Towards Safer Heuristics With $\mathcal{X}$Plain

Pantea Karimi[1*], Solal Pirelli[2*], Siva Kesava Reddy Kakarla[3], Ryan Beckett[3], Santiago Segarra[4], Beibin Li[3], Pooria Namyar[5], Behnaz Arzani[3]
[1]MIT [2]EPFL, Sonar [3]Microsoft Research [4]Rice University [5]University of Southern California

Abstract

Many problems that cloud operators solve are computationally expensive, and operators often use heuristic algorithms (that are faster and scale better than optimal) to solve them more efficiently. Heuristic analyzers enable operators to find when and by how much their heuristics underperform. However, these tools do not provide enough detail for operators to mitigate the heuristic's impact in practice: they only discover a *single input instance* that causes the heuristic to underperform (and not the full set) and they do not *explain why*.

We propose $\mathcal{X}$Plain, a tool that extends these analyzers and helps operators understand when and why their heuristics underperform. We present promising initial results that show such an extension is viable.

CCS Concepts

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

Keywords

Heuristic Analysis, Explainable Analysis, Domain-Specific Language

ACM Reference Format:
Pantea Karimi*, Solal Pirelli*, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Beibin Li, Pooria Namyar, Behnaz Arzani. 2024. Towards Safer Heuristics With $\mathcal{X}$Plain. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696348.3696884

*Equal contribution: work done partly as an intern at Microsoft Research.

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696884

1 Introduction

Operators use heuristics (approximate algorithms that are faster or scale better than their optimal counterparts) in production systems to solve computationally difficult or expensive problems. These heuristics perform well across many typical instances, but they can break in unexpected ways when network conditions change [5, 6, 16, 35]. Our community has developed tools that enable operators to identify such situations [1, 2, 6, 16, 35]. These tools find the "performance gap" of one heuristic algorithm compared to another heuristic or the optimal — they identify an example instance of an input which causes a given heuristic to underperform.

For example, MetaOpt [35] describes a heuristic deployed in Microsoft's wide area traffic engineering solution and shows it could underperform by 30% (see §2). This means the company would either have to overprovision their networks to support 30% more traffic, drop that traffic, or delay it.

The potential benefit of heuristic analyzers is clear: they allow operators to quantify the risk of heuristics they want to deploy. Although these heuristic analyzers have already shed light on the performance gap of many deployed heuristics, they are still in their nascent stage and have limited use for operators who do not have sufficient expertise in formal methods and/or optimization theory. There are crucial features missing: operators have to (1) model the heuristics they want to analyze in terms of mathematical constructs these tools can support and (2) manually analyze the outputs from these tools to understand *how* to fix their heuristics or their scenarios — the tool only provides a performance gap and an example input that caused it. They do not produce the full space of inputs that can cause large gaps nor describe why the heuristic underperformed in these instances.

The latter problem limits the operator's ability to use the output of these tools to fix the problem and to either improve the heuristic, create an alternative solution for when it underperforms, or cache the optimal solution for those instances. In our earlier examples, the operator has to look at the tool's example demand matrix to understand why the heuristic routes 30% less traffic than the optimal.

The state of these heuristic analyzers today is reminiscent of the early days of our community's exploration of network verifiers and their potential to help network operators configure and manage their networks. In the same way that network

verifiers enabled operators to identify bugs in their configurations [10, 14, 15, 19, 22, 24, 27, 28, 30, 32, 39, 47, 49], a heuristic analyzer can help them find the performance gap of the algorithms they deploy. Tools that allow operators to leverage heuristic analyzers more easily, identify *why* the heuristics underperform, and devise solutions to remediate the issue serve a similar purpose to the tools our community crafted that *explained* the impact of configuration bugs [23, 25, 39, 40] (by producing all sets of packets that the bug impacted and the configuration lines that caused the impact).

We propose $\mathcal{X}$Plain — our vision for a "generalizer" that can augment existing heuristic analyzers and help operators either improve their heuristics (by helping them find *why* the heuristics underperform) or use them more safely (by finding all regions where they underperform).

We propose a domain-specific language (§5.1), which allows us to concretely describe the heuristic's behavior and that of a benchmark we want to compare it to for automated analysis. It is rooted in network flow abstraction, which allows us to model the behavior of many heuristics that operators use in today's networks, including *all* those from [16, 35]. Our compiler converts inputs in this language into an existing heuristic analyzer. Our *efficient* iterative algorithm uses the analyzer, extrapolates from the adversarial inputs it finds, and finds all adversarial subspaces where the heuristic underperforms. We then use our language again and visualize *why* (i.e., the different decisions the heuristic made compared to the optimal that caused it to underperform) the heuristic underperforms in these cases.

We also discuss open questions and a possible approach built on the solutions we propose in this work to uncover what *properties* in the input or the problem instance cause the heuristic to underperform (§5.4). Our proof-of-concept implementation of this idea uses MetaOpt [35] as the underlying heuristic analyzer because it is open source. But our proposal applies to other heuristic analyzers such as [1, 2, 16] as well.

2 What is heuristic analysis?

Heuristic analyzers [1, 16, 35] take a *heuristic model* and a *benchmark model* (e.g., the optimal) as input. Their goal is to characterize the performance gap of the heuristic compared to the benchmark. Recent tools [16, 35] use optimization theory or first-order logic to solve this problem and return a single input instance that causes the heuristic to underperform.

Example heuristics from these work include:

Demand Pinning (DP) was deployed in Microsoft's wide area network. DP is a heuristic for the traffic engineering problem. The optimal algorithm assigns traffic (demands) to paths and maximizes the total flow it routes without exceeding the network capacity. Operators use DP to reduce the size of the optimization problem they solve. DP first filters all demands below a pre-defined threshold and routes them through (pins them to) their shortest path. It then routes the remaining demands optimally using the available capacity (see Fig. 1).

MetaOpt authors modeled DP as an optimization problem and provided helper functions to simplify operator use (Fig. 1b). MetaOpt solves a bi-level optimization that identifies the performance gap and the demand causing it (the flow in Fig. 1a). However, it is left to the operator to examine the output and determine why DP underperformed. While DP allows for manual analysis (see [35]), not all heuristics do. It is also difficult for operators to extrapolate from this adversarial input to find other regions where DP may fail. These limitations are exacerbated as we move to larger problems with more demands, where it is harder to pinpoint how a heuristic's decision to route a particular demand interferes with its ability to route others.

Vector bin packing (VBP) places multi-dimensional balls into multi-dimensional bins and minimizes the number of bins in use. Operators use VBP in many production systems, such as to place VMs onto servers [9]. The VBP problem is APX-hard [45]. One heuristic solving VBP is first-fit (FF), which greedily places an incoming ball in the first bin it fits in. Fig. 1c shows how one encodes it in MetaOpt.

MetaOpt produces the adversarial ball sizes 1%, 49%, 51%, 51% (as a percentage of the bin size) for an example with 4 balls and 3 equal-sized bins (we use single-dimensional balls) — the optimal uses 2 bins while FF uses 3 (we show a more complex version in Fig. 2). Once again, operators have to reason through this example to identify *why* FF underperforms and what *other inputs* cause the same problem. This is harder in FF and other VBP heuristics, such as best fit or first fit decreasing, as evidenced by the years of research by theoreticians in this space [36].

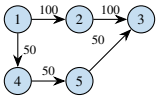
In this paper, we use the DP and VBP as running examples. These examples are representative of the heuristics prior work has studied [16, 35] (the scheduling examples Virley studies are conceptually similar to VBP, and we think our discussions directly translate to those use-cases).

Prior work [5] shows that, using a single adversarial instance, it is difficult to understand why a heuristic underperformed. It is even harder to generalize from why an adversarial input causes the heuristic to underperform on a single problem instance (or a few instances) to what *properties* in the input and the problem instance cause it to underperform.

3 The case for comprehensive analysis

Prior work [2, 5, 35] show explaining adversarial inputs can have benefits: we can improve DP's performance gap by an order of magnitude and produce congestion control algorithms that meet pre-specified requirements [2]. But these results require manual analysis [35] or problem-specific models [2, 5].

We see an opportunity for a new tool that enables operators to identify the full *risk surface* of the heuristic (the set



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**

(a) DP from [35]. (left) Topology. (right) A set of demands and their flow allocations using the DP heuristic and the optimal (OPT) solution.

OuterVar: d_k requested rate of demand k
Input: P_k paths for demand k
Input: $\hat{p}_k$ shortest path
Input: T_d demand pinning threshold
for all demand $k \in \mathcal{D}$ **do**
  ForceToZeroIfLeq$(d_k - f_k^{\hat{p}_k}, d_k, T_d)$
end for
MaxFlow()

(b) DP in MetaOpt.

OuterVar: Y(size of balls)
Input: C(capacity of bins)
for all ball i **and** bin j **do**
  $\mathrm{r}_{ij} = \mathrm{C}_j - \mathrm{Y}_i - \sum_{\text{ball } u<i} \mathrm{x}^d_{uj}$
  f_{ij} = AllLeq($[-\mathrm{r}^{\mathrm{d}}_{\mathrm{ij}}]_{\mathrm{d}}, 0$)
  γ_{ij} = AllEq($[\mathrm{x}^{\mathrm{d}}_{\mathrm{ik}}]_{\mathrm{d,k<j}}, 0$)
  α_{ij} = AND(f_{ij}, γ_{ij})
  IfThenElse($\alpha_{\mathrm{ij}}, [(\mathbf{x}_{ij}, \mathrm{Y}_i)], [(\mathbf{x}_{ij}, 0)]$)
end for

(c) Heuristic for VBP in MetaOpt.

Figure 1: Example heuristics and their encoding in MetaOpt (sub-figures (b) and (c)). Heuristic in sub-figure (b) forces the demands less than a threshold to be pinned and then solves a flow maximization problem, heuristic in sub-figure (c) assigns the first bin that can fit the ball.

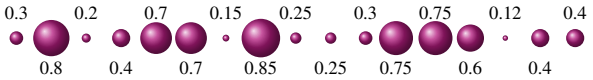


Figure 2: Example adversarial instance for FF with equal-sized bins with size of 1; the optimal uses 8 bins and the heuristic 9.

of inputs where the heuristic underperforms) and to identify *why* the heuristic underperforms automatically. It can produce (1) a description of the entire area(s) where a heuristic has a high performance gap; or (2) a description of what choices the heuristic makes that cause it to underperform (the difference in the actions of the heuristic and the optimal can point us to *why* the heuristic underperforms). Through these outputs, these tools can make it safer for operators to use heuristics in practice as they can mitigate the cases where they underperform and maybe even design safer heuristics.

There are three levels of information we can provide: (1) for *a given problem instance*, the *sets of inputs* that cause the heuristic to underperform; (2) for *a given problem instance*, a reason as to *why* the heuristic underperforms in each contiguous region of the adversarial input space; and (3) for *the general case*, the *characteristics* of the inputs and problem instances that cause the heuristic to underperform.

Take DP as an example. The ideal tool would produce:

Type 1. For a given topology, the adversarial input sets are of the form $\cup D_i$ where each $D_i \in \mathbb{R}^n_+$ represents a contiguous subspace of the n-dimensional (8-dimensional in Fig. 1a for 8 demands) space.

For a given D_i: (a) an entry $d_{ij} = T - \epsilon$ (here T is the demand pinning threshold and ϵ is a small positive value) if there are multiple paths between the nodes i and j (we call a demand $d : d \leq T$ a pinnable demand); (b) for all other uv where a portion of the path between the nodes u and v intersects with the shortest path of a pinnable demand we have $d_{uv} \geq \min(C_{uv} - T)$. Here, the set C_{uv} contains the capacity of all links on the path between u and v. The adversarial instance in our example in Fig. 1a fits this behavior.

Type 2. For a given topology, DP routes pinned demands on their shortest paths, but the optimal routes them through alternate paths. We expect the pinned demands in each contiguous subspace would all have a common pattern where they have the same shortest path, and DP does shortest-path routing for these demands, whereas the optimal does not.

Type 3. The heuristic's performance is worse when the length of the shortest path of the pinned demands is longer or the capacity of the links along these paths is lower — pinned demands limit the heuristic's ability to route other demands.

4 Challenges

It is hard to arrive at low-level models of a heuristic in order to use existing analyzers [2, 16, 35], and operators need to have expertise in either formal methods [2, 16] or optimization theory [12, 35] to do so. We see an analogy with writing imperative programs in assembly code: we can write any program in assembly but it takes time, has a high risk of being buggy, and makes code reviews (i.e., explanations) difficult.

Low-level models operate over variables and constructs that are often hard to connect to the original problem ("Greek letters" and "auxiliary variables" instead of "human-readable" text). To model the first fit behavior, MetaOpt uses an auxiliary, binary variable α_{ij} that captures whether bin j is the first bin where ball i fits in, and sets its value through:

$$\alpha_{ij} \leq \frac{f_{ij} + \sum_{\{k \in \text{BINS} | k<j\}} (1 - f_{ik})}{j} \quad \forall i \in \text{BALLS}, \forall j \in \text{BINS}$$

$$\sum_{j \in \text{BINS}} \alpha_{ij} = 1 \quad \forall i \in \text{BALLS}.$$

It is hard to derive an explanation from such a model and harder still to connect it to how the heuristic works to explain its behavior. We need a better and more descriptive language to encode the behavior of the heuristic. We also need to:

Find adversarial subspaces and validate them. These are subspaces of the input space where the inputs that fall in those subspaces cause the heuristic to underperform. To find them, we need a search algorithm that iterates and extrapolates from the adversarial inputs existing analyzers find (similar to the all-SAT problem [17, 34, 48], the input space is large, and we cannot blindly search it to find adversarial inputs [35]). Once we find a potential "adversarial subspace," we should validate it: we need to check whether the heuristic's performance gap is higher for inputs that belong to the adversarial subspace compared to those that do not with statistical significance.

Find why the inputs in each subspace cause bad performance. It is reasonable to assume the inputs in the same contiguous adversarial subspace trigger the same "bad behavior" in the heuristic. To find and explain these behaviors, we need to automatically reason through the heuristic's actions and compare them to those of the benchmark: we need to *concretely* encode the heuristic and benchmark's choices as part of the language we design for our solution. The challenge is to ensure this language applies to a broad range of problems and is amenable to the types of automation we desire.

Generalize beyond a single instance. Perhaps the hardest challenge is to generalize from the instance-based explanations to one that applies to the heuristic's behavior in the general case: we have to find a valid extrapolation from these instance-based examples and discover patterns that apply to the heuristic's behavior across different problem instances.

5 The $\mathcal{X}$Plain proposal

We propose $\mathcal{X}$Plain (Fig. 3). Users describe the heuristic and benchmark through its **domain specific language** (§5.1). The main purpose of this domain-specific language (DSL) is to *concretely define* the behavior of the heuristic and benchmark, which allows automated systems to analyze, compare, and explain their behavior. The **compiler** translates the DSL into low-level optimization constructs.

The **adversarial subspace generator**(§5.2) generates a set of contiguous subspaces where the inputs in each subspace cause the heuristic to underperform and the **significance checker** filters the outputs and ensures the subspaces are statistically significant — it checks that the inputs that fall into these subspaces produce higher gaps compared to those that do not with statistical significance.

The **explainer** (§5.3) describes how the heuristic's actions differ from the benchmark in each contiguous subspace for a given problem instance. The **generalizer** (Fig. 5.2) extrapolates from these instance-based observations to produce the properties of the inputs and the instance that cause the heuristic to underperform. It uses instance-based explanations across many instances to do so — we use the **instance generator** to create such instances.

5.1 The domain-specific language

To auto-generate the information we described in §3 we need a DSL to concretely encode the heuristic and benchmark algorithms. We need a DSL that: (1) can represent diverse heuristics; (2) we can use to automatically compile into optimizations that we can efficiently solve (those that existing solvers support and that do not introduce too many additional constraints and variables compared to hand-written models); and (3) is easy and intuitive to use.

We design an abstraction based on *network flow problems* [11]. Network flow problems are optimizations that, given a set of sources and destinations, optimize how to route traffic to respect capacity constraints, maximize link utilization, etc. Network flow problems impose two key constraints: the total flow on each link should be below the link capacity, and what comes into a node should go out (flow conservation).

There are advantages to using network flow problems: they have an intuitive graph representation [11] — operators know how to reason about the flow of traffic through such graphs; we can easily translate them into convex optimization or feasibility problems [11]; and they have many variants which we can use and build upon.

We can use the network flow model and extend it through a set of new "node behaviors" to ensure we can apply it to a broad class of heuristics. Node behaviors are a set of constraints that operate on the flows coming in and going out of each node: "split nodes" (enforce flow conservation constraints); "pick nodes" (enforce flow conservation constraints but only allow flow on a single outgoing edge); "copy nodes" (copy the flow that comes in onto all of their outgoing edges); "source" and "sink" nodes (produce or consume traffic); etc. A node can enforce multiple behaviors simultaneously. We include node behaviors that do not enforce flow conservation constraints (such as the "copy nodes") or capacity constraints by default so that we can model a broad set of heuristics. Users can also add metadata to each node or edge, which we can use later to improve the explanations we produce.

Users encode the problem, the heuristic, and the benchmark in the DSL in abstract terms. For example, to model VBP they specify that the problem operates over (abstract) sequences of different node types that correspond to the balls and bins in the VBP problem. Users also encode the actions the heuristic and the optimal can make in terms of the relationship between the different sequences of nodes and the edges that connect them and rules that govern how flow can traverse from one node to the next. To analyze a specific instance of the VBP problem, users input the number of balls and bins and then $\mathcal{X}$Plain concretizes the encoding (we show a concretized example with 4 one-dimensional balls and 3 bins in Fig. 4b).

Our DSL allows us to model the examples from prior work. We can model DP with split, source, and sink nodes (Fig. 4a),

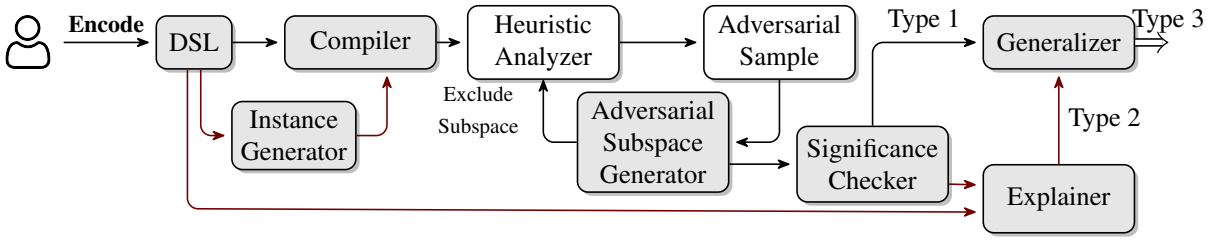
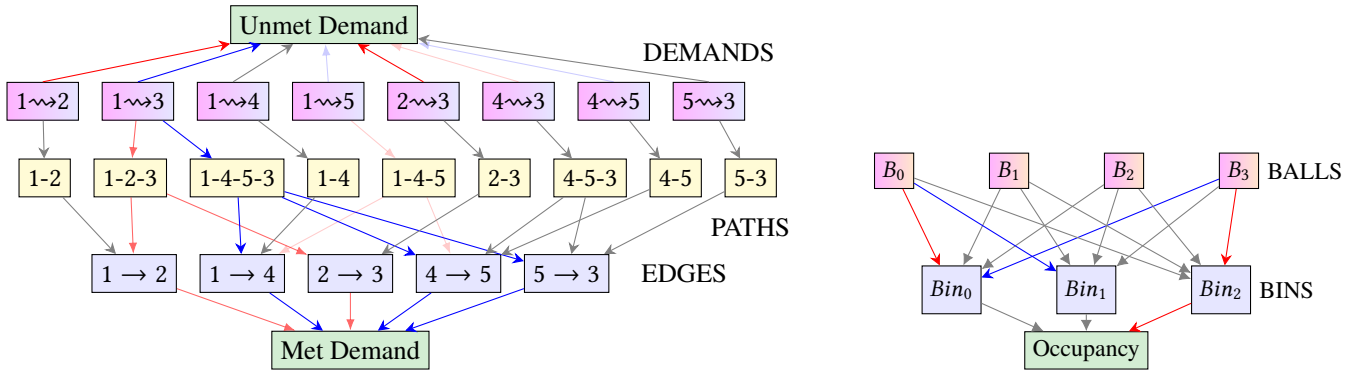


Figure 3: $\mathcal{X}$Plain: the system architecture we propose to extend existing heuristic analyzers.



(a) How we model our DP example (Fig. 1a) in the DSL. **(b) How we model FF in the DSL.**

Figure 4: Encoding heuristics in our DSL. We show sink nodes in □; source nodes enforcing behavior of split nodes in □ and source nodes enforcing behavior of pick nodes in □; copy nodes in □; and split nodes with limited outgoing capacity in □. The edge colors show type 2 explanations: more intense red (blue) edges show there are more samples in the subspace that only the heuristic (optimal) uses. In (a), DP uses the shortest path for the demand between 1⇝3 and the optimal does not. In (b), we see FF places a large ball (B_0) in the first bin, causing it to have to place the last ball differently, too. We used 3000 samples for each explanation. $\mathcal{X}$Plain took 20 minutes to produce each figure.

and we use "pick nodes" with limited capacity that only allow a ball to be assigned to a *single* bin (Fig. 4b) to model FF.

We prove that we can represent *any* linear or mixed integer problem through a small set of node behaviors (our abstraction is sufficient) in App. A.

We can easily compile node behaviors into efficient optimizations. Our encoding allows us to solve the optimization faster compared to the hand-coded optimization: our DSL allows us to find redundant constraints and variables. This, in turn, reduces the number of variables and constraints MetaOpt adds in its re-writes[1]. We have implemented a complete DSL in a LINQ [41]-style language: compared to the original MetaOpt implementation, the compiled DSL analyzes our DP example 4.3× faster. MetaOpt does not re-write FF, and we do not provide any run-time gains in that case.

Open questions. We can describe any heuristic that MetaOpt can analyze in our DSL. To support other analyzers (e.g., [16]) we may need to change our compiler and add other node behaviors. We also need to understand what metadata the user can (or should) provide to enable $\mathcal{X}$Plain. This may require a co-design with $\mathcal{X}$Plain's other components.

Although we have proved that any linear problem can be mapped to our DSL (App. A), that does not mean such a mapping is the most efficient representation of the heuristic in the DSL. We need further research to formalize and guide users in how to do so and optimize their representations.

5.2 The adversarial subspace generator

Random search cannot find adversarial subspaces (it may not even find an adversarial point [35]). We propose an algorithm where we extrapolate from the heuristic analyzer's output and: (1) use the analyzer to find an adversarial example; (2) find the adversarial subspace around that example; (3) exclude that subspace and repeat until we can no longer find an adversarial example (where the heuristic significantly underperforms) outside all of the subspaces we have found so far.

To find each adversarial subspace, we first find a rough candidate region: we sample in a cubic area around the initial

[1] Gurobi's pre-solve can also do this, but it changes the variable names, making it hard to connect them back to the original problem.

adversarial point given by a heuristic analyzer and expand our sampling area based on the density of adversarial (bad) samples we find in each direction. We define these "directions" based on where the sub-cube (slice) lies with respect to the initial adversarial point that MetaOpt found. We stop when the density of bad samples drops in all possible expansion directions (Fig. 5a).

We go "slice by slice" when we investigate the cubic region around the initial bad sample because the adversarial subspace may not be uniformly spread around the initial point. We extend our sampling regions only around the slices where the density of bad samples is high. We pick the number of samples we use based on the DKW inequality [33].

These subspace boundaries we have so far are not exact: how big we pick our slices and how much we expand them in each iteration influence how many false positives fall into the subspace. We refine the subspace based on an idea from prior work in diagnosis [13]. We train a regression tree that predicts the performance gap on samples in our rough subspace. The predicates that form the path that starts at the root of this tree and reaches the leaf that contains the initial bad sample more accurately describe the subspace (Fig. 5b).

The significance checker ensures the subspaces we find are statistically significant: the points in a subspace cause a higher performance gap compared to those immediately outside it. We only report those subspaces with a low-p-value (less than 0.05) as adversarial.

We use the Wilcoxon signed-rank test [44], which allows for dependant samples — the subspace fully describes what points are inside and what points are not (the samples in the two pools are dependent). We find subspaces for DP and VBP with p-values 2×10^{-60} and 8×10^{-11}, respectively.

Our approach allows us to find all *statistically significant* subspaces that meet our exploration granularity. If we do not include an adversarial input in a subspace (if it is outside of the region we explored), the analyzer will find it in the next iteration. Users can control $\mathcal{X}$Plain's ability to find all adversarial scenarios: they can use smaller cube-sizes to explore the space in more detail but it comes at the cost of a slower runtime. They can also elect to include those parts of the initial subspaces $\mathcal{X}$Plain finds (before we apply the decision tree) as part of MetaOpt's decision space (if they do so they need to include the number of times they are willing to re-examine an area to avoid an infinite cycle — there may be regions that are not statistically significant and $\mathcal{X}$Plain would revisit them if they contain a input instance that produces a high gap).

Open questions. The decision tree helps us identify predicates (of the form $f \geq t$ where f is a feature and t a threshold) that describe a subspace. What features we train the tree on influence what predicates we can get. On small instances we can use raw inputs but on larger instances this would require a deep decision tree to fully describe the space — the output becomes computationally more difficult to use in the next step (step (3) above). We need to define functions $\mathcal{F}(\mathcal{I})$ of the input $\mathcal{I}$ that allow us to describe these subspaces efficiently and which we can use in the analyzers to execute step (3) (i.e., where we exclude a subspace and re-run the analyzer).

It may be better if we apply the adversarial subspace generator (steps (1)-(3) above) directly to the "projected" input space: where each function $\mathcal{F}(\mathcal{I})$ describes one dimension of the m-dimensional space (note, m need not be the same as the dimensions of the input space $\mathcal{I}$). If the space defined by the adversarial subspaces is sparse this approach may allow us to find these adversarial subspaces more efficiently.

We may need additional mechanisms to help scale $\mathcal{X}$Plain — it may take a long time to find adversarial subspaces if we analyze a large problem instance or if there are many disjoint subspaces.

5.3 The explainer

We hypothesize that the inputs in a contiguous subspace share the same root cause for why they cause the heuristic to underperform. This is where a network-flow-based DSL explicitly encoding the decisions of the heuristic and the benchmark algorithm proves useful. We run samples from within each contiguous subspace through the DSL and score edges based on if: (1) both the benchmark and the heuristic send flow on that edge (score = 0); (2) only the benchmark sends flow (score = 1); or (3) only the heuristic sends flow (score = -1).

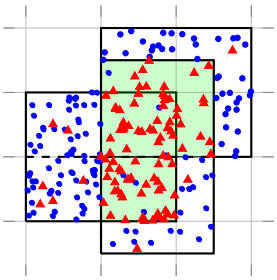
Such a "heatmap" of the differences between the benchmark and the heuristic shows how inputs in the subspace interfere with the heuristic. In Fig. 4a, in a given subspace with 3000 samples, all pinnable demands share the *same* shortest path (red arrows in 1-2-3 path), and the optimal routes them through alternative paths (blue arrows in 1-4-5-3 path).

Open questions. As the instance size (the scale of the problem we want to analyze) grows, the above heatmap may become harder to interpret. We need mechanisms that allow us to summarize the information in this heatmap in a way that the user can interpret and use to improve their heuristic.

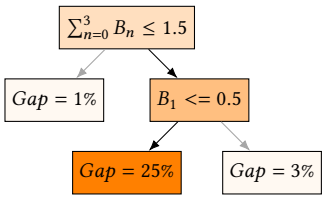
The heuristic and benchmark also differ in how much flow they route on each edge. We need to define the appropriate data structure to represent this information to a user so that they are interpretable and actionable.

5.4 The generalizer and instance generator

We can enable operators to improve their heuristics or know when to apply mitigations if we can extrapolate from the type 1 and 2 explanations to form type 3: what properties in the adversarial inputs cause the heuristic to underperform and what aspects of the problem instance exacerbate it? We need to find trends across instance-based information and find instance-agnostic explanations for why the heuristic underperformed.



(a) Identifying dense adversarial slices.



(b) Refinement by regression tree.

Adversarial subspaces: $\cup_i D_i$

$$D_i = \cup_j \left\{ \mathbf{X} \in \mathbb{R}^{+4} \;\middle|\; \begin{bmatrix} \mathbf{A} \\ \mathbf{T}_i \end{bmatrix} \mathbf{X} \leq \begin{bmatrix} \mathbf{C}_j^i \\ \mathbf{V}_i \end{bmatrix} \right\}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{I}_{4\times4} \\ -\mathbf{I}_{4\times4} \end{bmatrix}, \quad \mathbf{X} = [B_0 \quad B_1 \quad B_2 \quad B_3]^T$$

$$D_0 : \mathbf{C}_0^0 = [0.01\ 0.51\ 0.51\ 0.51\ 0\ -0.49\ -0.49\ -0.49]^T$$

$$\mathbf{T}_0 = \begin{bmatrix} -1 & -1 & -1 & -1 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{V}_0 = [-1.5 \quad 0.5]^T$$

(c) The adversarial subspaces for FF.

Figure 5: The adversarial subspace generator: (a) finds a rough subspace and separates bad samples (▲) from good ones (●); (b) it trains a regression tree on these samples and uses it to refine the subspace and produces (c). We show the first subspace (D_0) for our FF example in (c). Here, C_j^i encodes the rough subspace and T_i and V_i the path in the regression tree.

To discover patterns, we need to consider a diverse set of instances and identify trends in the outputs of the subspace generator and the explainer. We build an *instance generator* that uses the problem description in the DSL to create such instances and feeds them into the pipeline.

We imagine the generalizer would contain a "grammar" that uses the metadata the user provides through the DSL along with the network flow structure to describe trends in the instance-based explanations. For example, one may consider this predicate from a hypothetical grammar:

$$\texttt{increasing}(\mathcal{P}) : \forall a, b \mid a, b \in \mathcal{P} \;\&\; |a| \geq |b| \rightarrow gap(a) \geq gap(b)$$

With such a grammar, a generalizer can go through the observations on the samples the instance generator produced and check if the predicates in the grammar are statistically significant. For example, if $\mathcal{P}$ describes the set of shortest paths of pinnable demands in DP, the generalizer might produce `increasing`$(\mathcal{P})$ for why DP underperforms — this predicate suggests that the gap is larger when the shortest path of the pinnable demands is longer.

Open questions. One may envision a solution similar to enumerative synthesis [3, 18, 20], which searches through the grammar, finds all predicates that hold for a particular heuristic, and forms clauses that explain the heuristic's behavior. We need more work to define the generalizer's grammar and how to build valid clauses from them.

6 Related work

To our knowledge, this is the first work that focuses on a *general* framework to provide more insights into the outputs of heuristic analysis tools [16, 35] and provides an explainability feature for these tools. We build on prior work:

Domain customized performance analyzers. The work we do in $\mathcal{X}$Plain also applies to custom performance analyzers, which only apply to specific heuristics [5–7].

Explainable AI. $\mathcal{X}$Plain resembles prior work in explainable AI, which provided more context around what different ML models predict [31, 38, 42]. Parts of our solution (including the three types) are inspired by these works [4, 8, 37].

Enumerative Synthesis. This field generates programs that meet a specification through systematic enumeration of possible program candidates [3, 18, 20]. We believe these ideas can help us to design the generalizer.

Large Language Models (LLMs). we may be able to use LLMs [46] for various parts of our designs these include: to generate the DSL, to summarize Type 2 explanations, and to generate the grammer we need to produce Type 3 explanations. But LLMs are prone to hallucination [21, 29] and also require additional step-by-step mechanisms to guide them [26, 43]. We may be able to build a natural language interface that can help us automatically generate the DSL. Such an interface will enable non-experts to more easily use $\mathcal{X}$Plain. This, too, is an interesting topic for future work.

7 Acknowledgements

We would like to thank Besmira Nushi, Ishai Menache, Konstantina Mellou, Luke Marshall, Amin Khodaverdian, Chenning Li, Joe Chandler, and Weiyang Wang for their valuable comments. We also thank the HotNets program committee for their valuable feedback.

References

[1] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2022. Automating network heuristic design and analysis. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 8–16.

[2] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2024. Towards provably performant congestion control. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 951–978. https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M.K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. *Proceedings of the International Conference on Formal Methods in Computer-Aided Design* (2013). https://doi.org/10.1109/FMCAD.2013.6679385

[4] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.

[5] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal Methods for Network Performance Analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 645–661.

[6] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. 2022. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 177–192. https://doi.org/10.1145/3544216.3544223

[7] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 1–16.

[8] Behnaz Arzani, Kevin Hsieh, and Haoxian Chen. 2021. Interpretable feedback for AutoML and a proposal for domain-customized AutoML for networking. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*. 53–60.

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

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

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

[12] Stephen P Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.

[13] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. 2004. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 36–43.

[14] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 217–232. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz

[15] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. ACM, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

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

[17] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 2004. Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis. In *Formal Methods in Computer-Aided Design*, Alan J. Hu and Andrew K. Martin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–289.

[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 62–73.

[19] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 735–749. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex

[20] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1159–1174.

[21] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232* (2023).

[22] Karthick Jayaraman, Nikolaj Bjorner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating Datacenters at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. ACM, New York, NY, USA, 200–213. https://doi.org/10.1145/3341302.3342094

[23] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies*. Technical Report MSR-TR-2014-102. Microsoft.

[24] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive Verification of DNS Configurations. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 310–328. https://doi.org/10.1145/3387514.3405871

[25] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding Network Misconfigurations by Automatic Template Inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 999–1013. https://www.usenix.org/conference/nsdi20/presentation/kakarla

[26] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Kaya Stechly, Mudit Verma, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. 2024. LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. *arXiv preprint arXiv:2402.01817* (2024).

[27] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[28] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 15–27. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[29] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).

[30] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 499–512.

[31] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.* 41, 4 (aug 2011), 290–301. https://doi.org/10.1145/2043164.2018470

[33] P. Massart. 1990. The Tight Constant in the Dvoretzky-Kiefer-Wolfowitz Inequality. *The Annals of Probability* 18, 3 (1990), 1269–1283. https://doi.org/10.1214/aop/1176990746

[34] Ken L. McMillan. 2002. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Computer Aided Verification*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–264.

[35] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. 2024. Finding Adversarial Inputs for Heuristics using Multi-level Optimization. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 927–949. https://www.usenix.org/conference/nsdi24/presentation/namyar-finding

[36] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. Heuristics for vector bin packing. *research. microsoft. com* (2011).

[37] P Jonathon Phillips, P Jonathon Phillips, Carina A Hahn, Peter C Fontana, Amy N Yates, Kristen Greene, David A Broniatowski, and Mark A Przybocki. 2021. Four principles of explainable artificial intelligence. (2021).

[38] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.

[39] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 748–761. https://doi.org/10.1145/3452296.3472925

[40] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. https://doi.org/10.1145/3341302.3342088

[41] Mads Torgersen. 2007. Querying in C# how language integrated query (LINQ) works. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 852–853.

[42] Sandra Wachter, Brent Mittelstadt, and Chris Russell. 2017. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harv. JL & Tech.* 31 (2017), 841.

[43] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (2023).

[44] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

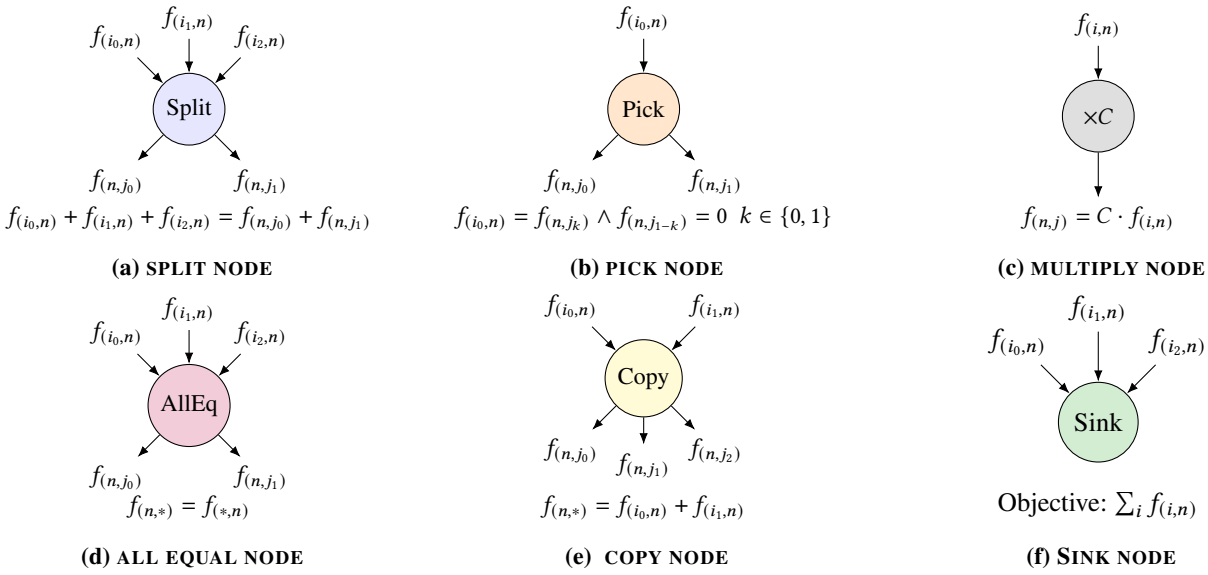
[45] Gerhard J Woeginger. 1997. There is no asymptotic PTAS for two-dimensional vector packing. *Inform. Process. Lett.* 64, 6 (1997), 293–297.

[46] Zheyu Yan, Yifan Qin, Xiaobo Sharon Hu, and Yiyu Shi. 2023. On the Viability of Using LLMs for SW/HW Co-Design: An Example in Designing CiM DNN Accelerators. In *2023 IEEE 36th International System-on-Chip Conference (SOCC)*. 1–6. https://doi.org/10.1109/SOCC58585.2023.10256783

[47] Hongkun Yang and Simon S. Lam. 2016. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (April 2016), 887–900. https://doi.org/10.1109/TNET.2015.2398197

[48] Yinlei Yu, Pramod Subramanyan, Nestan Tsiskaridze, and Sharad Malik. 2014. All-SAT Using Minimal Blocking Clauses. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. 86–91. https://doi.org/10.1109/VLSID.2014.22

[49] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 241–255. https://www.usenix.org/conference/nsdi20/presentation/zhang-peng



(a) SPLIT NODE **(b) PICK NODE** **(c) MULTIPLY NODE**

(d) ALL EQUAL NODE **(e) COPY NODE** **(f) SINK NODE**

Figure 6: Different node types in $\mathcal{X}$Plain's DSL.

A Formalizing $\mathcal{X}$Plain's DSL

We prove that we can model any linear optimization in $\mathcal{X}$Plain.

A.1 $\mathcal{X}$Plain's node description

PRELIMINARIES. Our network-flow-based DSL is a directed graph where we denote the set of nodes with $\mathcal{N}$ and the set of directed edges as $\mathcal{E}$. We treat each edge $(i,j) \in \mathcal{E}$ as a variable with a non-negative flow value $f_{(i,j)} \geq 0$. We impose constraints on these flow variables as needed. We define incoming edges to node $n \in \mathcal{N}$ as those edges which are directed towards n (i.e., $(i,n) \in \mathcal{E}$). Outgoing edges are those exiting n. The incoming (outgoing) traffic to a node is the sum of all flow that arrives at that node from all the incoming (outgoing) edges.

We have the following node behaviors:

SPLIT NODES ($\mathcal{N}_{split}$) split the incoming traffic between the outgoing edges (Fig. 6a). They enforce the traditional flow conservation constraints:

$$\sum_{\{i \in \mathcal{N}, (i,n) \in \mathcal{E}\}} f_{(i,n)} = \sum_{\{i \in \mathcal{N}, (n,i) \in \mathcal{E}\}} f_{(n,i)} \qquad \forall n \in \mathcal{N}_{split}$$

They can also optionally enforce (1) an upper bound on the traffic on an outgoing edge (capacity constraint) and (2) the traffic on an incoming edge to be constant.

$$f_{(n,i)} \leq C_{(n,i)} \quad C_{(n,i)} \in \mathbb{R}^+, \forall i \in \{i \in \mathcal{N}, (n,i) \in \mathcal{E}\} \quad \forall n \in \mathcal{N}_{split}$$
$$f_{(i,n)} = d_{(i,n)} \quad d_{(i,n)} \in \mathbb{R}_{\geq 0}, \forall i \in \{i \in \mathcal{N}, (i,n) \in \mathcal{E}\} \quad \forall n \in \mathcal{N}_{split}$$

PICK NODES ($\mathcal{N}_{pick}$) satisfy flow conservation *but* only allow one of the outgoing edges to carry traffic (Fig. 6b):

$$\sum_{\{i \in \mathcal{N}, (i,n) \in \mathcal{E}\}} f_{(i,n)} = \sum_{\{i \in \mathcal{N}, (n,i) \in \mathcal{E}\}} f_{(n,i)} \qquad \forall n \in \mathcal{N}_{pick}$$
$$\sum_{\{i \in \mathcal{N}, (n,i) \in \mathcal{E}\}} \mathbb{1}[f_{(n,i)} > 0] = 1 \qquad \forall n \in \mathcal{N}_{pick}$$

where $\mathbb{1}[x > 0]$ is an indicator function (=1 if $x > 0$, otherwise = 0).

MULTIPLY NODES ($\mathcal{N}_{mult}$) only have one incoming and one outgoing link. They multiply the incoming traffic by a constant $C \in \mathbb{R}^+$ before sending it out (Fig. 6c). They only satisfy flow conservation when $C = 1$.

$$f_{(n,i)} = C f_{(j,n)} \quad \forall (i,j) \in \{(i,j) \mid i,j \in \mathcal{N}, (n,i), (j,n) \in \mathcal{E}\} \; \forall n \in \mathcal{N}_{mult}$$

ALL EQUAL NODES ($\mathcal{N}_{allEq}$) require all the incoming and outgoing edges to carry the same amount of traffic (Fig. 6d):

$$f_{(n,i)} = f_{(j,n)} \quad \forall (i,j) \in \{(i,j) \mid i,j \in \mathcal{N}, (n,i), (j,n) \in \mathcal{E}\} \; \forall n \in \mathcal{N}_{allEq}$$

To make it simpler to encode a heuristic in the DSL, we also add the following node types to our DSL:

COPY NODES ($\mathcal{N}_{copy}$) copy the total incoming flow into each outgoing edge (Fig. 6e):

$$f_{(n,j)} = \sum_{\{i \in \mathcal{N}, (i,n) \in \mathcal{E}\}} f_{(i,n)} \quad \forall j \in \{j \mid j \in \mathcal{N}, (n,j) \in \mathcal{E}\} \; \forall n \in \mathcal{N}_{copy}$$

We can recreate this node's behavior if we combine split nodes and equal nodes (Fig. 7). However, using a copy node directly is more intuitive and straightforward for users, and we include it in our DSL for that reason.

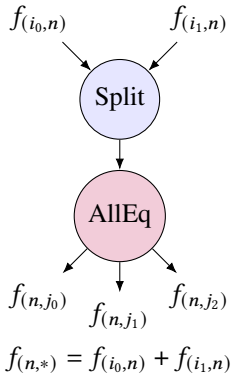


Figure 7: Recreating COPY NODE with SPLIT NODE and ALL EQUAL NODE

We use source and sink nodes to define the objective:

SOURCE NODES ($\mathcal{N}_{source}$) are special cases of split or pick nodes that represent the inputs to the problem. For example, Fig. 4a illustrates the input traffic demand modeled as source nodes that enforce split node behavior (□). Also, Fig. 4b shows the input ball sizes as source nodes with pick node behavior (□, each ball can only be placed in one bin).

SINK NODE ($\mathcal{N}_{sink}$) is a specific node that (1) only has incoming edges and (2) measures the performance of the problem as the total incoming traffic through these edges (Fig. 6f). When the DSL represents an optimization problem, the sink node is designated as the objective, and the compiler translates the value of the sink node into the optimization objective.

A.2 $\mathcal{X}$Plain can model any linear optimization

THEOREM A.1. *We can model any linear optimization (linear programming or mixed integer linear programming) as a flow network using the six node behaviors ($\mathcal{N}_{split}$, $\mathcal{N}_{pick}$, $\mathcal{N}_{mult}$, $\mathcal{N}_{allEq}$, and $\mathcal{N}_{sink}$)*

PROOF. An optimization problem maximizes (or minimizes) an objective subject to inputs that fall within a feasible space that the optimization constraints characterize. We can express a linear optimization problem as (linear programming or mixed integer linear programming):

$$\begin{aligned} \max_{\mathbf{x},\mathbf{y}} \quad & \mathbf{c}_\mathbf{x}^\mathsf{T}\mathbf{x} + \mathbf{c}_\mathbf{y}^\mathsf{T}\mathbf{y} \\ & \mathbf{A}_\mathbf{x}\mathbf{x} + \mathbf{A}_\mathbf{y}\mathbf{y} \le \mathbf{b} \\ & \mathbf{x} \ge \mathbf{0} \\ & \mathbf{y} \in \{0,1\}^{|\mathbf{y}|} \end{aligned}$$

To show that our DSL is complete, we need to show that we can capture both the feasible space and the objective correctly through our flow model for every possible linear optimization.

We first present a general algorithm to express the feasible space of any given linear optimization as a flow model and prove it is correct. Next, we show how we can use the same algorithm to express any linear objective.

How to represent the feasible space with a flow model. We can express the feasible space of any linear optimization as:

$$\mathbf{A}_\mathbf{x}\mathbf{x} + \mathbf{A}_\mathbf{y}\mathbf{y} \le \mathbf{b} \tag{1}$$

$$\mathbf{x} \ge \mathbf{0} \tag{2}$$

$$\mathbf{y} \in \{0,1\}^{|\mathbf{y}|} \tag{3}$$

where we denote matrices and vectors in bold. $\mathbf{x}$ and $\mathbf{y}$ are vectors of continuous and binary variables of size $|\mathbf{x}| \times 1$ and $|\mathbf{y}| \times 1$, respectively. $\mathbf{b}$ is a constant vector of size $|\mathbf{b}| \times 1$. $\mathbf{A}_\mathbf{x}$ and $\mathbf{A}_\mathbf{y}$ are constant matrices of sizes $|\mathbf{b}| \times |\mathbf{x}|$ and $|\mathbf{b}| \times |\mathbf{y}|$ respectively. Note that we can enforce an equality constraint as two inequality constraints (Eq. 1), and represent any integer variable as the sum of multiple binary variables. We map the variables to flows in our model.

We need to transform the above optimization before we can model it with our node behaviors:

► *Transformation 1.* The matrices $\mathbf{A}_\mathbf{x}$ and $\mathbf{A}_y$, and the vector $\mathbf{b}$ may contain negative entries. This conflicts with the non-negativity requirement of the flows in our flow model. To address this, we decompose these matrices and vector into their positive and negative components:

$$\mathbf{A}_\mathbf{x} = \mathbf{A}_\mathbf{x}^+ - \mathbf{A}_\mathbf{x}^-, \quad \mathbf{A}_\mathbf{y} = \mathbf{A}_\mathbf{y}^+ - \mathbf{A}_\mathbf{y}^-, \quad \mathbf{b} = \mathbf{b}^+ - \mathbf{b}^-$$

where all the elements in $\mathbf{A}_\mathbf{x}^+ = [a_{ij}^{(+,\mathbf{x})}]$ and $\mathbf{A}_\mathbf{x}^- = [a_{ij}^{(-,\mathbf{x})}]$ are non-negative such that **at most one** of $a_{ij}^{(+,\mathbf{x})}$ or $a_{ij}^{(-,\mathbf{x})}$ is non-zero for every $i \in \mathbb{Z}_{[0,|\mathbf{b}|)}$ and $j \in \mathbb{Z}_{[0,|\mathbf{x}|)}$. Note that $\mathbb{Z}_{[0,m)} = \{0,\dots,m-1\}$. Same holds for both (1) $\mathbf{A}_\mathbf{y}^+$ and $\mathbf{A}_\mathbf{y}^-$, and (2) $\mathbf{b}^+ = [b_i^+]$ and $\mathbf{b}^- = [b_i^-]$ over every i. All matrices have the same size as their originating matrix. After substituting these decompositions into Eq. 1, we have:

$$\mathbf{A}_\mathbf{x}^+\mathbf{x} + \mathbf{A}_\mathbf{y}^+\mathbf{y} + \mathbf{b}^- \le \mathbf{A}_\mathbf{x}^-\mathbf{x} + \mathbf{A}_\mathbf{y}^-\mathbf{y} + \mathbf{b}^+ \tag{4}$$

► *Transformation 2.* Eq. 4 and SPLIT NODEs qualitatively represent similar behaviors. SPLIT NODEs split the incoming traffic across outgoing edges and ensure the traffic on each edge does not exceed the capacity constraints. Ideally, we can

enforce the Eq. 4 constraints using a SPLIT NODEs and as a flow conservation a constraint:

$$\begin{aligned} &\mathbf{A}_{\mathbf{x}}^{+}\mathbf{x} + \mathbf{A}_{\mathbf{y}}^{+}\mathbf{y} + \mathbf{b}^{-} + \mathbf{f} = \mathbf{A}_{\mathbf{x}}^{-}\mathbf{x} + \mathbf{A}_{\mathbf{y}}^{-}\mathbf{y} + \mathbf{b}^{+} && \text{(Flow conservation)} \\ &\mathbf{0} \le \mathbf{f} && \text{(Flow constraint)} \quad (5) \end{aligned}$$

The problem is that Eq. 4 also involves coefficients associated with each variable ($\mathbf{A}$), while SPLIT NODEs do not accept weights. We address this by replacing each term (coefficient multiplied by a variable) in each of the Eq. 4 constraints with an auxiliary variable:

$$\begin{aligned} &u_{ij}^{+} = a_{ij}^{(+,\mathbf{x})} x_j,\ u_{ij}^{-} = 0 \quad \text{if } a_{ij}^{(+,\mathbf{x})} \ge 0 \quad \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{x}|)} \\ &u_{ij}^{-} = a_{ij}^{(-,\mathbf{x})} x_j,\ u_{ij}^{+} = 0 \quad \text{if } a_{ij}^{(-,\mathbf{x})} > 0 \quad \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{x}|)} \\ &v_{ij}^{+} = a_{ij}^{(+,\mathbf{y})} y_j,\ v_{ij}^{-} = 0 \quad \text{if } a_{ij}^{(+,\mathbf{y})} \ge 0 \quad \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{y}|)} \\ &v_{ij}^{-} = a_{ij}^{(-,\mathbf{y})} y_j,\ v_{ij}^{+} = 0 \quad \text{if } a_{ij}^{(-,\mathbf{y})} > 0 \quad \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{y}|)} \end{aligned} \quad (6)$$

We define $\mathbf{U}^{+} = [u_{ij}^{+}]$, $\mathbf{U}^{-} = [u_{ij}^{-}]$, $\mathbf{V}^{+} = [v_{ij}^{+}]$, and $\mathbf{V}^{-} = [v_{ij}^{-}]$. We can then express Eq. 5 in terms of these auxiliary variables:

$$\mathbf{U}^{+}\mathbf{d}_{\mathbf{x}} + \mathbf{V}^{+}\mathbf{d}_{\mathbf{y}} + \mathbf{b}^{-} + \mathbf{f} = \mathbf{U}^{-}\mathbf{d}_{\mathbf{x}} + \mathbf{V}^{-}\mathbf{d}_{\mathbf{y}} + \mathbf{b}^{+},\ \ \mathbf{0} \le \mathbf{f}$$

where $\mathbf{d}_{\mathbf{x}}$ and $\mathbf{d}_{\mathbf{y}}$ are vectors with all elements equal to 1 and sizes of $|\mathbf{x}| \times 1$ and $|\mathbf{y}| \times 1$ respectively. This is because each of the auxiliary variables u_{ij} or v_{ij} appear in exactly one inequality constraint.

► *Transformation 3.* We encounter a problem to enforce the constraints in Eq. 6 using MULTIPLY NODE for u_{ij} and v_{ij}: each MULTIPLY NODE has only one input and one output edge. Each edge also corresponds to one variable. This means each variable can appear in at most two constraints, corresponding to the two nodes at the two ends of the edge. However, the variables in Eq. 6 appear more than twice (for example, x_j can appear up to $|\mathbf{b}|$ times.)

We address this by introducing additional variables and constraints:

$$\begin{aligned} &u_{ij}^{+} = a_{ij}^{(+,\mathbf{x})} x_{ij}^{+},\ \ u_{ij}^{-} = a_{ij}^{(-,\mathbf{x})} x_{ij}^{-} && \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{x}|)} \\ &v_{ij}^{+} = a_{ij}^{(+,\mathbf{y})} y_{ij}^{+},\ \ v_{ij}^{-} = a_{ij}^{(-,\mathbf{y})} y_{ij}^{-} && \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{y}|)} \\ &x_{ij}^{+} = x_{ij}^{-} = x_j && \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{x}|)} \\ &y_{ij}^{+} = y_{ij}^{-} = y_j && \forall i \in \mathbb{Z}_{[0,|\mathbf{b}|)}, \forall j \in \mathbb{Z}_{[0,|\mathbf{y}|)} \end{aligned}$$

With these modifications, each variable x_{ij}^{+} and x_{ij}^{-} appears in exactly two constraints (same for y).

The final resulting optimization after all the transformations is:

$$\mathbf{U}^{+}\mathbf{d}_{\mathbf{x}} + \mathbf{V}^{+}\mathbf{d}_{\mathbf{y}} + \mathbf{b}^{-} + \mathbf{f} = \mathbf{U}^{-}\mathbf{d}_{\mathbf{x}} + \mathbf{V}^{-}\mathbf{d}_{\mathbf{y}} + \mathbf{b}^{+},\ \ \mathbf{0} \le \mathbf{f} \quad (7)$$

$$u_{ij}^{+} = a_{ij}^{(+,\mathbf{x})} x_{ij}^{+} \qquad \forall i\ \forall j \quad (8)$$

$$x_{ij}^{-} = \frac{1}{a_{ij}^{(-,\mathbf{x})}} u_{ij}^{-} \quad \text{if } a_{ij}^{(-,\mathbf{x})} > 0 \qquad \forall i\ \forall j \quad (9)$$

$$v_{ij}^{+} = a_{ij}^{(+,\mathbf{y})} y_{ij}^{+} \qquad \forall i\ \forall j \quad (10)$$

$$y_{ij}^{-} = \frac{1}{a_{ij}^{(-,\mathbf{y})}} v_{ij}^{-} \quad \text{if } a_{ij}^{(-,\mathbf{y})} > 0 \qquad \forall i\ \forall j \quad (11)$$

$$x_{ij}^{+} = x_{ij}^{-} = x_j \qquad \forall i\ \forall j \quad (12)$$

$$y_{ij}^{+} = y_{ij}^{-} = y_j \qquad \forall i\ \forall j \quad (13)$$

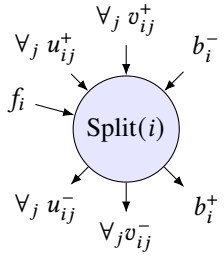
$$\mathbf{x} \ge \mathbf{0} \quad (14)$$

$$\mathbf{y} \in \{0,1\}^{|\mathbf{y}|} \quad (15)$$

where for each of the equations above, notation $\forall i\ \forall j$ means all the possible i and j values should be considered according to the specific constraints or conditions given for each equation.

► *Constructing the flow model.* We can encode the above constraints using a flow model. We first create one edge per variable and then enforce each constraint using one node:

(S1) We encode Eq. 7 using SPLIT NODEs. We will have a node for each possible i. The inputs to each node are (1) one edge per variable on the left-hand side of the constraint ($\mathbf{U}^{+}$ and $\mathbf{V}^{+}$), (2) one edge with a constant rate $\mathbf{b}^{-}$, and (3) one additional edge associated with $\mathbf{f}$. The outputs are (1) one edge per variable on the right-hand side of the constraint ($\mathbf{U}^{-}$ and $\mathbf{V}^{-}$) and (2) one additional edge with constant rate $\mathbf{b}^{+}$. Fig. 8 shows how this encoding is done.



$$\textstyle\sum_j [u_{ij}^{+} + v_{ij}^{+}] + b_i^{-} + f_i = \sum_j [u_{ij}^{-} + v_{ij}^{-}] + b_i^{+}$$

Figure 8: Step 1 of the encoding: SPLIT NODE for i. There will be a SPLIT NODE for each possible $i \in \mathbb{Z}_{[0,|\mathbf{b}|)}$. If a variable is 0, we do not need to assign it to the node. There are at most $|\mathbf{x}|$ arrows present for u_{ij}^{+} and u_{ij}^{-} since at most one of $a_{ij}^{(-,\mathbf{x})}$ or $a_{ij}^{(+,\mathbf{x})}$ is non-zero. Similarly, there are at most $|\mathbf{y}|$ arrows present for y_{ij}^{+} and y_{ij}^{-}.

(S2) We express Eq. 8 – 11 using MULTIPLY NODEs. The $\mathbf{U}^-$ edges originate from SPLIT NODEs to these MULTIPLY NODEs while $\mathbf{U}^+$ edges are in the opposite direction. So, the node that models Eq. 8 has x_{ij}^+ as its input edge and u_{ij}^+ as its output edge. Conversely, the input edge is u_{ij}^- and the output edge is x_{ij}^- for Eq. 9 (same holds for y and v). Fig. 9 shows this step.

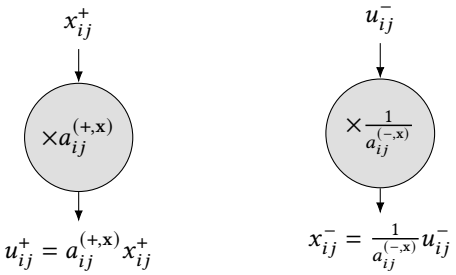


Figure 9: Step 2 of the encoding. There will be a MULTIPLY NODE for each possible i and j. At most of these two MULTIPLY NODEs will be needed since at most one of $a_{ij}^{(-,\mathbf{x})}$ or $a_{ij}^{(+,\mathbf{x})}$ is non-zero.

(S3) We model Eq. 12 – 13 using ALL EQUAL NODEs. Note that for a fixed i and j, since at most one of $a_{ij}^{(-,\mathbf{x})}$ and $a_{ij}^{(+,\mathbf{x})}$ is non-zero, at most of the equations in Eq. 8 and Eq. 9 are needed for that i and j (same holds for Eq. 10 and Eq. 11). Consequently, at most of x_{ij}^+ and x_{ij}^- is needed in Eq. 12 (same holds for y_{ij}^+ and y_{ij}^- in Eq. 13). The x_j and x_{ij}^-s are input edges and x_{ij}^+s are the output edges (same for y). Fig. 10 illustrates this step.

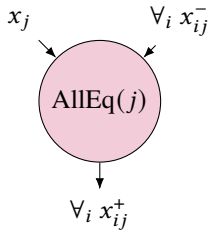


Figure 10: Step 3 of the encoding. There will be a ALL EQUAL NODE for each possible $j \in \mathbb{Z}_{[0,|\mathbf{x}|)}$.

(S4) The input variables are the variables in $\mathbf{x}$ and $\mathbf{y}$. We represent binary variables in Eq. 15 using PICK NODEs. It has one incoming edge with a constant rate of 1 and two outgoing edges. One of the outputs corresponds to the binary variable. If the node selects that specific edge to carry the flow, the binary variable is 1. Otherwise, it is 0. Eq. 14 is inherently satisfied as flows are all non-negative.

This flow model provably captures the optimization's feasible space as there is a one-to-one correspondence between the constraints in the optimization and the constraints enforced by the nodes.

How to capture the optimization objective. We can express the objective of any linear optimization as $\max_{\mathbf{x},\mathbf{y}} \ \mathbf{c}_\mathbf{x}^\intercal \mathbf{x} + \mathbf{c}_\mathbf{y}^\intercal \mathbf{y}$ where $\mathbf{c}_\mathbf{x}$ and $\mathbf{c}_\mathbf{y}$ are constant vectors. We can reformulate and add a constraint that enforces $p = \mathbf{c}_\mathbf{x}^\intercal \mathbf{x} + \mathbf{c}_\mathbf{y}^\intercal \mathbf{y}$, so the objective of the optimization changes to maximizing p. Then, we can use similar transformations, as we explained before, to capture this constraint within the flow model. We add a sink node that has one incoming edge p. This way, we can express any linear optimization objective with our model.

□