allow the heuristic to use more of the fragmented capacity.

The discovered optimality gaps are significantly larger than reported in [21, 29]. DP is challenged when nodes further apart have demands below the threshold (serving small demands on longer paths uses capacity along more edges which

[2] A small portion of the space of valid demand values are pinned (e.g., 5%).
[3] All methods are parallelizable but to slightly varying degree. Latency would also improve using SIMD or hardware support.

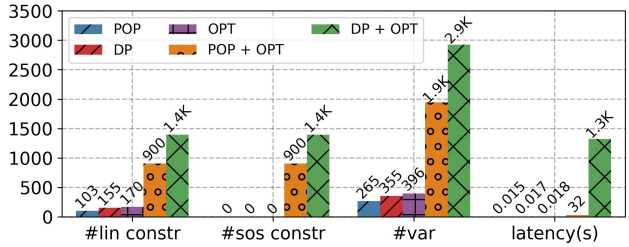


Figure 6: Optimization size (#linear and #SOS constraints, and #variables) and latency on B4 for the POP and DP heuristics.

could have been used to carry multiple flows on shorter paths). The random partitioning of capacity and demands in POP causes its optimality gap since unused capacity in a partition can be used to carry demands of another partition.

Problem sizes (Figure 6). The metaoptimization in (1) (POP + OPT and DP + OPT) has more constraints and variables than the individual heuristic/optimal formulations. The solver latency (on a single thread), however, is disproportionately larger for the metaoptimization due to the multiplicative constraints (SOS in Gurobi) introduced by the KKT rewrite.

5 OPEN ISSUES AND FUTURE WORK

Practical considerations. Besides identifying the worst-case adversarial input, we allow users to examine gaps under different practical constraints on inputs (see §3.3). Users can also search for diverse kinds of bad inputs by iteratively removing the previously-found inputs from the search space of subsequent iterations. Given these example inputs, operators may decide to use different heuristics for different input spaces or pre-compute safe solutions for particular corner cases. Moreover, the metaoptimization in (1) can be used to find topology changes that cause the worst-case gap for a specific heuristic instead of focusing only on the adversarial demands.

Scaling to larger problem sizes. As KKT introduces a large number of multiplicative constraints, we are exploring alternative rewrites based on primal/dual relationships [12]. For a subspace of heuristics, we find the worst gaps happen only at extremum points; thus, constraining or quantizing the space of inputs can speedup the search without sacrificing quality. Preliminary results show we can scale to larger topologies.

Searching for sufficient conditions. A use case of our techniques is identifying realistic constraints on the input space with small worst-case optimality gap, then safely use the heuristic on inputs in that space. We are exploring AutoML-style techniques [35, 36] to discover such constraints.

Generalization. Heuristics for capacity planning [2] and failure resilient routing [24, 37] have uncertain optimality gaps. Extending our approach to such heuristics that approximate mixed integer programs (non-convex) is an open problem. We support heuristics that are specifiable in a convex form and offer some standard encoding techniques [6].

Identifying infeasibility. Our method (1) finds an adversarial input among the feasible set. However, certain heuristics have infeasible inputs; e.g., with DP it is possible to have a set of demands with value below the threshold and a common link on their shortest path such that the total demands exceeds the link's capacity. Finding infeasible inputs remains open.

6 RELATED WORK

To the best of our knowledge, no prior work finds adversarial inputs for heuristics that approximate optimal problems, specifically networking heuristics. Our techniques (e.g., big-M and KKT rewrites, and generally translating the problem to one that is amenable to off-the-shelf solvers) are not per-se novel [4, 7, 12] but no other work has combined them in this way. We further show extensions to randomized and conditional heuristics. Without our changes, we could not apply existing solvers directly or find large gaps in a short duration.

Our qualitative results – the optimality gap and hard examples for POP and DP – are novel. Microsoft actively uses DP [21]; POP [29] provides high-level guidance (e.g., need for granularity) on when it does well but neither works show hard inputs nor discuss how to find them.

For learnt techniques, in congestion control, video bitrate selection etc., some recent works identify malicious inputs [13, 25]. However, they are specific to the individual cases and none consider an explicitly stated optimal algorithm nor identify provably large gaps. Some PL techniques identify code paths that require too much computation or memory and probabilistically model edge cases [19, 30]. We mathematically specify the heuristics in this paper: they are amenable to direct analyses without learning the model from analyzing code. [10, 31] are also broadly related to our work.

7 FINAL THOUGHTS

We show how to find adversarial inputs for heuristics written as convex programs while accounting for additional practical constraints. A key enabler is the use of optimization techniques that convert seemingly intractable problems to ones implementable in commercial, production-grade solvers such as Gurobi and Z3. Using our techniques, a practitioner can identify when their heuristic is guaranteed to perform well. They can also find the worst-case behavior for realistic inputs.

Acknowledgements. We thank our anonymous reviewers and Ranveer Chandra, Ramesh Govindan, Luis Irun-Briz, Umesh Krishnaswamy, Konstantina Mellou, and Luke Marshall.

This work does not raise any ethical concerns.

REFERENCES

[1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. 2021. Contracting Wide-area Network Topologies to Solve Flow Problems Quickly. In *NSDI*.

[2] Satyajeet Singh Ahuja, Varun Gupta, Vinayak Dangui, Soshant Bali, Abishek Gopalan, Hao Zhong, Petr Lapukhov, Yiting Xia, and Ying Zhang. 2021. Capacity-efficient and uncertainty-resilient backbone network planning with hose. In *SIGCOMM*.

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Datacenter Networks. In *NSDI*.

[4] Bryan Arguello, Richard L. Chen, William E. Hart, John D. Siirola, and Jean-Paul Watson. [n. d.]. Modeling Bilevel Program in Pyomo. https://www.osti.gov/servlets/purl/1526125. ([n. d.]).

[5] Jeremy Bogle et al. 2019. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *SIGCOMM*.

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

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

[8] Benoît Colson, Patrice Marcotte, and Gilles Savard. 2007. An overview of bilevel optimization. *Annals of Operations Research* 153, 1 (2007), 235–256.

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

[10] Anthony Corso, Robert Moss, Mark Koren, Ritchie Lee, and Mykel Kochenderfer. 2021. A Survey of Algorithms for Black-Box Safety Validation of Cyber-Physical Systems. *Journal of AI Research* (2021).

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

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

[13] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. 2019. Robustifying Network Protocols with Adversarial Examples. In *HotNets*. ACM.

[14] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. (2022). https://www.gurobi.com

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

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

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

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

[19] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic Profiling of Stateful Data Planes for Adversarial Testing. In *ASPLOS*. ACM.

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

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

[22] Alok Kumar et al. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM*.

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

[24] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic engineering with forward fault correction. In *SIGCOMM*.

[25] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. 2019. (Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs. In *HotNets*. ACM.

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

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

[28] Pooria Namyar, Sucha Supittayapornpong, Mingyang Zhang, Minlan Yu, and Ramesh Govindan. 2021. A Throughput-Centric View of the Performance of Datacenter Topologies. In *SIGCOMM*.

[29] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. In *SOSP*.

[30] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated Synthesis of Adversarial Workloads for Network Functions. In *SIGCOMM*. ACM.

[31] Pedro Reviriego and Daniel Ting. 2021. Breaking Cuckoo Hash: Black Box Attacks. *IEEE Transactions on Dependable and Secure Computing* (2021).

[32] Thomas Sauerwald. [n. d.]. Sorting Networks. https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/advalg.pdf. ([n. d.]).

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

[34] Stanford University IT. 2015. Abilene Core Topology. (2015). https://uit.stanford.edu/service/network/internet2/abilene

[35] Christian Steinruecken, Emma Smith, David Janz, James Lloyd, and Zoubin Ghahramani. 2019. The automatic statistician. In *Automated Machine Learning*. Springer, Cham.

[36] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: A fast and lightweight autoML library. *Proceedings of Machine Learning and Systems* 3 (2021), 434–447.

[37] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. 2021. ARROW: Restoration-Aware Traffic Engineering. In *SIGCOMM*.

A POP CLIENT SPLITTING

In §2, we introduce the (basic) POP heuristic [29], which incorporates *resource splitting* for our WAN TE problem, and in §3.2, we argue that POP can be represented as a convex optimization problem. The work in [29] also specifies an extended full-fleshed version of the heuristic that incorporates an additional operation named *client splitting*. In this appendix, we demonstrate that the extended version of POP can also be expressed 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 (6). The function ClientSplit() generates several duplicates of the existing demands and reduces their volume accordingly. In essence, it performs several operations where $(s_k, t_k, d_k) \in \mathcal{D}$ is replaced by two elements of the form $(s_k, t_k, d_k/2)$. This operation is iteratively performed until the process is terminated; see [29] for more details.

Below we propose a way to encode a version 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[4] or the split demand is lower than d_{th}.

Without loss of generality, we illustrate this idea for a single demand d_1: we can replicate this process for all demands in $\mathcal{D}$. For concreteness, let us set that we will split this demand at most twice (giving rise to at most 4 virtual clients). Hence, we construct a priori the flows for all the possible splits of the client d_1. More precisely, instead of having a single variable f_1 for the first demand (see Table 1) we have seven variables 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. Notice that these flows must satisfy the following linear constraints

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

Now, we want $f_{1,1}$ to be exactly zero unless $d_1 < d_{\text{th}}$ (hence, no client splitting occurs), which we can achieve using big-M constraints as illustrated in §3.2, i.e.,

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

Similarly, 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

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

where the small pre-specified $\epsilon > 0$ is added 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 $d_1 \ge 2d_{\text{th}}$. This can be encoded as

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

The procedure here illustrated for the first demand can be replicated for all d_k, thus effectively encoding POP with client splitting as a convex optimization problem. Once this is done, the techniques in §3 apply.

[4]Notice that [29] 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.