Generalization in Learned Congestion Control: Decision Trees, Deep RL, and LLM-Guided Evolution

Duy Nguyen
Stanford University
duynguy@stanford.edu

Andrew Grant
Stanford University
agrant27@stanford.edu

Armaan Abraham
Stanford University
armaana@stanford.edu

ABSTRACT

Remy [12] demonstrated that computer-generated congestion control algorithms (CCAs) could outperform decades of human design within their training distribution. However, Remy's decision tree CCAs generalize poorly to network conditions outside their training range. Despite this being discovered over a decade ago, we are not aware of prior work that evaluates CCAs generated by other learning schemes on Remy's generalization graphs—training on the same environments, with the same observation and action spaces. We implement two additional CCA learning schemes within the Remy simulation and evaluation framework—a deep reinforcement learning (RL) agent and a LLM-guided code evolution system—and compare their performance on link rates outside the training distribution to that of Remy CCAs. We find that different CCA learning schemes exhibit generalization advantages that depend on whether the link rate is greater or less than the training link rate range. Ultimately, none of the CCA learning schemes we evaluate are superior to the others in both their in-distribution and out-of-distribution performance, potentially indicating a more fundamental tradeoff between these two qualities.

1 INTRODUCTION

We evaluate two additional policy representations under Remy's generalization setup. Keeping the observation space fixed, we ask: **how do a neural network and a large language model (LLM)-guided code evolution system compare to Remy's decision trees on out-of-distribution generalization?**

We compare three representations:

(1) **Decision trees** (Remy): A WhiskerTree maps observation bins to actions, optimized by evolutionary search over tree structure.
(2) **Neural networks** (PPO): A neural network maps the same observations to the same actions, trained by proximal policy optimization (PPO) [10] in Remy's C++ simulator.
(3) **Source code** (AlphaCC): An LLM generates source-code policies mapping observations to actions, evolved by mutation in a Python simulator and translated to C++ for final evaluation.

2 RELATED WORK

Remy [12] pioneered automated CCA design, with decision trees as the underlying model class. Sivaraman et al. [11] further examined their generalization, finding that trees trained on narrow ranges degrade on wider conditions while broader training ranges trade peak performance for robustness.

Several learning-based approaches followed: PCC [5] used online learning for utility maximization; Aurora [7] applied deep RL to congestion control; Yan et al. [13] built the Pantheon benchmarking platform for systematic CCA comparison. These demonstrated that neural networks can learn congestion control but did not systematically compare generalization across learning schemes within the same simulation and evaluation framework.

CCAC [2] introduced formal CCA verification using SMT solvers. CCmatic [3] extended this to automated synthesis with formal guarantees, but only within a fixed window-adjustment template.

AlphaEvolve [9] demonstrated LLM-guided code evolution for mathematical optimization. Our approach adapts this paradigm to congestion control, using simulation-based fitness instead of mathematical verification.

Copa [1] and BBR [4] represent hand-designed CCAs that encode domain knowledge about delay-based and model-based congestion control, respectively.

3 METHODS

3.1 Simulation Setup and Evaluation

Simulation setup. We use the same dumbbell network topology as in Remy [12], consisting of two senders which randomly start and stop flows. Each flow's duration is sampled from an exponential distribution, and during a flow a sender chooses when and how many packets to send. For each set of CCAs we evaluate, this decision of whether to send a packet at a given time is determined by the congestion window and the minimum time interval between sent packets, and these are directly updated by the action chosen by each policy. To compare the generalization of each CCA learning scheme, we fix all network conditions of the simulator except for the link rate, which we vary in both the learning and evaluation process.

Table 1: Network conditions used for training and evaluation.

Quantity	Design range	Distribution
n max senders	2	uniform
"on" process	mean 5 s	exponential
"off" process	mean 5 s	exponential
link speed	varies	uniform
round-trip time	150 ms	uniform
queue capacity	unlimited	—

Utility and normalized score. Remy and PPO are trained using the utility

$$U = \log(\text{throughput}) - \delta \cdot \log(\text{delay}), \tag{1}$$

with $\delta = 1$ in our experiments, so training directly optimizes the throughput-delay tradeoff. Because evaluation spans a wide range of link rates, raw utility values are not directly comparable across capacities. For cross-rate comparison, plotting, and aggregation, we therefore use the normalized score

$$S = \log_2\left(\frac{\text{throughput}}{\text{capacity}}\right) - \log_2\left(\frac{\text{delay}}{\text{min\_rtt}}\right). \tag{2}$$

This preserves the same throughput-delay structure as the training utility, but measures performance relative to link capacity and minimum RTT. A score of 0 means perfect utilization with no queuing delay.

Whisker Tree representation. Remy represents a congestion-control algorithm as a lookup table from sender state to a three-parameter action. A RemyCC consists of piecewise-constant rules, each mapping a rectangular region of the state space to a fixed action. During optimization, Remy repeatedly refines heavily used regions, producing an octree-like partition that concentrates representational capacity where it matters most while keeping the controller compact.

Observation space. In the original Remy formulation, the sender state consists of three variables: `ack_ewma`, an exponentially weighted moving average (EWMA) of ACK inter-arrival time; `send_ewma`, an EWMA of the spacing between echoed sender timestamps; and `rtt_ratio`, the ratio of the current RTT to the minimum RTT seen on the flow. This state is updated on each ACK arrival and reset at the start of a new flow.

Action space. Each Remy rule outputs a three-dimensional action (m, b, r), where m is a multiplicative update to the congestion window, b is an additive window adjustment, and r is a minimum inter-send interval. Together, these control window growth and pacing.

Evaluation protocol. All results use Remy's original C++ simulator (`sender-runner`). Unless otherwise noted, we evaluate on 99 log-spaced link rates from 0.1–100 packets/ms (1–1000 Mbps), with RTT = 150 ms, 2 senders, mean on/off duration 5000 ms, and infinite buffers.

Training and evaluation. All three methods are evaluated in the same C++ simulator. Remy and PPO train directly in C++. AlphaCC trains in Python but is translated to C++ for evaluation: a fixed prompt provides GPT-5.3-codex with the Python source code and the C++ `Rat` class interface, from which it generates a compilable `AlphaCCRat` subclass (mirroring the `NeuralRat` integration for PPO). If the generated code fails to compile, the translation is retried up to three times before the candidate is rejected.

3.2 PPO: Deep Reinforcement Learning

For our PPO agent, we consider a single neural network that is executed locally for each sender that ultimately maps local observations to a decision about when to send the next packet. The parameters of this neural network are identical for each sender and unchanged throughout a given simulation. The neural network accepts the same inputs and uses the same discretized action space for each action dimension as the Remy decision tree.

3.2.1 Slotting our PPO agent into the Remy codebase. We implemented a new `NeuralRat` class in the original Remy codebase as a subclass of the existing `Rat` class used by the Remy decision tree, isolating the decision-making algorithm from the surrounding agent interface and simulation infrastructure. This subclass inherits the same packet-sending logic, memory tracking, and simulation interface, and overrides only the method that maps the current memory state to window increment, window multiple, and intersend values. Where the base `Rat` performs a lookup in the whisker tree to obtain these values, `NeuralRat` runs a forward pass through the neural network and samples from the distribution described by its output, described next.

3.2.2 Network inputs and action selection. The network takes as input the four-dimensional observation $\mathbf{x}$ = (sewma, rewma, rtt_ratio, slow_rewma) and produces a matrix of unnormalized scores $\mathbf{Z} \in \mathbb{R}^{K \times N_k}$, where $K = 3$ is the number of action dimensions (window increment, window multiple, and minimum intersend) and N_k is the number of discrete bins for dimension k. Each row is independently normalized into a probability distribution via a temperature-scaled softmax:

$$\pi(a_i^k) = \frac{\exp(z_{k,i}/\tau)}{\sum_j \exp(z_{k,j}/\tau)}, \tag{3}$$

where τ is a temperature parameter that controls the entropy of each output distribution. At each decision point, an action index $a^k \sim \text{Categorical}(\pi(a_1^k), \ldots, \pi(a_{N_k}^k))$ is sampled independently for each dimension k, selecting a specific discretized value for each of these action dimensions.

3.2.3 Training: an overview. To learn the parameters of our neural network, we use proximal policy optimization, which is a member of a broader class of policy gradient methods. Abstractly, in this process, at every training iteration, the single current neural network is loaded into all senders in a fixed number of simulations (i.e., rollouts), and the final utility is recorded for each rollout. Because actions are sampled from a distribution defined by our network at a given input, this will naturally yield diversity in actions at each input that can be used to generate a gradient signal.

Then, the parameters of the network are updated by slightly increasing the probability of actions in rollouts with a final utility that is greater than the mean utility across all rollouts in the current iteration and decreasing the probability of actions in rollouts with a utility less than the mean. Because the entire domain of the action distribution is defined by the policy network, we can increase the probability of sampling an action from our policy network in a fully differentiable way by increasing the element of the score vector, $z_{k,i}$, corresponding to that action and decreasing all other elements of $\mathbf{z_k}$ along the same action dimension. This process of upweighting and downweighting actions is applied equally to all senders in each rollout based on the aggregate final utility of the rollout.

3.2.4 Training: the specifics. During each rollout, every sender records its experience as a sequence of events, where each event consists of an observation, the sampled action, the log-probability of that action under the current policy, the final utility of the rollout, and the total number of events across all senders in the rollout, which we call the rollout event count. We assume the joint probability over actions factorizes across the three dimensions, so the joint log-probability is $\log \pi(a \mid s) = \sum_k \log \pi(a^k \mid s)$.

To compute the update, a batch of events is sampled uniformly from the experience collected in the current iteration. An "advantage" $\hat{A}_i$ for each event is obtained by normalizing the utilities to zero mean and unit variance across all events in the batch. For each sampled event, the network is evaluated on the recorded observation to produce new log-probabilities for the actions that were taken. The probability ratio between the new and old policies, $r_i = \exp(\log \pi_\theta(a \mid s) - \log \pi_{\theta_{\text{old}}}(a \mid s))$, is used to define the PPO clipped surrogate loss:

$$L_i^{\text{PPO}} = -\min\big(r_i \hat{A}_i,\ \text{clip}(r_i,\ 1-\epsilon,\ 1+\epsilon)\,\hat{A}_i\big), \tag{4}$$

Table 2: PPO Hyperparameters

Parameter	Value
Optimizer	AdamW [8]
Training steps	432
Rollouts per training step	72
Hidden dimension	256
Number of hidden layers	2
Activation function	GeLU [6]
Learning rate	3×10^{-4}
Weight decay	1×10^{-3}
Gradient clipping max norm	1
PPO clipping threshold ϵ	0.2
Batch size (events)	4.5×10^6
Gradient updates per training step	5

where ϵ is the clipping threshold. Each event is assigned a weight w_i equal to the average of the rollout event counts across the batch divided by this event's rollout event count to prevent rollouts with more events from being weighted more heavily in the loss. The overall loss, minimized over the batch of size B, is:

$$L = \sum_{i=1}^{B} w_i\, L_i^{\text{PPO}}. \tag{5}$$

We initially found that training with raw observations performed poorly. In an attempt at improvement, we maintained a running mean μ_d and standard deviation σ_d for each observation dimension d, updated across all experience collected so far, and normalized each input as $\hat{x}_d = (x_d - \mu_d)/\sigma_d$ before passing it to the network. This improved final performance and all results we report use this normalization. We list all PPO-related hyperparameters in Table 2.

3.3 AlphaCC: LLM-Guided Code Evolution

AlphaCC evolves Python source code directly, inspired by AlphaEvolve [9]. Candidates are evaluated in a Python reimplementation of Remy's event-driven simulator, calibrated to ≤3% mean error against the C++ `sender-runner`.[1] Each candidate is a Python function, `evolved_policy(memory)`, that maps the same four-dimensional EWMA observation to the same action triple. Unlike Remy's stateless tree lookup or PPO's feedforward pass, these functions can maintain persistent state across invocations via function attributes (e.g., phase variables, bandwidth estimates).

3.3.1 Evolution loop. The evolutionary process begins with six hand-written Python policies spanning loss-based (AIMD), delay-based (Copa [1]), rate-based

[1] Calibration script: `cs244c-submission/scripts/calibrate.py`. Evaluated across 4 whisker trees × 9 link rates; per-tree mean errors range from 2.4% (1x) to 3.8% (20x).

(BBR-inspired [4]), and minimal (constant) families. In each of 15 generations, five new candidates are produced. For each, a parent is selected from the archive—either the highest-fitness policy (exploitation) or a random member (exploration), with equal probability. The full source code of the parent, its fitness scores, and the source of other high-performing policies are provided to GPT-5.3-codex, which returns a complete modified Python function.

Before evaluation, each candidate undergoes a two-call sanity check: it is executed at `rtt_ratio` = 1.5 (moderate congestion) and `rtt_ratio` = 1.05 (near-idle). If the policy fails to return a valid action or never increases its window under low congestion, it is rejected as degenerate. Candidates that pass are evaluated in the Python simulator on the training condition (15 Mbps, 150 ms RTT, 2 senders). A candidate is added to the archive if its normalized score S improves upon its parent's. The full process comprises 75 candidate evaluations (15 generations × 5 candidates).

3.3.2 Multipoint training. In the baseline configuration, each candidate receives a single fitness score derived from one network condition. This provides the LLM with a scalar signal but no information about how the policy behaves across different link rates. In preliminary experiments, this single-point setup produced poor policies. To provide a richer feedback signal, we evaluate a multipoint configuration in which each candidate is tested across six link rates spanning the Remy 10× training range. The resulting vector of per-rate scores is included in the mutation prompt, giving the LLM a signal about which conditions cause degradation. We study this multipoint configuration alongside a diverse set of initial seeds in §4.3.1.

4 RESULTS

4.1 Main Comparison: All Methods Across Link Rates

Figure 1 shows normalized scores for CCAs produced by all three learning schemes, evaluated in Remy's C++ simulator across 99 log-spaced link rates from 1–1000 Mbps. Mirroring the pattern seen in [12], we see that the Remy CCA trained on the 4.7-47.4 mbps range of link rates (10x) performs better across this range than the Remy CCA trained at the single link rate of 15 mbps (1x), but the 1x Remy CCA performs better at its single training link rate of 15 mbps.

The PPO and Remy 1x CCA achieve comparable performance at the training link rate, with Remy 1x performing slightly better (achieving the highest performance of all CCAs). Interestingly, the PPO CCA seems to generalize slightly better near the training link rate than the Remy CCA.

While AlphaCC underperforms the other CCAs at the training link rate, it generalizes exceptionally well to higher link rates. It also shows spiky performance at lower link rates in a way that is unique among all CCAs.

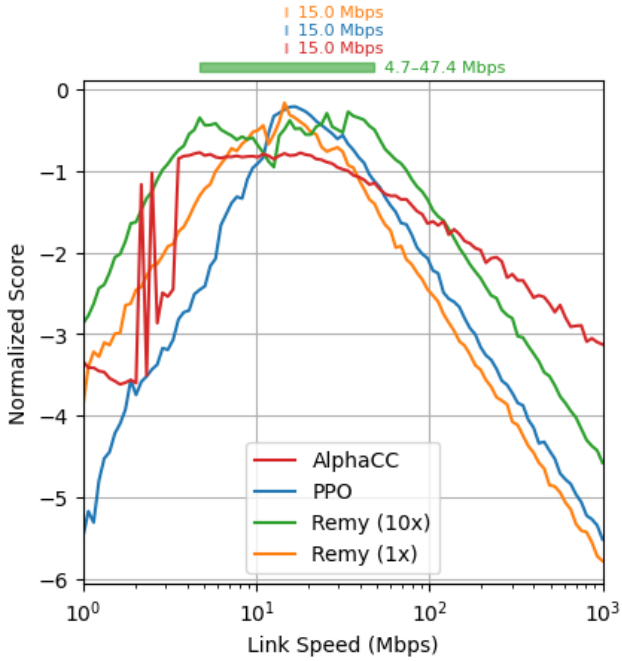

Figure 1: CCAs from all three CCA learning schemes evaluated in Remy's C++ simulator. Link ranges used in training are specified by horizontal bars (or vertical lines for single rate training) above.

4.2 PPO-Specific Analysis

4.2.1 PPO training progress plateaus. In our PPO learning scheme, we observed an initial slow rate of improvement, followed by a rapid rate of improvement, followed by a plateauing of performance (Figure 2). The fact that the final PPO-generated CCA is quite similar in performance to the final Remy-generated CCA indicates that this performance level may be a more fundamental limit that is general across all CCA classes. It may also be possible that the plateauing we see for our PPO agent is actually just a slow rate of increase that allows more improvement in performance given a longer training budget. We suspect that a lot of optimization could be done to improve the efficiency of our PPO learning scheme, which could enable this type of analysis in the future.

4.2.2 PPO policy sampling temperature affects generalization in an unexpected way. We decided to take our PPO agent trained with a sampling temperature of one and evaluate it with varying temperatures, with the hypothesis that a higher temperature—and thus an action distribution with higher

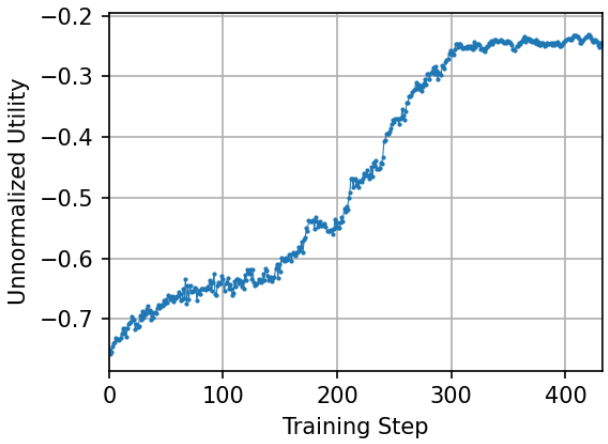

Figure 2: PPO unnormalized utility over the course of training (15 Mbps).

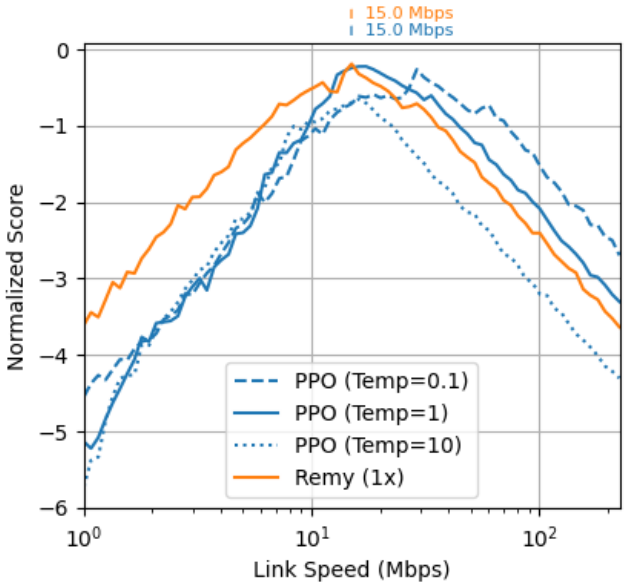

Figure 3: PPO CCA performance across link rates vs sampling temperature. Higher temperature means higher entropy of action distribution.

entropy—may yield better generalization, because, in a sense, this makes the policy less committal. In addition to evaluating our policy on a temperature greater than one (10), we also evaluate with a temperature of 0.1, yielding a lower entropy policy than the one we trained. We find that the higher entropy policy performs worse over all link rates. However, surprisingly, we also find that the lower entropy policy, while performing worse at the training link rate, performs better at higher link rates than the base policy.

4.3 AlphaCC-Specific Analysis

4.3.1 Single-Point Failure and Multipoint Training. Under single-point training (one fitness scalar from 15 Mbps), the LLM produced no improvement over any seed baseline across 15 generations. A single scalar provides no gradient—the LLM cannot determine *which rates* degrade or *why*.

Switching to multipoint training across 6 link rates (matching Remy 10×'s training range) provides the LLM with a vector of per-rate scores in the mutation prompt. This richer feedback enabled improvement: every seed family improved within 1–3 generations.

To test robustness to initialization, we ran 8 independent evolutions (2 per family) from four structurally different seeds:

- **Minimal**: no congestion-control logic.
- **AIMD**: loss-based; multiplicative decrease on RTT-ratio threshold.
- **Copa**: delay-based; adjusts window toward $1/(\delta \cdot q_{\text{delay}})$.
- **BBR**: rate-based; sets intersend from receive-rate EWMA.

```
# From constant_r1 (started from scratch):
bdp_pkts = max(1.0, min_rtt / rec)
scale = max(1.0, min(12.0, math.sqrt(bdp_pkts)))
intersend = 1.0 / max(1e-6, target_rate * pace_gain)
inc = int(max(1.0, round(0.55 * scale + ...)))

# From aimd_r1 (started from AIMD seed):
bdp_est = max(0.2, min(600.0, min_rtt / rec))
bdp_scale = max(1.0, min(18.0, math.sqrt(bdp_est)))
intersend = rec / boost
inc = int(max(1.0, min(36.0, 1.2 + 0.95*bdp_scale)))

# From pacing_r1 (started from pacing seed):
bdp_pkts = max(1.0, min_rtt / rec)
bdp_scale = max(0.7, min(6.0, math.sqrt(bdp_pkts)))
intersend = 1.0 / max(1e-6, target_rate)
inc = max(1, int(1 + 1.4 * bdp_scale))
```

Figure 4: Core mechanism excerpted from evolved Python policies (one per seed family). All compute BDP from `min_rtt/rec_ewma`, scale by $\sqrt{\text{BDP}}$, and pace from receive rate.

Key findings. No evolved policy beats Remy 1x on the 50-rate mean (best evolved: −0.44 vs. Remy −0.63). However, every seed family improves substantially over its baseline, with the largest gain from the Minimal seed (+2.22 fitness, from zero CC logic to BDP-scaled pacing in one generation). The Minimal-seeded policy achieves the flattest generalization

profile, staying between −0.22 and −0.48 from 39–1000 Mbps while Remy degrades to −1.88.

4.3.2 Mechanistic Convergence. We inspected each of the 8 final evolved policies and classified the presence of four structural elements. This categorization was performed by one author post hoc; labels were not used during evolution.

All 8 policies, regardless of seed family, converge on the same architecture:

(1) **BDP proxy** (8/8): `bdp = min_rtt / rec_ewma`, approximating in-flight packets at the bottleneck rate.
(2) **In-flight scaling** (8/8): window increments proportional to $\sqrt{\text{bdp}}$.
(3) **Rate-based pacing** (8/8): `intersend` set from receive-rate EWMA, not purely window-driven.
(4) **Queue pressure sensing** (8/8): RTT-ratio thresholds (~1.3 and ~2.0) gate transitions between growth, maintenance, and backoff.

The four seed families encode structurally different CCA philosophies: the minimal seed provides *no* congestion-control logic, AIMD is loss-based, Copa is delay-based, and BBR is rate-based. All four converge on in-flight-scaled pacing—including the minimal seed, which starts from zero. We cannot distinguish whether this convergence reflects a genuine fitness-landscape attractor or the LLM's inability to explore beyond familiar TCP templates.

Appendix A, Figure 4 shows core code from three policies evolved from different seeds. Despite different variable names and thresholds, the structural pattern is identical: estimate BDP from `min_rtt / rec_ewma`, scale window changes by $\sqrt{\text{BDP}}$, and pace at the estimated receive rate.

4.3.3 Evolution Trajectory. Figure 5 shows the running-best score per seed family across 8 runs (4 families × 2 repetitions). Most improvement occurs in the first 3 generations, with diminishing returns afterward. BBR-seeded runs converge fastest, approaching Remy 1x by generation 1.

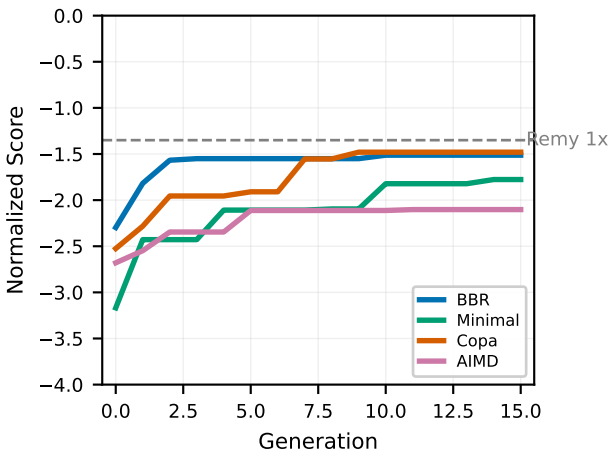

Figure 5: Evolution trajectory by seed family (mean of 2 runs each). Dashed line: Remy 1x (−1.35). BBR seeds converge fastest; Minimal seeds start lowest but improve steadily.

5 CONCLUSION

We compared two additional CCA learning schemes under Remy's training and evaluation setup—a neural network learned with PPO and an LLM-guided code evolution system—alongside Remy's decision trees. In general, we find that no method achieves both superior in-distribution and out-of-distribution performance. Moreover, instead of finding that different learned CCAs consistently generalize better or worse, we instead find that generalization depends on direction—some CCAs perform better for higher link rates than what they saw during training, while others perform better for lower link rates than what they saw during training.

The lack of generalization across all CCA learning schemes may indicate that something fundamental is wrong with learning CCAs by maximizing final utilities measured offline. Perhaps changing the problem framing or information available to the policies is necessary to yield both performant and robust CCAs. For example, maybe allowing policies to learn online from experience gathered real time and from the network it currently occupies is necessary for good generalization.

While the publication of original source code from the Remy work has enabled controlled comparisons between our learning schemes and Remy, there remain several limitations in this analysis. For one, we faced compute bottlenecks in generating the Remy CCAs, and the lack of optimization time of these Remy CCAs may have yielded differences between our Remy CCAs and those presented in the original Remy papers [11, 12]. Additionally, for both Remy and PPO, we only train a single policy for each training condition, and a more robust analysis would be enabled by training from scratch from several initial random seeds.

REFERENCES

[1] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *Proc. USENIX NSDI*. 329–342.
[2] Venkat Arun and Hari Balakrishnan. 2021. Automated Reasoning about Congestion Control. In *Proc. ACM SIGCOMM*.
[3] Venkat Arun and Hari Balakrishnan. 2024. Automated Synthesis of Congestion Control Algorithms. In *Proc. USENIX NSDI*.
[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (2017), 20–53.

[5] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proc. USENIX NSDI*. 395–408.

[6] Dan Hendrycks and Kevin Gimpel. 2023. Gaussian Error Linear Units (GELUs). (2023). arXiv:cs.LG/1606.08415 https://arxiv.org/abs/1606.08415

[7] Nathan Jay, Noga Rotman, P. Brighten Godfrey, Michael Schapira, and Aurojit Panda. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proc. ICML*. 3150–3159.

[8] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. (2019). arXiv:cs.LG/1711.05101 https://arxiv.org/abs/1711.05101

[9] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borja Ibarz, Kieran Milan, Daniel Zheng, Manzil Zaheer, Yannis Assael, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Alhussein Fawzi. 2025. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. (2025). arXiv:cs.AI/2506.01942 https://arxiv.org/abs/2506.01942

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. (2017). arXiv:cs.LG/1707.06347 https://arxiv.org/abs/1707.06347

[11] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. 2014. An Experimental Study of the Learnability of Congestion Control. In *Proc. ACM SIGCOMM*. 479–490.

[12] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. In *Proc. ACM SIGCOMM*. 123–134.

[13] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the Training Ground for Internet Congestion-Control Research. In *Proc. USENIX ATC*. 731–743.

A SUPPLEMENTARY MATERIAL

Table 3: Statefulness ablation. AlphaCC: best 3 of 5 exploratory policies. Copa: mean ± std over 5 seeds. Stateless: single evolution run.

Condition	Stateful	Stateless	Copa
Near (9.5 Mbps)	**−0.72**±0.06	−2.54	−3.06±0.12
Low (2.4 Mbps)	−1.57±0.51	**−0.75**	−1.36±0.10
High (59.8 Mbps)	**−0.78**±0.07	−1.92	−5.70±0.12

B AI USE STATEMENT

Along with the use of AI within the AlphaCC algorithm itself, we also used AI tools to assist with development and implementation. We used AI tools for assistance with converting from Python to C++ in order to evaluate. However, we did not depend on any AI tooling as an authoritative source of factual or scientific truth.

We also used AI to assist with build errors due to outdated dependencies in the old Remy repository. For the PPO agent specifically, we set up a plan for refactoring the Remy codebase to enable the use of alternative CCA learning schemes, and Claude wrote a lot of the implementation for this refactoring given this plan. Claude wrote the initial prototype of the PPO algorithm, but the current version looks quite different from this initial prototype as we have manually adjusted it iteratively based on new experimental results.