Minding the gap between Fast Heuristics and their Optimal Counterparts

Paper #125, 6 pages + 1 for references

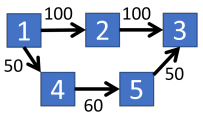
Abstract– Production systems use heuristics because they are faster or scale better than the corresponding optimal algorithms. Yet, practitioners are often unaware of how worse off a heuristic's solution may be with respect to the optimum in realistic scenarios. Leveraging two-stage games and convex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

1 Introduction

Several recent solutions to networking problems are heuristic approximations to potentially intractable optimal algorithms [2, 17, 18, 24, 31, 39]. 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., [2, 6, 39]), lack an optimality-gap proof or an understanding of when or how 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 work-arounds or otherwise ameliorate the negative impact.

Consider a production WAN traffic controller at Microsoft whose heuristic has two phases: First it routes all demands whose value is at or below a threshold through the shortest path [24]. It then jointly routes the remaining demands over multiple paths which leads to substantial speed-up since the latter has fewer demands to consider. We refer to such heuristics as demand pinning (DP). In contrast, the optimal scheme (OPT) jointly considers all demands. Figure 1 shows a case where DP is clearly suboptimal: the demand $1 \to 3$ is at the threshold $(= 50)$ and so DP consumes capacity on the shortest paths which reduces the available capacity for flows $1 \to 2$ and $2 \to 3$. The gap in flow carried between the heuristic and optimal is 100 units (over 38%). While this example is illustrative, it is not clear if its the largest gap we can achieve for this topology and heuristic. Similarly, its unclear what happens when the topology or the heuristic change.

A few questions are worth answering for heuristics in general. What is the worst-case outcome? That is, the input (e.g., topology or demands) that 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 are governed by a hose model [4, 20]), are there realistic inputs that trigger poor outcomes for a heuristic? Conversely, are there conditions we can impose to guarantee the heuristic performs well? Answers to these questions can guide practitioners in choosing between heuristics, devising alternative strategies for hard-to-solve inputs, or to motivate combining heuristics that have complementary strengths.



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) Network topology with unidirectional link capacities. (right) A set of demands and their flow allocations using the DP heuristic and the optimal (OPT) solution. The heuristic first sends those demands at or below the threshold (50) over their shortest paths and then routes the remaining demands optimally.

Answering these questions is hard. Traditional algorithmic worst- or average-case analyses [10, 29] are specific to individual heuristics and we can only apply them on a case-by-case basis. For many heuristics, we may not be able to do even that; we are unaware of any such analysis for DP, POP [31], or other heuristics (e.g., [39, 35]). Such analysis sometimes only identifies loose bounds and may not account for additional realistic constraints on the inputs. Verification methods also seek inputs that violate an invariant on a given function [21]; however, broadly speaking, these methods support *statically-specified* invariants on safety or correctness. In contrast, we seek inputs that maximize the gap between two functions (the optimal and heuristic algorithms). To find such inputs one can employ local search algorithms [12, 23] which iteratively pick an input, evaluate the gap between the heuristic and optimal for that input, and then switch to a neighboring input that increases the gap. While they apply to any (potentially black-box) heuristic or optimal algorithm, the flip side of such generality is they do not exploit any specific knowledge of the inner workings of the heuristic during the search. Consequently, for large input spaces, we find local search algorithms are often slow, get stuck in local optima, and do not succeed at finding (practical) inputs with a large gap.

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 basic idea is rather simple: we formulate searching for adversarial inputs that have the largest gap as the following optimization

$$\underset{\text{s.t. input } \mathcal{I} \in \mathsf{ConstrainedSet}}{\operatorname{argmax}} \mathsf{OPT}(\mathcal{I}) - \mathsf{Heuristic}(\mathcal{I}). \qquad (1)$$

In (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 [26], 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 today's solvers do not support multi-level optimization. In §3.1 we show how to, using ideas from optimization theory [9], translate the two-level optimization into a single-shot optimization that off-the-shelf solvers can solve. We discuss support for heuristics which take random decisions (e.g., POP [31]) and heuristics that take conditional actions and may not appear at first blush to be convex (e.g., DP [24]) so that they fit within this framework. Compared to local search methods, we find significantly larger gaps ($\geq 20\%$) while reducing the search time 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 explore.

2 Background and Why Heuristics Matter

As a running example, we focus on heuristics used to solve flow allocation problems for WAN traffic engineering (TE). The optimal form of these problems typically involves solving a multi-commodity flow problem. Given a set of nodes, capacitated edges, demands between nodes, 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 which optimizes a given objective such as total flow, weighted max-min fairness [17, 18], or utility curves [25].

Using notation from Table 1, we define the feasible flow over a pre-configured set of paths as:

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

Among all the feasible flows, the optimal solution seeks to maximize the total flow across the network, i.e.,

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

The optimization in (3) contains a number of variables and constraints that increase with $|\mathcal{D}| + |\mathcal{E}|$. The number of demands is typically quadratic in the number of nodes. Prior work show this does not scale well: at large network sizes, the solver can take tens of minutes to finish, hindering its use in practice for managing networks with dynamic demand [2].

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 1: Multi-commodity flow problems' notation.

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 [24], pre-allocates flow via shortest paths for all node pairs whose demand is below a configuration threshold T_d:

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

We can write DP as an optimization with additional constraints to encode that either the demand is above the threshold or is fully routed along the shortest path. Thus, DP has fewer demands to optimize.

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

Partitioned Optimization Problems (POP) [31]. POP divides node pairs (and their demands) uniformly and at random into a specified number of partitions and solves the original problem in parallel, once per partition, with edge capacities also uniformly divided across the problems. If the number of demands is much larger than the number of edges and there are c partitions, each of the per-partition problems is roughly $1/c$ the size of the original problem leading to a substantial speedup — LP solver times are typically super-linear in problem sizes [7]. We describe POP as (cf. (3)):

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

where union here is the vector union and 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). In the context of the outer problem in (1), (3) is the inner problem OPT() and (5) or (6) are the inner problem Heuristic().

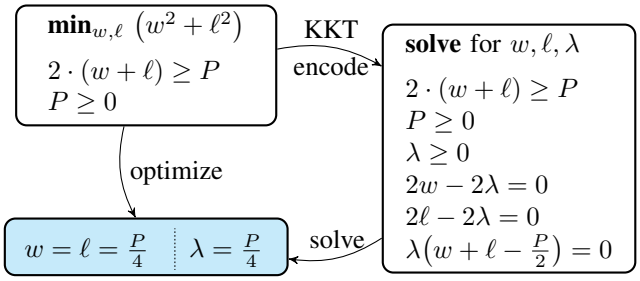


Figure 2: Example optimization: minimize the diameter of a rectangle with width w and length ℓ whose perimeter is at least P. Top left: inner optimization. We convert the optimization into a feasibility problem using the KKT theorem. The equations with λ variables correspond to first order derivatives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

3 Techniques to Find Adversarial Gaps

A direct approach to find adversarial inputs that have a large optimality gap would be to apply *black-box* search algorithms such as simulated annealing [23], which iteratively select a new input to try, execute both the optimal and heuristic solvers on that input, and then use the gap to guide future iterations. But black-box solutions are slow in practice (see §4). We present a novel *white-box* approach that applies whenever both the heuristic and optimal algorithm can be represented as convex optimizations, our approach uses the KKT theorem [7] to rewrite the two-stage game as a single-shot optimization which can be implemented in commodity solvers, is generally faster, and finds higher quality (as well as optimal) gaps compared to black-box approaches.

Scope of applicability: We show a broad class of heuristics can be specified as convex optimizations. We also show extensions to support randomized heuristics and those that take conditional actions — e.g., POP and DP respectively. 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, making timeliness a less stringent constraint. Indeed, all the results in this paper finish within one hour.

Although the overall problem in (1) is larger (in the number of constraints and variables) than the underlying problems, we show that the key computational challenge stems from the non-linear (convex) constraints introduced by the KKT rewrite. Thus, currently, we only report results on small albeit practical topologies (e.g., B4 [18], SWAN [17] and Abilene [36]). Scaling to larger topologies remains an open problem; see §5.

3.1 Single-shot Optimization

To reformulate the two-stage metaoptimization in (1) as a single-shot optimization, when both inner problems are convex, we use KKT conditions to replace the inner problems with sets of constraints. Consider a simple example of such rewrite in Figure 2. The new set of constraints on the right do not have any maximization objective. Any feasible solution to these constraints will be an optimal solution for the problem on the left. This property holds for any convex problem which has at least one strictly feasible point that satisfies all constraints (Slaters condition [7]). Observe also that variables for the outer problem, which are treated as constants in the inner problems (such as P in Figure 2 and $\mathcal{I}$ in (1)), play no role in the KKT rewrite. We are unaware of any commodity solvers that support two-stage games directly.

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 show how to encode multiplicative constraints using features already available in commodity solvers –special ordered sets in Gurobi [16] and disjunctions in Z3 [30]. Second (see Figure 2), the KKT conversion increases the problem size (2 constraints and 2 variables on the left increase to 3 and 6 respectively on the right). The increase in problem size is provably no more than a constant factor since KKT adds one new variable per inequality in the original problem and one per objective; as well, KKT adds one new constraint per variable in the original problem and one per new variable. In our experiments, we find the numbers of multiplicative constraints (and not the overall size of the outer problem) determines the latency of the outer optimization.

3.2 Codifying Heuristics as Convex Problems

The proposed method above requires $\mathsf{OPT}()$ and $\mathsf{Heuristic}()$ in (1) to be representable as convex optimizations. From (2) and (3) it is easy to see that $\mathsf{OPT}()$ 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

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

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

where M is a large pre-specified constant. Notice that, whenever 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 then convert these $\max$ functions into a convex form using standard techniques [7].

Supporting POP. We can see from (6) that POP is convex because it is the union of solutions to several disjoint linear optimal problems. Nonetheless, there is an additional element of complexity in analyzing POP because the partitions in (6) are randomly drawn. Thus, $\mathsf{POP}(\mathcal{I})$ in (1) is a random variable and we must look for inputs which maximize the gap with respect to some deterministic descriptor of the

random variable. For such random heuristics, we can find adversarial inputs that are worse in expectation by replacing $\mathsf{Heuristic}(\mathcal{I})$ in (1) with its expected value $\mathbb{E}\left(\mathsf{Heuristic}(\mathcal{I})\right)$. In practice, we can approximate this expectation with empirical averages that we generate over a few different random partitions of POP. As another example, instead of the expected value, we can look for inputs where the gap between the optimal solution and a pre-specified tail percentile of the random heuristic is maximized. In practice, we can achieve this by using multiple random partition instantiations and a sorting network [19, 34] to bubble up the worst outcomes. Finally, a more complex variant of POP [31] requires using a form of the conditional rewrite above; we defer details to our extended version [1].

3.3 Additional Details

Gap search. To improve the scalability of our solution, 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 inputs that are only marginally better or to prove that no such input exists. Thus, for solvers which show incremental progress (e.g., Gurobi) we time-out the solver when the incremental progress in a given time window is smaller than 0.5% (the primal-dual gap — which solvers report — in these cases bounds how far our solution is from optimal when stopped [7]); for solvers which do not show progress (e.g., Z3), instead of asking to maximize the gap, we iteratively ask for any input with gap at least as large as some specified value and binary sweep the specified 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 demands that were observed historically; we can specify 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 two local search algorithms, hill climbing [12] and simulated annealing [23], which can find adversarial gaps for any arbitrarily specified heuristic.

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

Algorithm 1 Hill climbing

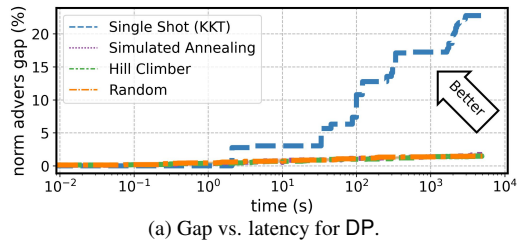
Input: $\mathbf{d}_0, \sigma^2, K$
$\mathbf{d} \leftarrow \mathbf{d}_0, \;\; k \leftarrow 0$
while $k < K$ **do**
  $\mathbf{d}_{\text{aux}} \leftarrow \max(\mathbf{d} + \mathbf{z}, \mathbf{0}) \quad \text{where} \quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$
  if $\text{gap}(\mathbf{d}_{\text{aux}})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

Simulated annealing is a refinement of hill climbing that seeks to avoid getting trapped in local maxima [23]. 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, even if $\text{gap}(\mathbf{d}_{\text{aux}}) \leq \text{gap}(\mathbf{d})$ we have that $\mathbf{d} \leftarrow \mathbf{d}_{\text{aux}}$ with probability $\exp(\frac{\text{gap}(\mathbf{d}_{\text{aux}}) - \text{gap}(\mathbf{d})}{t_p})$, which depends on the temperature parameter t_p. The temperature is initialized at t_0 and every K_p iterations it is decreased as $t_{p+1} = \gamma t_p$ for some positive $\gamma < 1$. Notice that $t_p \to 0$, i.e., the probability of moving to a demand that does not increase the gap decreases with iterations, thus, simulated annealing more closely mimics hill climbing as the number of iterations increases. Like hill climbing, we repeat this whole process M_{sa} times and return the best solution overall. We set $t_0 = 500$, $\gamma = 0.1$, $K_p = 100$, and M_{sa} based on the desired latency.

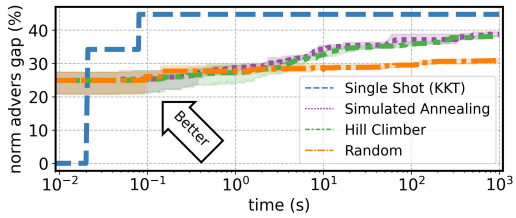
Hill climbing requires less hyperparameter tuning than simulated annealing and is well suited for smooth optimization landscapes. Simulated annealing is better suited to intricate non-convex optimization landscapes because its exploration phase, although initially slow, tends to work better in the long run. We compare with both approaches to illustrate the trade-off between performance and time complexity.

4 Early Results

Our goals with an initial evaluation are:
i) Verify our method finds worst-case gaps between practical heuristics and their optimal counterparts. We show our white-box technique significantly outperforms black-box alternatives which are unable to find examples with large gaps and have orders of magnitude higher latency.
ii) Verify the inputs and gaps we discover are qualitatively

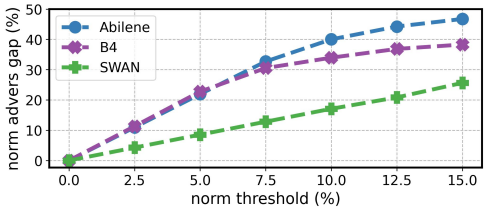


(a) Gap vs. latency for DP.

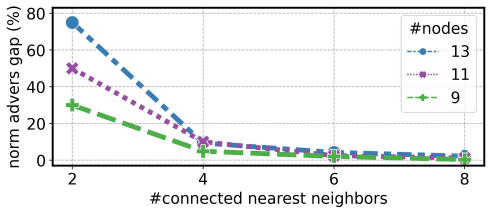


(b) Gap vs. latency for POP.

Figure 3: Gap between OPT and heuristics versus execution time on the B4 topology. Our white-box technique identifies larger gaps in shorter time than the alternatives.



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



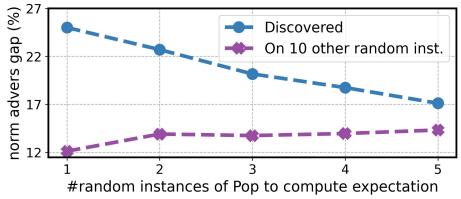
(b) Gap vs. average path length.

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

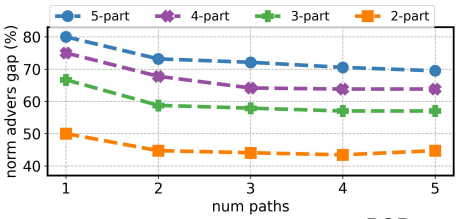
useful, i.e., allow us to shed light on what causes the optimality gap for heuristics in realistic conditions.

Methodology. We report the worst-case gaps we discover between OptMaxFlow and the POP and DP heuristics (see §2) on three production topologies: SWAN [17], B4 [18], Abilene [36] 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 report results for all of the techniques discussed in §3. We also vary the number of paths available for each node pair (=2 if unspecified) and vary the solvers for our white-box method (Gurobi [16] if unspecified[1]).

Maximum discovered gap vs. latency. Figure 3 shows that both heuristics have sizable optimality gaps: 20%–45%. The



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



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

Figure 5: Gap between OPT and POP on B4.

solution for the optimal and heuristics is the total demand that they can carry; to obtain a metric that is comparable across topologies this figure plots the difference in demand carried divided by the sum of edge capacities in a topology. Our white-box technique outperforms the black-box solutions both in finding examples that have larger gaps and the amount of time it takes to find them. All methods ran on one thread on a desktop and there is room to improve.[2]

Qualitative findings. Figure 4a shows larger thresholds cause DP to incur a larger gap from optimal because the heuristic will force more demands onto their shortest paths. The gap increases at a faster rate for some topologies although all three topologies have roughly the same numbers 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 is larger when the average (shortest) path length is large. Intuitively, this is because pinning demands on longer paths uses up capacity on more edges and has a greater reduction in the total flow that the heuristic can carry.

Using a single random partition of POP in (1) will find inputs that have a large optimality 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 concern by using multiple random instances of POP and by seeking inputs that have a large average gap; 5 random instances suffice to find inputs that are consistently bad for POP. Larger number of partitions lead to higher optimality gaps in POP (Figure 5b) perhaps because edge capacity is statically divided between more partitions. When node-pairs have more paths available between them, the gap reduces somewhat because the additional paths allow the heuristic to use more of the capacity that is fragmented between partitions.

The optimality gaps we find are significantly larger than

[1]We also have an SMT based implementation using Z3 [30] but omit results due to its slow runtime.

[2]All methods are parallelizable but to slightly varying degree. Latency would also improve using SIMD or hardware support.

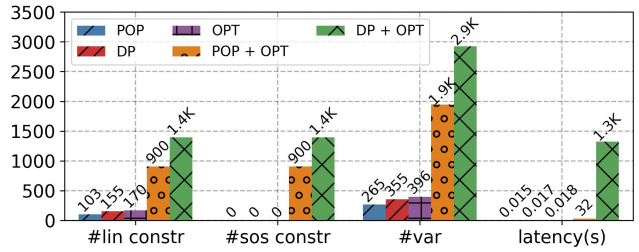


Figure 6: Optimization size (number of linear and SOS constraints, and number of variables) and latency for different solvers on B4.

reported in prior works [24, 31].[3] We find that demand pinning is challenged when nodes that are further apart have demands below the threshold (intuitively, serving small demands on longer paths uses capacity along more edges which 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 capacity can remain unused in a partition that could have been used to carry demands that happen to be assigned to some other partition.

Problem sizes (Figure 6). The size of the metaoptimization in (1) (denoted by POP + OPT and DP + OPT in the figure) is larger than the individual heuristic/optimal formulations in the number of constraints and variables. The solver latency (on a single thread), however, is disproportionately larger for the metaoptimization due to the multiplicative constraints (SOS in Gurobi [16]) that KKT introduces for each inequality in the inner problems.

5 Open Issues and Future Work

We discuss open issues regarding how to use the findings regarding worst-case gaps and hard example inputs as well as how to scale further.

Scaling to larger problem sizes. As noted already, the use of KKT introduces a large number of multiplicative constraints. To scale further, we are exploring alternative rewrites that leverage primal/dual relationships [13]. For a subspace of heuristics, we find the worst gaps happen only at extremum points; thus, constraining or quantizing the search space of inputs can speed up the search without sacrificing quality.

Searching for sufficient conditions. A special case use of our techniques would be to identify realistic constraints on the input space for which the worst-case optimality gap is small. We can then safely use a heuristic on any input in that space. We are exploring how to automatically discover such realistic constraints using AutoML-style techniques [37, 38].

Generalization. Heuristics for capacity planning (e.g., [3]) and proactive failure resilient routing (e.g., [27, 39]) also have uncertain optimality gaps. Many such heuristics approximate mixed integer programs that are non-convex. Extending our approach to these heuristics is an open problem.

[3]We model POP's extension from [31] in [1] but omit it here due to space constraints.

Identifying in-feasibility. Certain heuristics have infeasible inputs; for example, with DP it is possible to have a set of demands which are (a) below the threshold and (b) have a common link on their shortest path such that the total value of the set of demands exceeds the capacity of that link. Finding such infeasible inputs remains open and interesting.

6 Related Work

To the best of our knowledge, no prior work finds adversarial inputs for heuristics that approximate optimal problems, and more specifically, networking heuristics. Our techniques (e.g., rewriting the inner optimizations as constraints using KKT, big-M rewrites for multiplicative constraints, translating the problem to one that is amenable to off-the-shelf solvers) are not per-se novel [5, 8, 14] but no other work has combined them in the way we have. We also show extensions to randomized and conditional heuristics and show how to encode conditional heuristics as convex problems. Without our changes, we would not be able to directly apply existing solvers 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 [24]; POP [31] provides high-level guidance (e.g., need for granularity) on when POP will do well but neither works show hard example inputs nor discuss how to find them.

For learnt techniques, in congestion control, video bitrate selection etc., some recent works identify malicious inputs [15, 28]. However, the techniques 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 [22, 32]. We can mathematically specify the heuristics we consider in this paper: they are amenable to direct analyses without having to learn the model from analyzing code. Other works such as [33] and [11] are also broadly related to our work.

7 Final Thoughts

For a large set of heuristics (which we can write as convex programs), including randomized and conditional algorithms, we show how to find the worst-case optimality gap. As well, we find inputs that achieve maximum gap while meeting additional specified constraints. A key enabler is our use of convex optimization techniques (KKT rewrites, big-M, expected gaps) that convert seemingly intractable problems to ones that are implementable in commercial, production-grade solvers such as Gurobi and Z3. Using our techniques, a practitioner can identify sufficient conditions when their heuristic is guaranteed to perform well. They can also identify the worst-case behavior for realistic inputs. Much work remains, specifically, on supporting larger problem sizes and a more general class of heuristics and we discuss some initial ideas that can help.

This work does not raise any ethical concerns.

8 References

[1] Anonymous extended version. https://github.com/AdversInputHeuristics/Adversarial-Input-For-Heuristics.

[2] F. Abuzaid, S. Kandula, B. Arzani, I. Menache, M. Zaharia, and P. Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *NSDI*, 2021.

[3] S. S. Ahuja, V. Gupta, V. Dangui, S. Bali, A. Gopalan, H. Zhong, P. Lapukhov, Y. Xia, and Y. Zhang. Capacity-efficient and uncertainty-resilient backbone network planning with hose. In *SIGCOMM*, 2021.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Datacenter Networks. In *NSDI*, 2010.

[5] B. Arguello, R. L. Chen, W. E. Hart, J. D. Siirola, and J.-P. Watson. Modeling bilevel program in pyomo. https://www.osti.gov/servlets/purl/1526125.

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

[7] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[8] B. Colson, P. Marcotte, and G. Savard. Bilevel programming: A survey. *4OR*, 2005.

[9] B. Colson, P. Marcotte, and G. Savard. An overview of bilevel optimization. *Annals of Operations Research*, 153(1):235–256, 2007.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[11] A. Corso, R. Moss, M. Koren, R. Lee, and M. Kochenderfer. A survey of algorithms for black-box safety validation of cyber-physical systems. *Journal of AI Research*, 2021.

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

[13] P. Garcia-Herreros, L. Zhang, P. Misra, E. Arslan, S. Mehta, and I. E. Grossmann. Mixed-integer bilevel optimization for capacity planning with rational markets. *Computers & Chemical Engineering*, 86:33–47, 2016.

[14] P. Garcia-Herrerosa, L. Zhang, P. Misra, S. Mehta, and I. E. Grossmann. Mixed-integer bilevel optimization for capacity planning with rational markets. *Computers & Chemical Engineering*, 2016.

[15] T. Gilad, N. H. Jay, M. Shnaiderman, B. Godfrey, and M. Schapira. Robustifying network protocols with adversarial examples. In *HotNets*. ACM, 2019.

[16] Z. Gu, E. Rothberg, and R. Bixby. Gurobi optimizer reference manual, version 5.0. *Gurobi Optimization Inc., Houston, USA*, 2012.

[17] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[18] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, and M. Zhu. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[19] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *SIGCOMM*, 2016.

[20] V. Jeyakumar et al. EyeQ: Practical Network Performance Isolation for the Multi-tenant Cloud. In *Usenix HotCloud*, 2012.

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

[22] Q. Kang, J. Xing, Y. Qiu, and A. Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *ASPLOS*. ACM, 2021.

[23] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[24] U. Krishnaswamy, R. Singh, N. Bjørner, and H. Raj. Decentralized cloud wide-area network traffic engineering with BLASTSHIELD. In *NSDI*, 2022.

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

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

[27] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[28] R. Meier, T. Holterbach, S. Keck, M. Stähli, V. Lenders, A. Singla, and L. Vanbever. (self) driving under the influence: Intoxicating adversarial network inputs. In *HotNets*. ACM, 2019.

[29] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[30] L. d. Moura and N. 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.

[31] D. Narayanan, F. Kazhamiaka, F. Abuzaid, P. Kraft, A. Agrawal, S. Kandula, S. Boyd, and M. Zaharia. Solving large-scale granular resource allocation problems efficiently with POP. In *SOSP*, 2021.

[32] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *SIGCOMM*. ACM, 2018.

[33] P. Reviriego and D. Ting. Breaking cuckoo hash: Black box attacks. *IEEE Transactions on Dependable and Secure Computing*, 2021.

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

[35] R. Singh, N. Bjorner, S. Shoham, Y. Yin, J. Arnold, and J. Gaudette. Cost-effective capacity provisioning in wide area networks with Shoofly. In *SIGCOMM*, 2021.

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

[37] C. Steinruecken, E. Smith, D. Janz, J. Lloyd, and Z. Ghahramani. The automatic statistician. In *Automated Machine Learning*. Springer, Cham, 2019.

[38] C. Wang, Q. Wu, M. Weimer, and E. Zhu. FLAML: A fast and lightweight autoML library. *Proceedings of Machine Learning and Systems*, 3:434–447, 2021.

[39] Z. Zhong, M. Ghobadi, A. Khaddaj, J. Leach, Y. Xia, and Y. Zhang. Arrow: Restoration-aware traffic engineering. In *SIGCOMM*, 2021.

APPENDIX

A POP Client Splitting

In §2 we introduce the (basic) POP heuristic [31], 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 [31] also specifies an extended full-fleshed version of the heuristic that incorporates an additional operation denominated *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 $\text{ClientSplit}()$ 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 [31] 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

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

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.,

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

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

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

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 $d_1 \geq 2d_{\text{th}}$. This can be encoded as

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

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 [31] 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.