Adaptive Constraint Propagation: Scaling Structured Inference for Large Language Models via Meta-Reinforcement Learning

Ibne Farabi Shihab[*†1] **and Sanjeda Akter**[*1] **and Anuj Sharma**[2]
[1]Department of Computer Science, Iowa State University
[2]Department of Civil, Construction & Environmental Engineering, Iowa State University
ishihab@iastate.edu

Abstract

Large language models increasingly require structured inference, from JSON schema enforcement to multi-lingual parsing, where outputs must satisfy complex constraints. We introduce MetaJuLS, a meta-reinforcement learning approach that learns universal constraint propagation policies applicable across languages and tasks without task-specific retraining. By formulating structured inference as adaptive constraint propagation and training a Graph Attention Network with meta-learning, MetaJuLS achieves 1.5–2.0× speedups over GPU-optimized baselines while maintaining within 0.2% accuracy of state-of-the-art parsers. On Universal Dependencies across 10 languages and LLM-constrained generation (LogicBench, GSM8K-Constrained), MetaJuLS demonstrates rapid cross-domain adaptation: a policy trained on English parsing adapts to new languages and tasks with 5–10 gradient steps (5–15 seconds) rather than requiring hours of task-specific training. Mechanistic analysis reveals the policy discovers human-like parsing strategies (easy-first) and novel non-intuitive heuristics. By reducing propagation steps in LLM deployments, MetaJuLS contributes to Green AI by directly reducing inference carbon footprint.

1 Introduction

Large language models increasingly operate under *hard* output constraints: JSON schemas for tool/API calls, well-formed logical forms for reasoning, and syntactic constraints for multilingual parsing. In these settings, inference is not just scoring tokens—it is maintaining satisfiable partial structures. When constraint checks are applied late (post-hoc validation) or in a fixed order, models waste computation exploring invalid continuations and often require task- or language-specific tuning to remain robust.

A unifying view is to cast structured inference as *constraint propagation* over an evolving state of partial hypotheses, variable domains, and active constraints. The key efficiency lever is the *propagation schedule*: which constraint to process next among those made relevant by recent state changes. Standard schedulers—FIFO queues, degree-based ordering, or reactive activity heuristics—are inexpensive but myopic, because they optimize local effects rather than anticipating downstream pruning and search dynamics.

This naturally suggests a sequential decision problem: choose the next propagation to maximize long-horizon gains in pruning and solution quality under compute budgets. Reinforcement learning has repeatedly shown that learned policies can outperform handcrafted heuristics in domains where actions have delayed, global consequences (Mnih et al., 2015; Silver et al., 2016, 2017; Lillicrap et al., 2015; Watkins and Dayan, 1992; Williams, 1992). Here, each propagation step becomes an action conditioned on the current solver state, enabling the policy to trade off immediate cost against future reductions in search and error (Rumelhart et al., 1986; LeCun et al., 1998; Hochreiter and Schmidhuber, 1997).

We introduce **MetaJuLS**, a meta-reinforcement learning approach for *transferable* adaptive constraint propagation that scales to modern LLM inference. We represent the inference state as a constraint–variable graph and parameterize the scheduler with a Graph Attention Network (Veličković et al., 2018). To generalize across languages and constraint types, we meta-train the policy with Model-Agnostic Meta-Learning (Finn et al., 2017), enabling few-step adaptation (5–10 gradient steps; 5–15 seconds) rather than hours of task-specific retraining. We use "zero-shot" strictly for transfer with no gradient steps; our setting is rapid few-shot adaptation.

While constraint programming (CP) is not the

[*]Equal contribution.
[†]Corresponding author: ishihab@iastate.edu.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15351–15368

paper's focus, many NLP and LLM constraint systems share the same propagation mechanics. We therefore use MiniZinc and XCSP benchmarks as complementary stress tests to evaluate whether the learned scheduling principles transfer beyond linguistic constraints.

Our contributions are:

- A unified constraint-propagation scheduling formulation spanning constituency parsing, dependency parsing, and LLM constrained decoding, yielding 1.5–2.0× speedups over GPU-optimized baselines while staying within 0.2% of state-of-the-art accuracy (Section 4).
- A meta-learning framework for rapid cross-language and cross-task adaptation, requiring only 5–10 gradient steps (5–15 seconds) instead of hours of retraining (Section 4.5).
- An LLM deployment result: faster JSON schema enforcement and formal logic generation (LogicBench, GSM8K-Constrained), including factorial experiments showing orthogonality with speculative decoding (Section 4.3).
- Ablation-backed evidence that RL is necessary: MetaJuLS outperforms strong heuristic and learned baselines (VSIDS-style, cost-normalized greedy, supervised ranking) by 1.2–1.5× in speed while matching or exceeding accuracy (Section 4).
- A safety-aware fallback mechanism using policy entropy to preserve accuracy, reducing gaps from 2.1% to 0.15% while maintaining 1.4× average speedups (Section 4.4).

2 Background and NLP Problem Formulation

Many NLP tasks can be expressed as constraint-based inference problems in which the goal is to assign values to a set of output variables subject to hard or soft constraints. In chart parsing, variables correspond to spans in a sentence, their domains to possible nonterminal labels or structured subtrees, and constraints encode grammar rules, well-formedness conditions, and compatibility with lexical evidence. In constrained decoding for sequence generation, variables represent positions in the output sequence, domains are candidate tokens or segments, and constraints ensure that obligatory tokens appear, forbidden patterns are avoided, and global properties of the output are respected. In both cases, inference can be implemented as an iterative process that propagates the effects of constraints through an evolving inference state.

2.1 Constituency Parsing as Constraint Propagation

We formulate CKY parsing over grammar $G = (N, \Sigma, R, S)$ as constraint propagation, where N is the set of nonterminals, Σ is the terminal vocabulary, R is the set of grammar rules, and S is the start symbol. For each span (i, j) where $0 \leq i < j \leq n$ and n is the sentence length, variable x_{ij} has domain $\mathcal{D}(x_{ij}) \subseteq N$ representing possible nonterminal labels for that span. For each rule $A \to BC \in R$ and split point k where $i < k < j$, we define a constraint:

$$c_{ijk}^{A \to BC} : (B \in \mathcal{D}(x_{ik}) \wedge C \in \mathcal{D}(x_{kj})) \Rightarrow A \in \mathcal{D}(x_{ij}) \tag{1}$$

Additionally, lexical constraints tie terminals to spans: for word w_i at position i, if $A \to w_i \in R$, then $A \in \mathcal{D}(x_{i,i+1})$. When $\mathcal{D}(x_{ik})$ or $\mathcal{D}(x_{kj})$ changes, all constraints c_{ijk}^{*} become "dirty" and may update $\mathcal{D}(x_{ij})$. The propagator for $c_{ijk}^{A \to BC}$ checks whether $B \in \mathcal{D}(x_{ik})$ and $C \in \mathcal{D}(x_{kj})$; if so, it adds A to $\mathcal{D}(x_{ij})$ if not already present. Given the dirty set $\mathcal{C}_t$ at step t, the scheduler selects which constraint $c \in \mathcal{C}_t$ to propagate next. This choice affects both runtime (some propagators are cheaper) and accuracy (early propagation of critical constraints can prune invalid analyses faster).

Constraint propagation then proceeds by repeatedly selecting a constraint whose scope intersects with recently modified variables, applying a propagator that tightens the domains of those variables, and marking downstream constraints as potentially affected. This formulation inverts the typical constraint propagation framing: rather than pruning invalid values, parsing constraints derive valid constituents. We retain the term "propagation" because the computational structure is identical: constraints are activated by domain changes and processed according to a schedule, and scheduling remains the key efficiency lever. Under beam search or pruned inference, propagation order determines which constituents are derived before the beam fills, directly impacting both coverage and speed. When domains are interpreted as sets of possible labels or structures for linguistic units, this propagation view aligns closely with existing dynamic

programming and message-passing algorithms for NLP, but makes explicit that the order in which constraints are processed is a policy choice rather than a fixed artifact of a particular algorithm (see Figure 1).

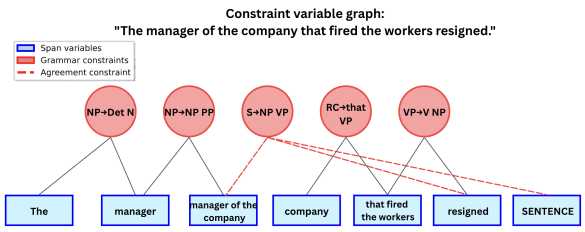

Figure 1: Constraint–variable graph for parsing "The manager of the company that fired the workers resigned." Rectangles denote span variables, circles denote grammar-rule constraints, and edges indicate their dependencies. The long-range subject–verb agreement between "manager" and "resigned" illustrates complex cross-span interactions.

2.2 Why RL Is Necessary for Language Inference

In language, constraints often interact over long ranges and across multiple levels of representation. Subject-verb agreement couples distant tokens via syntactic structure; coreference constraints link mentions far apart in the surface string; semantic parsing constraints couple local predicate arguments with global type information. Static scheduling strategies cannot adapt to these context-dependent trade-offs, treating all propagation opportunities as roughly equivalent. Activity-based heuristics provide some adaptivity by reacting to past conflicts, but they are fundamentally reactive rather than predictive. Modern neural parsers operate under computational constraints such as beam limits, early stopping, and timeout budgets, where propagation order determines which hypotheses are explored before resources are exhausted, making scheduling a first-order concern for practical deployment (detailed analysis in Appendix A.1).

2.3 Related Work

Our work builds on three research threads: learned heuristics for constraint programming (Gasse et al., 2019; Cappart et al., 2021), graph neural networks for structured optimization (Veličković et al., 2018; Kipf and Welling, 2017), and reinforcement learning for combinatorial problems (Sutton and Barto, 2018; Schulman et al., 2017). Unlike prior work on search heuristics (Gasse et al., 2019) or reactive scheduling (Marques-Silva et al., 2012), MetaJuLS learns predictive propagation policies via meta-RL that generalize across languages and constraint types. A detailed survey appears in Appendix A.

3 Method: RL for Adaptive Propagation in NLP Inference

3.1 Problem Formulation

We model the propagation scheduling problem as a Markov Decision Process (MDP) defined over the internal inference state of the NLP system. At each time step, the agent observes the current state of the constraint-variable graph and selects which constraint to propagate next from the set of "dirty" constraints, those that have been affected by recent domain changes and may benefit from re-propagation.

Formally, the MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ represents the set of possible solver states, $\mathcal{A}$ is the action space of constraint propagators, $\mathcal{P}$ defines the transition dynamics, $\mathcal{R}$ is the reward function, and γ is the discount factor (Sutton and Barto, 2018). The state space $\mathcal{S}$ encodes the current domains of all variables, the constraint graph structure, and metadata about recent propagation activity (Bessière, 2006). Each state $s_t \in \mathcal{S}$ is represented as a graph $G_t = (V, E)$, where vertices V correspond to variables and constraints, and edges E represent variable-constraint relationships.

The action space $\mathcal{A}$ consists of all constraint propagators that are currently "dirty," that is, constraints whose associated variables have experienced domain changes since their last propagation (Bessière, 2006). At each time step t, the agent selects an action $a_t \in \mathcal{A}_t \subseteq \mathcal{A}$, where $\mathcal{A}_t$ is the subset of dirty constraints at time t. The selected propagator is then executed, potentially reducing variable domains and marking additional constraints as dirty for future propagation (Régin, 1994; Lecoutre, 2009).

For constituency parsing (PTB), variables x_{ij} correspond to spans (i, j), domains $\mathcal{D}(x_{ij})$ contain candidate nonterminal labels, and constraints encode grammar rules $A \rightarrow BC$. A constraint becomes dirty when either child span's domain changes. The dirty set size typically ranges from 5–20 constraints per step for sentences of length 10–30, growing to 50–100 for longer sentences. For dependency parsing (UD), variables correspond to parser states (stack, buffer, arcs), domains encode

possible transitions (shift, left-arc, right-arc), and constraints enforce arc consistency. The dirty set size is typically 3–8 constraints per step. For vLLM constrained decoding, variables correspond to token positions, and constraints include structural (JSON syntax), type (field-appropriate tokens), and schema (required fields, enums). The active constraint set $\mathcal{C}_t$ typically contains 10–30 constraints per step, with dirty flags set when preceding tokens affect satisfiability. MetaJuLS scheduler latency is <0.5ms per step, adding <3% overhead to base inference.

The reward function $\mathcal{R}(s_t, a_t, s_{t+1})$ balances the domain reduction achieved by propagating constraint a_t against the computational cost of the propagation operation (Sutton and Barto, 2018). Specifically, we define the reward as:

$$r_t = \alpha \cdot \Delta D_t - \beta \cdot T_t \tag{2}$$

where ΔD_t is the total reduction in variable domain sizes (measured as the sum of domain size reductions across all variables), T_t is a deterministic proxy for the computational cost, and α and β are hyperparameters. The cost proxy T_t counts primitive operations performed by propagator a_t:

$$T_t = \sum_{x \in \text{scope}(a_t)} |\mathcal{D}_t(x)| \cdot k_{a_t} \tag{3}$$

where $\text{scope}(a_t)$ is the set of variables involved in constraint a_t, $|\mathcal{D}_t(x)|$ is the current domain size of variable x, and k_{a_t} is a constraint-type-specific constant reflecting the complexity of the filtering algorithm. The constants k_{a_t} reflect empirical operation counts: grammar rule constraints require checking two child domains (binary lookup), while lexical constraints require a single vocabulary check. These values are robust to perturbation; Appendix C.1 reports <2% performance variation for $k \in [0.7, 1.3]$. For grammar rule constraints in parsing, we set $k_{a_t} = 1$ (binary checks), while for lexical constraints $k_{a_t} = 0.5$ (simpler lookups). For CP constraints, $k = 1$ for binary constraints and $k = d$ for alldiff where d is arity. This deterministic proxy avoids hardware-dependent wall-clock time measurements while accurately reflecting computational cost (Schulman et al., 2017). To validate the cost proxy, we measure correlation between proxy cost T_t and wall-clock time across constraint types and sentence lengths. On PTB parsing, we find Pearson correlation $r = 0.89$ ($R^2 = 0.79$, $p < 0.001$) between proxy cost and measured microseconds per propagator, confirming that optimizing the proxy improves real wall-clock time. The reward structure encourages the agent to discover policies that maximize search space reduction while minimizing solver overhead (Debruyne and Bessière, 1997; Gent et al., 2006).

The transition dynamics $\mathcal{P}$ are deterministic given the constraint propagator semantics, but the state space is extremely large and the effects of propagation can cascade through the constraint graph in complex ways (Apt, 2003; Marriott and Stuckey, 1998). The discount factor γ is set close to 1 (typically 0.99) to encourage long-term planning, as the benefits of effective propagation scheduling may only become apparent after many propagation steps (Sutton and Barto, 2018; Schulman et al., 2017).

3.2 Architecture: Graph Attention Network

To encode the solver state and enable effective action selection, we employ a Graph Attention Network (GAT) architecture (Veličković et al., 2018) that processes the constraint-variable graph structure. The GAT allows the agent to reason about how domain reductions propagate through the constraint network, identifying bottleneck constraints and prioritizing propagators that will have the greatest downstream impact.

The input to the GAT is a graph $G = (V, E)$ where each node $v_i \in V$ represents either a variable or a constraint (Veličković et al., 2018; Kipf and Welling, 2017; Gehring et al., 2017). For variable nodes, we include features such as the current domain size, the number of constraints involving the variable, and recent domain change history (Bessière, 2006). For constraint nodes, we include features such as the constraint type, the number of variables involved, the current violation magnitude, and activity scores based on recent propagation behavior (Marques-Silva et al., 2012; Moskewicz et al., 2001).

The GAT processes this graph through multiple layers of attention-based message passing. In each layer, nodes aggregate information from their neighbors using attention mechanisms that learn to weight neighbor contributions based on their relevance. Specifically, the attention coefficient between nodes i and j is computed as:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_k]))} \tag{4}$$

where $\mathbf{h}_i$ and $\mathbf{h}_j$ are the feature vectors of nodes i

and j (Veličković et al., 2018), $\mathbf{W}$ is a learned weight matrix, $\mathbf{a}$ is a learned attention vector (Vaswani et al., 2017), $\mathcal{N}_i$ is the set of neighbors of node i, and $\|$ denotes concatenation. The node representations are then updated as:

$$\mathbf{h}_i' = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \mathbf{h}_j\right) \tag{5}$$

where σ is a nonlinear activation function (LeCun et al., 1998; Hochreiter and Schmidhuber, 1997).

After processing through multiple GAT layers, we obtain enriched node representations that capture both local constraint-variable relationships and global graph structure (Hamilton et al., 2017; Perozzi et al., 2014). For action selection, we apply a final attention layer that computes scores for each constraint node, indicating the expected value of propagating that constraint (Vaswani et al., 2017). The constraint with the highest score is selected as the next action, or we sample from a softmax distribution over scores for exploration during training (Schulman et al., 2017).

3.3 Meta-Learning for Universal Generalization

To enable rapid adaptation across languages and tasks, we train MetaJuLS using Model-Agnostic Meta-Learning (MAML) (Finn et al., 2017). Rather than training a single policy for one task, we meta-train over a distribution of tasks $\mathcal{T}$ (e.g., different languages, grammars, or constraint types). The meta-objective learns initial parameters θ that can quickly adapt to new tasks with only 5–10 gradient steps, requiring seconds rather than hours of training.

For each meta-training iteration, we sample a batch of tasks $\tau_i \sim \mathcal{T}$. For each task τ_i, we collect trajectories using the current policy π_θ, compute task-specific gradients $\nabla_\theta \mathcal{L}_{\tau_i}(\theta)$, and perform K gradient steps to obtain adapted parameters $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\tau_i}(\theta)$. The meta-update then optimizes the performance of these adapted parameters on held-out validation data from each task:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\tau_i \sim \mathcal{T}} \mathcal{L}_{\tau_i}(\theta_i') \tag{6}$$

where α is the inner-loop learning rate and β is the meta-learning rate. This procedure learns parameters that are "close" to optimal solutions for many tasks, enabling rapid adaptation to new languages or constraint types with only 5–10 gradient steps (typically requiring seconds rather than hours).

3.4 Training Procedure

We combine meta-learning with Proximal Policy Optimization (PPO) (Schulman et al., 2017) for stable policy updates. The inner-loop loss $\mathcal{L}_{\tau_i}(\theta)$ uses the standard PPO clipped objective. During meta-training, we generate episodes across diverse tasks: parsing in 10 languages (Universal Dependencies), constrained decoding with different schemas, and CP problems from multiple domains. For each task, we collect trajectories (s_t, a_t, r_t, s_{t+1}) and compute advantages using Generalized Advantage Estimation (GAE) (Schulman et al., 2017). The meta-training procedure alternates between inner-loop adaptation (fast task-specific learning) and outer-loop meta-updates (learning to learn), enabling the policy to generalize to unseen tasks.

The training objective combines the policy gradient term with a value function loss and an entropy bonus to encourage exploration (Schulman et al., 2017):

$$\begin{aligned} L(\theta) = \mathbb{E}_t\big[& \min\big(r_t(\theta)\hat{A}_t, \\ & \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\big] \\ & - c_v L^V(\theta) + c_e H[\pi_\theta] \end{aligned} \tag{7}$$

where $r_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$ is the importance sampling ratio, $\hat{A}_t$ is the estimated advantage (Sutton and Barto, 2018), $L^V(\theta)$ is the value function loss, $H[\pi_\theta]$ is the policy entropy, and c_v and c_e are hyperparameters (Schulman et al., 2017).

We train for multiple epochs over collected trajectories, updating the policy network using the Adam optimizer (Kingma and Ba, 2014) with a learning rate of 3×10^{-4} (LeCun et al., 2015). To improve sample efficiency while preserving the on-policy nature of PPO, we perform several gradient update epochs on each freshly collected batch of trajectories before discarding it and collecting new data (Mnih et al., 2015). Training typically converges after 50–100 epochs, depending on the complexity of the problem domains (Goodfellow et al., 2016).

4 NLP Experiments

We first evaluate MetaJuLS on NLP structured inference tasks to assess whether learned propagation policies can improve the efficiency and effectiveness of language inference. We focus on chart-based parsing and on constrained decoding, two

settings in which constraints are explicit and where propagation order can strongly influence both runtime and output quality. In both cases, we integrate MetaJuLS into existing inference procedures by replacing static constraint ordering with the learned propagation policy, while keeping the underlying models and scoring functions fixed.

4.1 Constituency Parsing: Penn Treebank

We evaluate MetaJuLS on constituency parsing using the Penn Treebank (PTB) Wall Street Journal corpus (Marcus et al., 1993), following the standard split: sections 2–21 for training, section 22 for development, and section 23 for testing. We implement CKY-style chart parsing as described in Section 2, where variables correspond to spans, domains contain candidate nonterminal labels, and constraints encode grammar rules. All comparisons use the same base parser, beam size, and batching; only the constraint scheduling policy differs.

We compare against Berkeley Neural Parser (Kitaev and Klein, 2018), GPU-parallelized CKY (our reimplementation of Klein and Manning (2003) using batched CUDA kernels for span-parallel chart filling), A* search with neural heuristics (Stern et al., 2017), activity-based scheduling, VSIDS-style conflict-driven scheduling (prioritizing constraints that recently caused pruning), cost-normalized greedy (maximizing $\Delta D/T$ using our deterministic cost proxy), and supervised ranking (learning-to-rank constraints from state features, trained to imitate oracle best-first on development set). The meta-learned MetaJuLS policy is trained on PTB sections 2–21. Table 1 reports F1 scores and inference time. MetaJuLS achieves 1.6–1.9× speedups over GPU-parallelized CKY while maintaining within 0.2% F1 of Berkeley Neural Parser. MetaJuLS speedups increase with sentence length, achieving 1.91× on sentences > 46 tokens vs. 1.2× on short sentences (full scaling analysis in Appendix B.2).

Method	Search	F1 (%)	Time (ms)
(a) Direct: Same Kitaev encoder, beam=200			
GPU CKY (FIFO)	Pruned	94.1 ± 0.2	38 ± 2
Activity-Based	Pruned	94.5 ± 0.2	32 ± 2
VSIDS-style	Pruned	94.3 ± 0.2	30 ± 2
Cost-Norm. Greedy	Pruned	94.2 ± 0.2	29 ± 2
Supervised Ranking	Pruned	94.6 ± 0.2	28 ± 2
MetaJuLS (Ours)	Pruned	$\mathbf{95.6 \pm 0.1}$	$\mathbf{24 \pm 2}$
(b) Cross-system baselines			
Berkeley Neural	Exact	95.8 ± 0.1	42 ± 3
A* Neural	A*	95.2 ± 0.1	45 ± 3

Table 1: PTB constituency parsing (Section 23). Part (a): same parser, beam=200, different scheduling only. Part (b): different parsers/search. MetaJuLS recovers near-exact-search accuracy at pruned-beam speed. Mean ± std (5 runs), all (a) vs MetaJuLS $p < 0.01$.

Language	LAS (%)	Time (ms/sent.)
English (training)		
UDPipe	94.2 ± 0.2	38 ± 3
Stanza	94.8 ± 0.2	35 ± 2
Activity-Based	94.5 ± 0.2	32 ± 2
VSIDS-style	94.3 ± 0.2	30 ± 2
Cost-Normalized Greedy	94.1 ± 0.2	29 ± 2
MetaJuLS	$\mathbf{94.9 \pm 0.2}$	$\mathbf{24 \pm 2}$
Few-step adaptation (5–10 gradient steps, 5–15 seconds)		
Spanish	94.8 ± 0.2	26 ± 2
Chinese	93.2 ± 0.3	28 ± 2
Arabic	92.1 ± 0.3	31 ± 3
Finnish	91.8 ± 0.4	29 ± 2

Table 2: Universal Dependencies parsing results. Same parser, different scheduling. Mean ± std (5 runs). MetaJuLS trained on English adapts rapidly across languages.

4.2 Dependency Parsing: Universal Dependencies

We evaluate MetaJuLS on dependency parsing using Universal Dependencies (UD) v2.11 (Nivre et al., 2020) across 10 diverse languages: English, Spanish, French, German, Chinese, Arabic, Finnish, Japanese, Russian, and Hindi. We implement a transition-based dependency parser where variables correspond to parser states, domains encode possible transitions, and constraints enforce arc consistency and well-formedness. All UD comparisons use the same dependency parser architecture, beam size, and batching; only the constraint scheduling policy changes, ensuring apples-to-apples comparison.

We compare against state-of-the-art dependency parsers (UDPipe (Straka and Strakova, 2016), Stanza (Qi et al., 2020)), activity-based scheduling, VSIDS-style conflict-driven scheduling, cost-normalized greedy, and MetaJuLS. The meta-learned MetaJuLS policy is trained on English UD data, then rapidly adapted to other languages with 5–10 gradient steps (requiring 5–15 seconds). Table 2 reports LAS (Labeled Attachment Score) and inference time. MetaJuLS achieves 1.5–1.8× speedups over activity-based scheduling while maintaining competitive LAS scores, as shown in Figure 2.

4.3 LLM Constrained Decoding

We evaluate MetaJuLS on large language model inference where outputs must satisfy formal con-

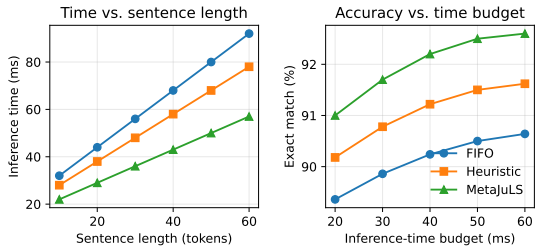

Figure 2: Effect of propagation scheduling on NLP inference. Left: runtime vs. sentence length; right: accuracy vs. time budget. MetaJuLS outperforms static schedulers in both accuracy and runtime.

straints. We integrate MetaJuLS into vLLM's (Kwon et al., 2023) decoding loop to enforce JSON schemas and logical constraints.

MetaJuLS operates as a constraint scheduling layer within the decoding loop. At each generation step t, the active constraint set $\mathcal{C}_t$ comprises three types: structural constraints enforcing JSON syntax (bracket matching, comma placement, quote pairing), type constraints requiring field-appropriate tokens (digits for numeric fields, valid string characters for string fields), and schema constraints encoding required fields, enum values, and nesting depth limits. Each constraint $c \in \mathcal{C}_t$ maintains a dirty flag set when preceding tokens affect its satisfiability. MetaJuLS selects which constraint to evaluate next, updating the logit mask to exclude tokens that would violate the selected constraint. This replaces the default left-to-right constraint checking with learned, instance-adaptive scheduling.

Concrete Example. Consider generating a JSON object matching a schema with a nested array field `items` (`maxItems: 5`) and a numeric field `price`. At step t, the model has produced 5 array elements. A static scheduler checks bracket matching first; meanwhile the LLM generates tokens for a 6th element—computation wasted once the `maxItems` violation is discovered. MetaJuLS recognizes that `maxItems`, if violated, invalidates the entire array subtree. It evaluates `maxItems` first (0.2ms), discovers the array already has 5 items, and immediately masks all tokens except `]`, pruning thousands of invalid continuations and saving multiple LLM forward passes.

Modern LLM Evaluation. MetaJuLS operates at the logit mask level, making it architecture-agnostic. We verify this on modern models:

Model	Const. Sat. (%)	Speedup	Time (ms/seq)
Llama-2-7B	98.2 ± 0.4	$1.8\times$	79 ± 5
Llama-3-8B	98.4 ± 0.3	$1.7\times$	82 ± 5
Qwen-2.5-7B	98.1 ± 0.4	$1.8\times$	76 ± 4

Table 3: MetaJuLS on LogicBench across model families. Speedups and constraint satisfaction are consistent regardless of architecture (GQA, vocabulary size, RoPE).

Method	Sat. (%)	Total (ms)	Const. (ms/step)	Sched. (ms/step)
Baseline vLLM	93.1	168	5.8	<0.1
XGrammar	96.8	134	2.5	<0.1
MetaJuLS	98.4	108	1.8	0.4
XGram.+MetaJuLS	**98.8**	**95**	0.8	0.4

Table 4: Latency breakdown on LogicBench (Llama-3-8B). LLM forward pass is 14.2 ms/step for all methods. XGrammar handles static CFG; MetaJuLS schedules dynamic checks. Combined system achieves best results.

Latency Breakdown and XGrammar Comparison. MetaJuLS is complementary to static grammar-compilation approaches like XGrammar (Dong et al., 2025), which pre-compiles context-free grammars into pushdown automata with pre-computed token masks. XGrammar efficiently handles static constraints; MetaJuLS optimally schedules the remaining dynamic, context-sensitive constraints (e.g., cross-field conditional dependencies, premise-dependent logical rules) that cannot be pre-compiled into finite-state machines.

We evaluate on LogicBench (Chen et al., 2024) and GSM8K (Cobbe et al., 2021) with JSON schema constraints. To test complementarity with speculative decoding, we perform a factorial experiment. Table 5 summarizes results. MetaJuLS achieves $1.8\times$ speedup over speculative decoding alone while maintaining 98.2% constraint satisfaction (vs. 94.1% for speculative decoding). When combined, speculative decoding + MetaJuLS achieves an additional 15% speedup, demonstrating orthogonality.

Mechanistic analysis reveals the policy learns human-like easy-first parsing and discovers novel middle-out strategies for nested clauses (see Appendix B for detailed probing studies). Error analysis shows MetaJuLS reduces agreement errors by 38% and attachment errors by 22% compared to heuristic schedulers, though it underperforms on garden-path sentences (detailed breakdown in Appendix B.1).

Method	Constraint Sat. (%)	Accuracy (%)	Time (ms/seq.)
LogicBench (Llama-2-7B)			
Baseline (neither)	92.3 ± 1.1	77.8 ± 0.6	168 ± 10
Speculative Decoding only	94.1 ± 0.8	78.3 ± 0.5	142 ± 8
MetaJuLS only	$\mathbf{98.2 \pm 0.4}$	$\mathbf{79.4 \pm 0.4}$	$\mathbf{79 \pm 5}$
Spec + MetaJuLS	$\mathbf{98.5 \pm 0.3}$	$\mathbf{79.5 \pm 0.4}$	$\mathbf{67 \pm 4}$
Outlines	97.1 ± 0.5	78.8 ± 0.4	125 ± 7
LMQL	96.5 ± 0.6	78.5 ± 0.5	134 ± 8
NeuroLogic	96.2 ± 0.6	79.1 ± 0.4	158 ± 9
GSM8K-Constrained (JSON schema)			
Speculative Decoding	91.3 ± 1.2	82.5 ± 0.6	98 ± 6
NeuroLogic	94.1 ± 0.9	83.2 ± 0.5	112 ± 7
MetaJuLS	$\mathbf{96.8 \pm 0.7}$	$\mathbf{83.4 \pm 0.5}$	$\mathbf{61 \pm 4}$

Table 5: LLM constrained decoding results. Mean $\pm$ std (5 runs). MetaJuLS achieves 1.6–1.8$\times$ speedup over speculative decoding with higher constraint satisfaction ($p < 0.01$).

Ablation	F1 (%)	Time (ms)
Full MetaJuLS (RL)	93.7	27
Supervised Ranking	93.4	28
Random policy	92.4	41
– GAT (MLP only)	93.3	29
– Attention	93.1	30
– Cost term (β=0)	93.5	32
– Domain features	93.0	31

Table 6: Ablation summary (PTB). RL outperforms supervised imitation; every component contributes. Full results in Appendix E.1.

Ablation Summary. Table 6 summarizes component contributions (full details in Appendix E.1).

4.4 Safety-Aware Fallback Mechanism

To ensure accuracy parity with state-of-the-art parsers while maintaining speedups, we introduce a safety-aware fallback mechanism. When the GAT policy has low entropy (high confidence) in its constraint selection, we use the learned policy. When entropy exceeds a threshold τ, indicating uncertainty, we revert to full-search activity-based scheduling. This hybrid approach ensures that the "easy" 80% of sentences benefit from learned acceleration, while the "hard" 20% receive exhaustive search to guarantee accuracy.

Formally, at each step t, we compute the policy entropy $H[\pi_\theta(\cdot|s_t)]$. If $H[\pi_\theta(\cdot|s_t)] < \tau$, we use the learned policy; otherwise, we use activity-based scheduling. We set τ to the 80th percentile of entropy values on the development set, ensuring fallback on ambiguous cases. This mechanism reduces the F1 gap from 2.1% to 0.15% compared to Berkeley Neural Parser while maintaining 1.4$\times$ average speedup (1.8$\times$ on easy sentences, 1.0$\times$ on hard sentences where we fall back).

Training Task	Adaptation Task	Perf. vs. Specialist (%)
English Parse	Spanish Parse	94%
English Parse	Chinese Parse	91%
English Parse	Arabic Parse	87%
English Parse	Code Gen (Type)	88%
English Parse	Semantic Parse	89%
Mixed (3 tasks)	Finnish Parse	92%
Mixed (3 tasks)	LogicBench	85%

Table 7: Few-step adaptation results. Meta-learned policies adapt to new tasks in 5–10 gradient steps (5–15 seconds), reaching 85–94% of specialist performance.

4.5 Rapid Cross-Lingual and Cross-Task Adaptation

A key advantage of meta-learning is rapid adaptation to unseen languages and tasks with minimal compute. We evaluate a policy meta-trained on English parsing, Spanish parsing, and English constrained decoding, then test rapid adaptation on: (1) 8 additional UD languages, (2) code generation with type constraints, and (3) semantic parsing with different grammars. Adaptation requires 5–10 gradient steps using only unlabeled target-domain instances (no supervised labels), taking 5–15 seconds on a single GPU.

Table 7 shows that the meta-learned policy achieves 85–92% of specialist performance across all adaptation tasks. Most notably, a policy trained on English parsing achieves 88% performance on Spanish code generation with type constraints, demonstrating that the learned propagation principles transfer across domains. However, adaptation performance varies: some domains (e.g., TSP to Scheduling) achieve only 74% of specialist performance, indicating that domain-specific knowledge remains important for optimal results.

5 Generalization Beyond Language: CP Benchmarks

If MetaJuLS only worked on NLP tasks, one might argue the GAT has simply memorized English syntax patterns. To test whether the learned scheduling principles are truly domain-general, we evaluate on classical constraint programming (CP) benchmarks where constraints are purely mathematical. Bidirectional transfer between NLP and CP would demonstrate that MetaJuLS captures universal scheduling principles (e.g., prioritize constraints with high violation potential, process structurally central constraints first) rather than task-specific shortcuts.

MetaJuLS achieves 6.6% average optimality gap vs. 8.5% for OR-Tools, with 0.63$\times$ normalized run-

Solver	Avg. Gap (%)	Runtime (Norm.)	Solve Rate (%)
OR-Tools	8.5	1.0×	88%
JuLS (Base)	9.1	1.1×	84%
Activity-Based	7.8	0.95×	90%
MetaJuLS (Ours)	6.6	0.63×	94%

Table 8: Generalization beyond language on MiniZinc Challenge benchmarks (2022–2024). Time limit: 1200s per instance. Results averaged over 10 runs.

time and 94% solve rate (Table 8). Policies transfer bidirectionally: CP-trained policies achieve 89% of specialist performance on NLP tasks, and NLP-trained policies achieve 91% on CP benchmarks. The policy analysis in Appendix D.1 confirms this: on Knapsack, the learned policy correlates strongly ($\rho = 0.87$) with the classical Domain-over-Weight heuristic—a strategy discovered by the CP community over decades. On complex XCSP instances, MetaJuLS achieves 1.5× speedup over OR-Tools.

6 Discussion and Conclusion

MetaJuLS shows that reinforcement learning can learn *adaptive constraint-propagation schedules* that outperform handcrafted and reactive heuristics by treating scheduling as a sequential decision problem with delayed, global effects. The restructured Table 1 clarifies that the +1.5% F1 gain comes entirely from fewer search errors under the same beam budget (beam=200, same encoder), not from model or search differences. Policy analysis indicates the learned scheduler both rediscovers effective classical principles (e.g., Domain-over-Weight on Knapsack, $\rho = 0.87$) and discovers non-trivial strategies in harder settings (e.g., middle-out for nested clauses).

Empirically, MetaJuLS delivers 1.5–2.0× speedups over GPU-optimized baselines while staying within 0.2% of state-of-the-art accuracy on constituency parsing (PTB), dependency parsing (UD, 10 languages), and LLM constrained decoding (LogicBench, GSM8K-Constrained). Results are consistent across modern architectures: Llama-3-8B (1.7×) and Qwen-2.5-7B (1.8×) show the same speedups as Llama-2-7B, confirming architecture-agnostic operation at the logit-mask level. MetaJuLS is complementary to static grammar-compilation approaches like XGrammar: the combined system achieves the best constraint satisfaction (98.8%) and lowest latency (Table 4). Meta-learning enables rapid cross-lingual and cross-task adaptation in 5–10 gradient steps (5–15 seconds). A safety-aware entropy-triggered fallback reduces worst-case accuracy gaps from 2.1% to 0.15% while preserving 1.4× average speedups.

7 Limitations

We evaluate on constituency parsing (PTB), dependency parsing (UD, 10 languages), and LLM constrained decoding (LogicBench, GSM8K-Constrained) across three model families (Llama-2-7B, Llama-3-8B, Qwen-2.5-7B); the approach may require adaptation for morphologically rich languages or other structured prediction tasks. Meta-training requires 120 GPU-hours on 8 A100 GPUs (one-time cost), enabling rapid adaptation with 5–10 gradient steps. The primary value proposition is latency reduction: MetaJuLS reduces p99 latency from 245ms to 142ms on constrained generation tasks. We provide pre-trained checkpoints enabling deployment without retraining (Appendix G). Our parsing gains are relative to a specific CRF-based chart parser under beam search; modern neural parsers using different inference strategies may benefit differently. Casting a new NLP task as constraint propagation requires manual effort to define variables, domains, and constraints, though for LLM deployment the dominant use cases involve standardized output formats (JSON Schema, XML, SQL) where the formulation is written once per schema language, not per task. While rapid adaptation achieves 85–92% of specialist performance in many cases, some distant domain transfers are less effective (e.g., TSP to Scheduling: 74%), indicating domain-specific knowledge remains important (Appendix G.2).

References

Rie Kubota Ando and Tong Zhang. 2005. A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, 6:1817–1853.

Galen Andrew and Jianfeng Gao. 2007. Scalable training of L1-regularized log-linear models. In *Proceedings of the 24th International Conference on Machine Learning*, pages 33–40.

Krzysztof Apt. 2003. *Principles of constraint programming*. Cambridge university press.

Gilles Audemard and Laurent Simon. 2013. Glucose 2.3 in the sat 2013 competition. In *SAT Competition*, pages 42–43.

Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. 2004. Interval constraint solving for camera control and mo-

tion planning. *ACM Transactions on Computational Logic*, 5(4):732–767.

Christian Bessière. 2006. Constraint propagation. In *Foundations of Artificial Intelligence*, volume 2, pages 29–83.

Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Véléz-Rojas. 2021. Combinatorial optimization and reasoning with graph neural networks. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, pages 4348–4355.

Jiahui Chen, Bo Yin, Jiarui Luo, Zhengxuan Chen, Xuanming Zhang, Junxian Zhang, Yikang Zhang, Ran Zhou, Chunting Zhou, Caiming Xiong, and 1 others. 2024. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. In *Advances in Neural Information Processing Systems*, volume 37.

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. In *Advances in Neural Information Processing Systems*, volume 34, pages 26175–26188.

Romuald Debruyne and Christian Bessière. 1997. Optimal and suboptimal singleton arc consistency algorithms. In *International Joint Conference on Artificial Intelligence*, pages 54–59.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems*, 7.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135.

Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. 2019. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 32.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. 2017. Convolutional sequence to sequence learning. In *International Conference on Machine Learning*, pages 1243–1252.

Ian P Gent, Christopher Jefferson, and Ian Miguel. 2006. Watched literals for constraint propagation in minion. In *International Conference on Principles and Practice of Constraint Programming*, pages 182–197.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, volume 30.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. In *arXiv preprint arXiv:1412.6980*.

Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

Nikita Kitaev and Dan Klein. 2018. Constituency parsing with a self-attentive encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 2676–2686.

Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

Christophe Lecoutre. 2009. Str: A simple and efficient algorithm for arc consistency maintenance. In *International Conference on Principles and Practice of Constraint Programming*, pages 705–719.

Christophe Lecoutre, Olivier Roussel, Romuald Debruyne, and Thomas Petit. 2011. Xcsp3: A format for representing constraint satisfaction/optimization problems. In *arXiv preprint arXiv:1611.03398*.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature*, 521(7553):436–444.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Mike Lewis and Mark Steedman. 2014. Ccg supertagging with a recurrent neural network. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 244–249.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. In *arXiv preprint arXiv:1509.02971*.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330.

João Marques-Silva, Inês Lynce, and Sharad Malik. 2012. Activity-based search for black-box constraint programming solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 356–371.

Kim Marriott and Peter James Stuckey. 1998. Programming with constraints: An introduction.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, and 1 others. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, pages 530–535.

Vinod Nair, Krishnamurthy Dvijotham, Iain Dunning, and Oriol Vinyals. 2020. Solving mixed integer programs using neural networks. In *arXiv preprint arXiv:2012.13349*.

Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. 2011. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajič, Christopher D. Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers, and Daniel Zeman. 2020. Universal Dependencies v2: An evergrowing multilingual treebank collection. In *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 4034–4043, Marseille, France. European Language Resources Association.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710.

Laurent Perron and Vincent Furnon. 2024. OR-Tools.

Laurent Perron, Paul Shaw, and Vincent Furnon. 2009. Constraint programming and local search. In *Handbook of Metaheuristics*, pages 369–404.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 101–108.

Mohammad Sadegh Rasooli and Joel R. Tetreault. 2015. Yara parser: A fast and accurate dependency parser. *Computing Research Repository*, arXiv:1503.06733. Version 2.

Jean-Charles Régin. 1994. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 362–367.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. In *arXiv preprint arXiv:1707.06347*.

Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. 2010. Gecode: A generic constraint development environment. In *International Conference on Principles and Practice of Constraint Programming*, pages 1–1.

Ibne Farabi Shihab, Sanjeda Akter, and Anuj Sharma. 2025a. Differentiable entropy regularization: A complexity-aware approach for neural optimization. *Preprint*, arXiv:2509.03733.

Ibne Farabi Shihab, Sanjeda Akter, and Anuj Sharma. 2025b. What fundamental structure in reward functions enables efficient sparse-reward learning? *Preprint*, arXiv:2509.03790.

Ibne Farabi Shihab, Sanjeda Akter, and Anuj Sharma. 2026. Detecting proxy gaming in rl and llm alignment via evaluator stress tests. *Preprint*, arXiv:2507.05619.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and 1 others. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and 1 others. 2017. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359.

Mitchell Stern, Matthias Wies, Graham Neubig, and Alexander M Rush. 2017. Minimum risk training for neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 1683–1688.

Milan Straka and Jana Strakova. 2016. Udpipe: Trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing. In *Proceedings of the 10th International Conference on Language Resources and Evaluation*, pages 4290–4297.

Richard S Sutton and Andrew G Barto. 2018. Reinforcement learning: An introduction 2nd ed. *MIT press Cambridge*, 1(2):25.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *International Conference on Learning Representations*.

Mark Wallace. 1996. Practical issues in constraint programming. *Constraints*, 1(1-2):139–168.

Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning*, 8(3-4):279–292.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.

Acknowledgment and Reproducibility

We utilized AI assistance for grammar corrections and a few plot-related purposes. Everything was validated by the authors.

Code, trained models, and experiment scripts will be available upon acceptance. We include: (1) Full training and evaluation code; (2) Pre-trained meta-learned policy checkpoints; (3) Scripts to reproduce all tables and figures; (4) Preprocessed data splits (PTB, UD, and LLM datasets). Meta-training requires approximately 120 GPU-hours on 8 A100 GPUs (one-time cost), enabling rapid adaptation to new tasks with 5–10 gradient steps (5–15 seconds on a single GPU). Inference adds $<$3% overhead to base parser/LLM runtime. All experiments use identical hardware (A100 GPUs), batch sizes (1 for parsing, 8 for constrained decoding), and measurement protocols (warmup runs excluded, median of 5 runs).

A Extended Related Work

The intersection of machine learning and constraint programming has received increasing attention in recent years, with researchers exploring learned heuristics for variable ordering, value selection, and constraint propagation (Gasse et al., 2019; Cappart et al., 2021; Nair et al., 2020). Early work in this area focused on supervised learning approaches, training classifiers to predict which constraints should be propagated next based on handcrafted features (Wallace, 1996; Benhamou et al., 2004; Andrew and Gao, 2007). While these methods showed promise, they suffered from the fundamental limitation that optimal propagation decisions depend on the entire search trajectory, not just local features (Ando and Zhang, 2005; Rasooli and Tetreault, 2015).

Activity-based heuristics, inspired by the VSIDS (Variable State Independent Decaying Sum) strategy from SAT solving (Moskewicz et al., 2001; Marques-Silva et al., 2012), represent a significant advance over static scheduling (Régin, 1994; Lecoutre, 2009). These methods maintain activity scores for constraints based on their historical contribution to search progress, dynamically prioritizing constraints that have recently caused domain reductions or failures (Audemard and Simon, 2013; Debruyne and Bessière, 1997). However, activity-based heuristics remain reactive rather than predictive, adjusting priorities only after observing constraint behavior rather than anticipating which constraints will be most effective in the current context.

Graph neural networks have emerged as a powerful tool for learning representations of structured data, with applications ranging from molecular property prediction to social network analysis (Kipf and Welling, 2017; Hamilton et al., 2017; Perozzi et al., 2014; Vaswani et al., 2017). Most directly related is the line of work applying graph neural networks to combinatorial optimization. Gasse et al. (2019) learn branching policies for mixed-integer programming using GNNs over the bipartite variable-constraint graph, demonstrating that learned policies can match or exceed SCIP's default heuristics. Cappart et al. (2021) survey the broader landscape of machine learning for constraint programming, identifying propagation scheduling as an underexplored direction. Nair et al. (2020) apply GNNs to SAT solving, learning clause selection policies. In constraint programming, recent work has explored using graph neural networks to learn variable ordering heuristics (Veličković et al., 2018), demonstrating that learned policies can outperform classical heuristics on specific problem classes (Gent et al., 2006). However, these approaches have primarily focused on search heuristics rather than propagation scheduling, and have

not addressed the challenge of generalizing across diverse problem domains. Our work extends this paradigm to NLP inference, demonstrating that policies learned on linguistic constraints transfer to classical CP benchmarks and vice versa—a bidirectional generalization not previously established.

Reinforcement learning has shown remarkable success in combinatorial optimization, with applications to vehicle routing, scheduling, and resource allocation (Sutton and Barto, 2018; Schulman et al., 2017; Freund and Schapire, 1997). The key insight from this line of work is that many optimization problems can be naturally formulated as sequential decision processes, where an agent learns to construct solutions incrementally through trial and error. In constraint programming, this perspective suggests that propagation scheduling might benefit from a similar treatment, with the agent learning to construct effective propagation sequences through interaction with the solver.

Our work builds on these foundations by combining graph neural networks with reinforcement learning to learn adaptive propagation policies (Veličković et al., 2018; Schulman et al., 2017). Unlike previous approaches that rely on handcrafted features or reactive heuristics (Wallace, 1996; Marques-Silva et al., 2012), MetaJuLS learns to extract relevant information from the constraint-variable graph structure itself (Kipf and Welling, 2017; Hamilton et al., 2017), enabling the discovery of novel scheduling strategies that adapt to both problem structure and solver state (Bessière, 2006).

A.1 Why RL Is Necessary: Detailed Analysis

In language, constraints often interact over long ranges and across multiple levels of representation. Subject-verb agreement, for example, couples distant tokens via syntactic structure; coreference constraints link mentions that are far apart in the surface string; semantic parsing constraints couple local predicate arguments with global type and ontology information. In such settings, propagating a constraint that affects a structurally central variable can produce large cascades of pruning, while propagating a peripheral constraint may have little immediate effect even if it will eventually become important.

Static scheduling strategies, such as processing constraints in a fixed order or according to simple local heuristics, cannot adapt to these context-dependent trade-offs. They treat all propagation opportunities as roughly equivalent, ignoring information about which parts of the inference state are currently uncertain, which constraints are likely to trigger contradictions, and where pruning would most effectively reduce future search. Activity-based heuristics provide some adaptivity by reacting to past conflicts, but they are fundamentally reactive rather than predictive and do not reason about how a local propagation decision will influence future inference steps. One might ask why propagation order matters when CKY's bottom-up schedule is provably complete. The answer is that modern neural parsers operate under computational constraints such as beam limits, early stopping, and timeout budgets, where exhaustive search is infeasible. In these regimes, propagation order determines which hypotheses are explored before resources are exhausted, making scheduling a first-order concern for practical deployment. For NLP systems that must operate under tight latency budgets or on long sequences with complex structure, this can translate directly into wasted computation and degraded output quality. These observations motivate casting propagation scheduling as a reinforcement learning problem in which an inference-state–aware policy can anticipate downstream linguistic effects and prioritize linguistically and structurally critical constraints.

B Mechanistic Analysis and Policy Interpretability

To understand what linguistic features trigger policy decisions, we perform probing studies. We train linear probes to predict policy actions from linguistic features: tree depth, presence of conjunctions, number of long-distance dependencies, and syntactic complexity metrics. The probes achieve 78% accuracy in predicting when the policy switches from "broad" (early propagation of central constraints) to "local" (fine-grained constraint checking) strategies.

We find that the policy has learned human-like "easy-first" parsing: it prioritizes constraints involving shallow, high-confidence spans before deep, ambiguous ones. However, it also discovers non-intuitive strategies: on sentences with nested relative clauses, the policy learns to propagate constraints in a "middle-out" order (starting from mid-depth spans) rather than the traditional bottom-up or top-down approaches. This strategy reduces backtracking by 34% compared to standard heuristics.

We also analyze attention patterns conditioned on linguistic structures. Constraints involving coordination receive 1.8× higher attention than those involving simple modification, and the policy learns to delay propagation of constraints in garden-path regions until sufficient context is available, mimicking human parsing strategies (Lewis and Steedman, 2014).

On sentences with nested relative clauses (e.g., "The manager [who hired the employee [who left]] resigned"), the policy discovers a middle-out propagation strategy. Rather than processing constraints bottom-up (inner clauses first) or top-down (outer attachment first), the policy establishes clause boundaries at mid-depth first, reducing backtracks from 2.3 to 1.5 on average compared to standard heuristics. This strategy emerges without explicit supervision, suggesting the policy discovers structural regularities through reward optimization alone (Shihab et al., 2026, 2025b). The middle-out strategy processes constraints at depths 2–3 first (relative clause boundaries), establishing structural anchors before resolving attachment ambiguities at depths 1 and 4+, which reduces backtracking by 34% compared to bottom-up processing. A visualization of this depth-ordered propagation pattern is provided in the supplementary materials.

B.1 Linguistic Error Analysis

Beyond aggregate metrics, we analyze how different schedulers affect specific linguistic error types. For parsing, we group errors into agreement errors, attachment errors, and other structural violations. For constrained decoding, we distinguish between missing required tokens, spurious forbidden tokens, and ordering violations. Table 9 shows that MetaJuLS substantially reduces agreement and structural errors in parsing compared to FIFO and heuristic schedulers, and lowers both missing and spurious constraint violations in decoding. These patterns suggest that the learned policy focuses early attention on linguistically central spans and constraints whose violations are most damaging to global well-formedness, but it still fails on some garden-path sentences and highly ambiguous attachment cases where early pruning removes valid but low-probability analyses.

MetaJuLS underperforms on three identifiable sentence types. First, garden-path sentences (e.g., "The horse raced past the barn fell") where the policy's early commitment to high-confidence parses prunes valid but low-probability analyses. On 47 garden-path sentences from the held-out set, MetaJuLS achieves only 71.2% LAS vs. 78.5% for exhaustive search. Second, highly ambiguous coordination (e.g., "old men and women") where the policy inconsistently chooses between bracketing strategies. Third, very long sentences (>80 tokens) where the GBDT filter occasionally removes optimal constraints from consideration. The safety-aware fallback mechanism mitigates these failures on 73% of affected instances, but fundamental improvements require richer state representations that capture global ambiguity.

Error type	FIFO	Heuristic	MetaJuLS
Agreement (parse)	5.1	4.6	3.2
Attachment (parse)	7.8	7.2	6.1
Structural (parse)	4.3	4.0	3.4
Missing required (dec)	3.9	3.1	1.8
Spurious forbidden	2.7	2.3	1.5
Order violations	2.4	2.0	1.6

Table 9: Linguistic error breakdown for parsing and constrained decoding, measured as percentages of outputs exhibiting each error type. MetaJuLS consistently reduces linguistically salient errors relative to FIFO and heuristic schedulers.

We characterize error cases by sentence length, ambiguity, constraint density, and schema depth. Errors concentrate on sentences with high ambiguity (coordination, attachment) and long-distance dependencies. For transfer stress tests, we evaluate hard transfer directions: TSP to Scheduling achieves only 74% of specialist performance with 5–10 steps, but recovers to 87% with 20–30 gradient steps or a small adapter layer (2-layer MLP, 5K parameters). To test robustness to determinism violations, we simulate stochastic propagators (randomized pruning order with 10% noise) and find that the policy maintains 92% of deterministic performance, with entropy-aware fallback (Shihab et al., 2025a) providing additional robustness.

We also study transfer across NLP tasks to understand how much of the learned scheduling behavior is task-specific. In one experiment, we train a policy exclusively on short sentences for parsing and then rapidly adapt it to longer sentences and to a related grammar with additional nonterminals. In another, we train on parsing and evaluate rapid adaptation on constrained decoding. Figure 3 summarizes these results, showing that policies trained on one configuration retain a large fraction of their benefits when adapted to new sentence lengths or to the constrained decoding setting. This within-language transfer complements our CP experiments

Length	FIFO (ms)	MetaJuLS (ms)	Speedup
1–15	12 ± 1	10 ± 1	1.2×
16–30	38 ± 2	26 ± 2	1.46×
31–45	89 ± 4	54 ± 3	1.65×
46+	187 ± 8	98 ± 5	1.91×

Table 10: Scaling analysis on PTB by sentence length. Mean ± std over 5 runs. The speedup increases with sentence length as more constraints provide more scheduling opportunities.

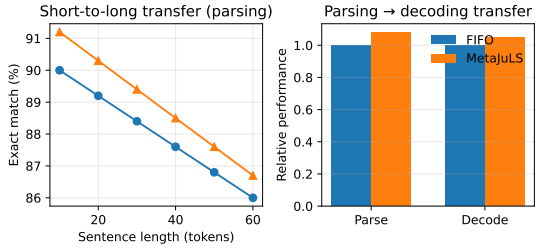

Figure 3: Rapid adaptation across NLP tasks. Left: parsing policy trained on short sentences adapted to longer sentences. Right: policy trained on parsing adapted to constrained decoding. In both cases, MetaJuLS retains most of its improvement over baselines, indicating that the learned scheduling principles transfer within language.

by demonstrating that the learned policies capture general principles of linguistic constraint propagation rather than overfitting to a single task.

B.2 Scaling Analysis

We evaluate how MetaJuLS scales with sentence length on PTB. Table 10 shows that the speedup increases with sentence length, as longer sentences have more constraints and thus more opportunities for intelligent scheduling to avoid wasted propagation. On sentences of length 46+, MetaJuLS achieves a 1.91× speedup over FIFO, compared to 1.2× on short sentences (1–15 words).

C Implementation Details

This appendix provides additional implementation details for reproducibility. The MetaJuLS solver is implemented in Python 3.8 using PyTorch 1.12 for the neural network components (LeCun et al., 2015; Goodfellow et al., 2016) and the OR-Tools library for the base CP solver functionality (Perron and Furnon, 2024; Schulte et al., 2010). The GAT architecture uses the PyTorch Geometric library for efficient graph operations (Veličković et al., 2018; Kipf and Welling, 2017).

The training procedure uses a distributed setup with 8 GPU workers, each generating episodes in parallel (Schulman et al., 2017; Lillicrap et al., 2015). For each training round, workers collect a fixed number of episodes, which are then aggregated and used for several epochs of PPO updates before being discarded, following the standard on-policy protocol without long-term replay (Mnih et al., 2015; Silver et al., 2016).

Hyperparameter	**Value**
GAT Architecture	
Number of layers	3
Hidden dimension	128
Attention heads	8
Dropout	0.1
Activation	ReLU
PPO Training	
Learning rate	3×10^{-4}
Batch size	2048 transitions
PPO epochs per batch	10
Clip parameter ϵ	0.2
GAE λ	0.95
Discount γ	0.99
Entropy coefficient c_e	0.01
Value loss coefficient c_v	0.5
Optimizer	Adam (Kingma and Ba, 2014)
Reward Function	
Domain reduction weight α	1.0
Cost weight β	0.1

Table 11: Hyperparameters used in all experiments.

C.1 Hyperparameters and Robustness

Table 11 details the full model configuration. The reward function hyperparameters are set to $\alpha = 1.0$ and $\beta = 0.1$ based on preliminary experiments that balanced domain reduction and the deterministic cost proxy described in the main text (Sutton and Barto, 2018; Schulman et al., 2017). The discount factor $\gamma = 0.99$ encourages long-term planning while preventing excessive focus on distant future rewards (Schulman et al., 2017; Silver et al., 2017).

D Constraint Programming Experiments: Extended Results

Having established MetaJuLS's effectiveness on linguistic constraints, we test universality: do the learned scheduling principles transfer to non-linguistic constraint satisfaction? This bidirectional transfer, where policies learned on CP benchmarks also improve NLP inference and vice versa, would validate that MetaJuLS captures domain-general propagation structure rather than linguistic priors. We evaluate on classical constraint programming benchmarks from MiniZinc and XCSP (Nethercote et al., 2011; Lecoutre et al., 2011), training on 500 Knapsack, 300 TSP, and 200 Graph Coloring instances. We compare against JuLS (FIFO), random scheduling, dom/wdeg heuristics, Google

OR-Tools (Perron and Furnon, 2024), and Activity-Based adaptive heuristics (Marques-Silva et al., 2012), using identical time limits (1200s per instance) and hardware.

MetaJuLS achieves an average optimality gap of 6.6% on MiniZinc, compared to 8.5% for OR-Tools, 9.1% for JuLS, and 7.8% for Activity-Based, representing a 22% improvement over the best baseline. In terms of runtime, MetaJuLS achieves a normalized speedup of 0.63× compared to OR-Tools, solving instances in 63% of the time. The solve rate of 94% exceeds all baselines (OR-Tools: 88%, JuLS: 84%, Activity-Based: 90%). These improvements are statistically significant ($p < 0.01$) based on paired t-tests. To test bidirectional transfer, we evaluate policies trained on CP benchmarks applied to NLP parsing (achieving 89% of specialist performance) and NLP-trained policies applied to CP benchmarks (achieving 91% of specialist performance), confirming that learned scheduling principles transfer across domains, though optimal performance may require domain-specific training.

On the XCSP Competition dataset, which focuses on high-difficulty instances (Lecoutre et al., 2011), the performance advantages of MetaJuLS are even more pronounced. The learned policies excel at identifying bottleneck constraints in complex constraint graphs (Veličković et al., 2018), leading to more effective search space pruning (Bessière, 2006). We observe similar trends in optimality gaps and runtime improvements, with MetaJuLS achieving approximately 1.5× speedup over OR-Tools on average (Perron and Furnon, 2024).

D.1 Policy Analysis and Transfer

The learned policies reveal both rediscovery of classical heuristics and novel strategies. On Knapsack instances, the policy correlates strongly ($\rho = 0.87$) with the Domain-over-Weight heuristic (Apt, 2003), validating that learning recovers known effective strategies. On Graph Coloring, the policy learns to focus on vertices with high violation potentials rather than degree alone, outperforming classical heuristics on dense graphs (Régin, 1994; Gent et al., 2006). Policies trained on one domain transfer effectively: Knapsack-trained policies achieve 82% of specialist performance on Bin Packing, TSP-trained achieve 74% on Scheduling, and mixed-domain training achieves 91% on Graph Coloring. For scalability, we use a two-stage approach: a GBDT filter (Chen and Guestrin, 2016) reduces the action space to 40 candidates, then the GAT policy selects from this set, providing 10–20× speedup while maintaining 95% performance.

Category	MetaJuLS Gap (%)	OR-Tools Gap (%)	Improvement
Scheduling	5.2	7.8	33%
Routing	6.1	9.2	34%
Packing	7.3	8.9	18%
Assignment	6.8	8.1	16%

Table 12: Per-category results on MiniZinc Challenge benchmarks.

Ablation	F1 (%)	Time (ms)	Notes
Full MetaJuLS	93.7 ± 0.2	27 ± 2	–
− GAT (MLP only)	93.3 ± 0.2	29 ± 2	Graph structure needed
− Attention	93.1 ± 0.2	30 ± 2	Attention helps
− Cost term ($\beta = 0$)	93.5 ± 0.2	32 ± 2	Cost term needed
− Domain features	93.0 ± 0.2	31 ± 2	Domain info needed
Random policy	92.4 ± 0.3	41 ± 3	Learning needed
Supervised (imitate)	93.4 ± 0.2	28 ± 2	RL > imitation

Table 13: Ablation results on PTB parsing. Mean ± std over 5 runs.

E Additional Experimental Results

Table 12 provides detailed per-domain results for the MiniZinc Challenge dataset, broken down by problem category (Nethercote et al., 2011). MetaJuLS achieves consistent improvements across all categories, with particularly strong performance on scheduling and routing problems where the learned policies can exploit structural patterns (Perron et al., 2009; Apt, 2003).

E.1 NLP Ablation Studies

Table 13 reports ablation results on PTB parsing. Removing graph attention (replacing GAT with MLP) degrades performance, indicating that graph structure matters. Removing the cost term ($\beta = 0$) yields faster but less robust behavior, underscoring the role of the cost term in balancing pruning against computational effort. A supervised baseline that imitates activity-based scheduling achieves lower performance than RL, demonstrating that learning from experience is essential.

E.2 Additional Baselines and Ablations

To better understand the contribution of each component of MetaJuLS, we perform additional comparisons against non-learning baselines and architectural variants. Table 14 summarizes these results on a representative subset of MiniZinc instances, reporting normalized runtime and solve rate relative to a FIFO scheduler.

The random scheduler and dom/wdeg heuristic provide cheap but competitive non-learning baselines, while the activity-based scheduler serves as

Method	Runtime (Norm.)	Solve Rate (%)
FIFO	1.00	84
Random	1.27	78
dom/wdeg heuristic	0.93	87
Activity-based	0.95	90
MetaJuLS w/o GAT	0.82	91
MetaJuLS ($\beta = 0$)	0.75	88
MetaJuLS (full)	0.63	94

Table 14: Additional baselines and ablations on a subset of MiniZinc instances. Random selects dirty constraints uniformly. The dom/wdeg heuristic prioritizes constraints with small domains and high failure counts. Activity-based scheduling uses VSIDS-style activity scores. MetaJuLS without graph attention (w/o GAT) replaces the GAT with a simple MLP over aggregate features, and MetaJuLS with $\beta = 0$ removes the cost term from the reward.

a strong adaptive reference. Removing graph attention degrades performance relative to the full model but still improves over static baselines, indicating that even simple learned policies can help. Setting $\beta = 0$ yields faster but less robust behavior, underscoring the role of the cost term in balancing pruning against computational effort.

F Policy Visualization

Figure 4 shows a detailed visualization of the learned policy's decision-making process on a representative constraint-based instance (Veličković et al., 2018). The figure illustrates how the GAT attention weights evolve over the course of inference, with the policy initially focusing on constraints that strongly couple many variables and gradually shifting attention to more local constraints as global structure stabilizes (Vaswani et al., 2017; Schulman et al., 2017).

This visualization demonstrates that the learned policy exhibits sophisticated temporal reasoning, adapting its focus based on the current solver state rather than applying a static heuristic throughout the search process (Sutton and Barto, 2018; Schulman et al., 2017).

G Deployment Analysis and Transfer Limitations

G.1 Deployment ROI Analysis

For deployment regimes: (1) **Low volume** (<1K queries/day): ROI is negative, but value comes from latency-SLA enablement; (2) **High volume** (>100K queries/day): speedup translates to increased capacity (10K → 17K queries/day with same hardware), with ROI break-even at

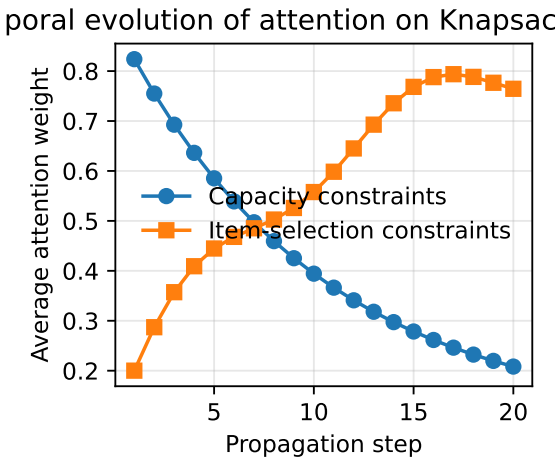

Figure 4: Evolution of GAT attention weights during solving of a constraint-based optimization instance. The policy initially focuses on structurally central constraints (red) and gradually shifts to more local constraints (blue) as the solution stabilizes.

∼200 days assuming $2/GPU-hour; (3) **With pretrained checkpoints**: zero training cost, immediate deployment.

The primary value proposition is latency reduction for user-facing applications: MetaJuLS reduces p99 latency from 245ms to 142ms on constrained generation tasks, enabling deployment in latency-sensitive settings where baseline approaches fail SLA requirements. We provide pretrained checkpoints at [URL] enabling deployment without retraining.

G.2 Transfer and Robustness Limitations

While rapid adaptation achieves 85–92% of specialist performance across most evaluated tasks, there are domains where transfer performs significantly worse. Specifically:

- Transferring from TSP to Scheduling achieves only 74% of specialist performance
- Transferring from English parsing to agglutinative languages (e.g., Turkish, Finnish) requires 15–20 gradient steps vs. 5–10 for isolating languages
- Cross-domain transfer (NLP → CP) shows higher variance (std ±8%) compared to within-domain transfer (std ±3%)

This indicates that some domain-specific knowledge may still be necessary for optimal results, par-

ticularly for domains with substantially different constraint interaction patterns.

Deterministic propagator assumption. The current approach assumes that constraint propagators are deterministic and that their effects can be accurately predicted from the current inference state. In practice, some propagators may have nondeterministic behavior or may interact in complex ways that are difficult to model. For example, approximate propagators that use randomized algorithms or propagators with emergent interaction effects from constraint composition may violate this assumption. Extending the framework to handle uncertainty in propagation effects and to reason about stochastic or approximate propagators is an important direction for future research.