Automatically Uncovering Intended Domain Constraints in Automated Planning

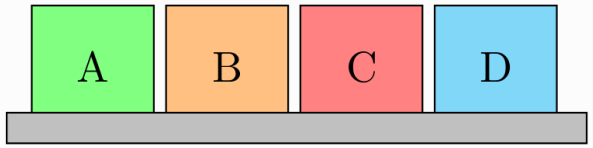
Elliot Gestrin · Johannes Fichte · Jendrik Seipp

Linköping University, Sweden

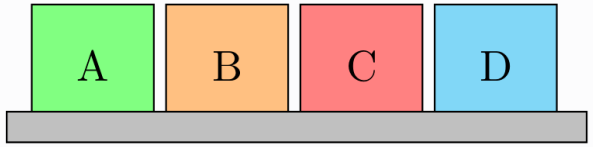
KEPS 2026

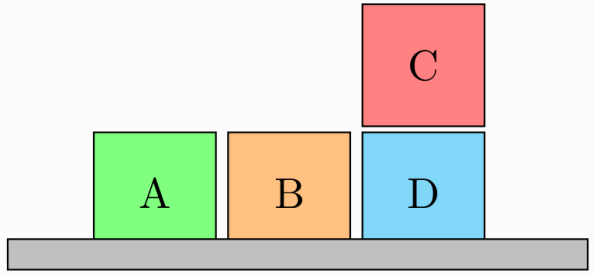
Is it Blocksworld?

Is it Blocksworld?

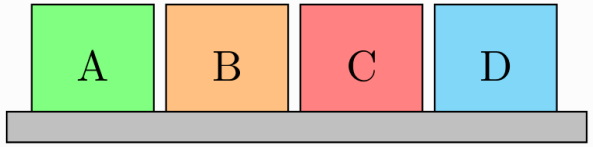


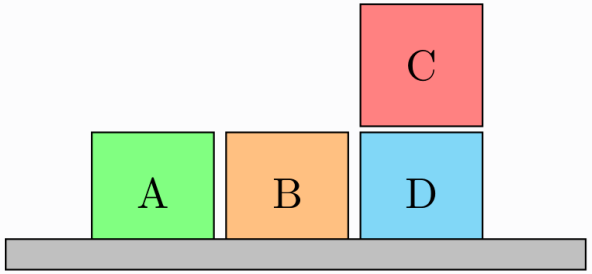
Is it Blocksworld?

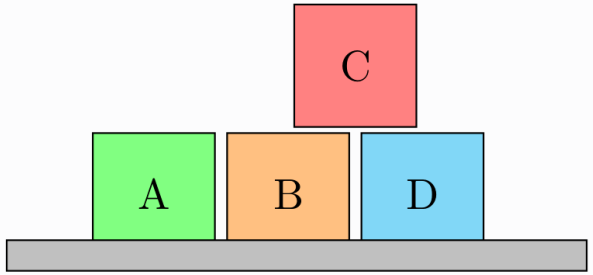




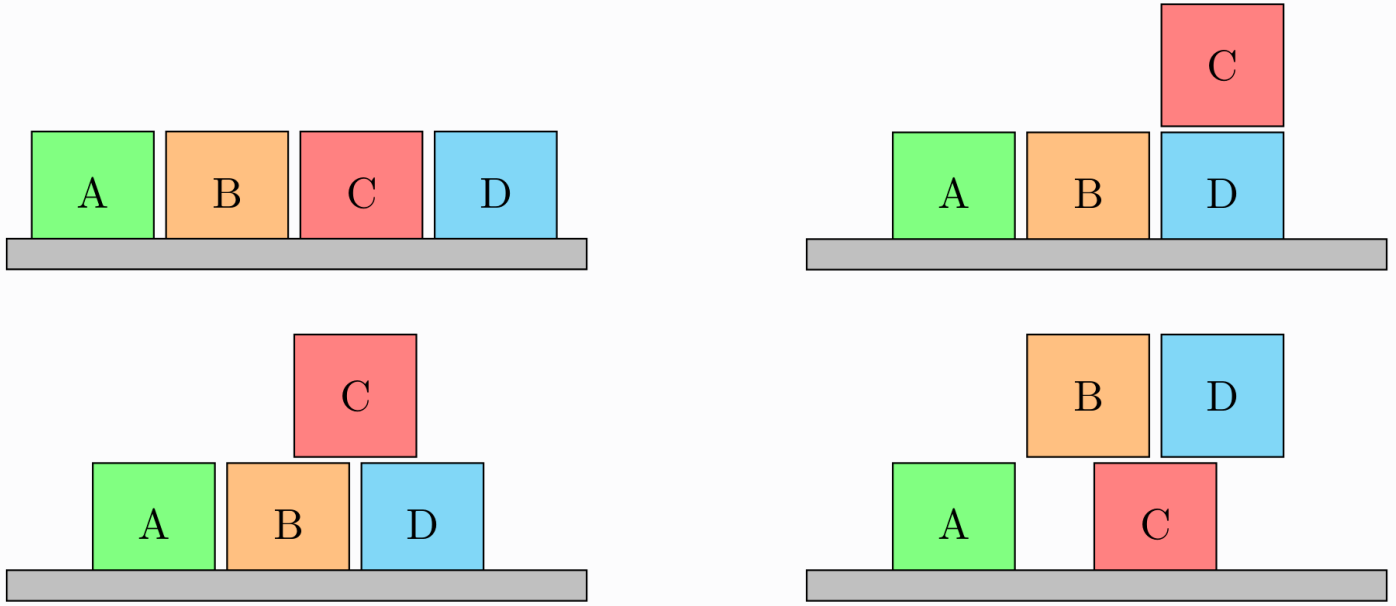
Is it Blocksworld?



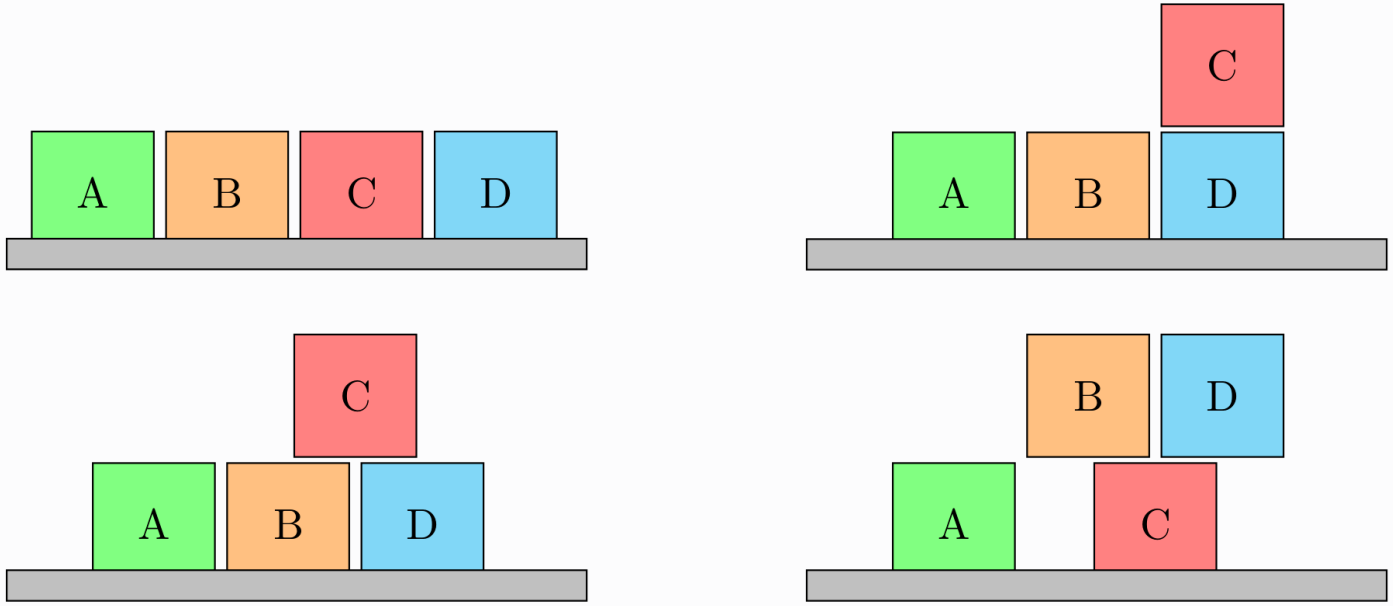




Is it Blocksworld?

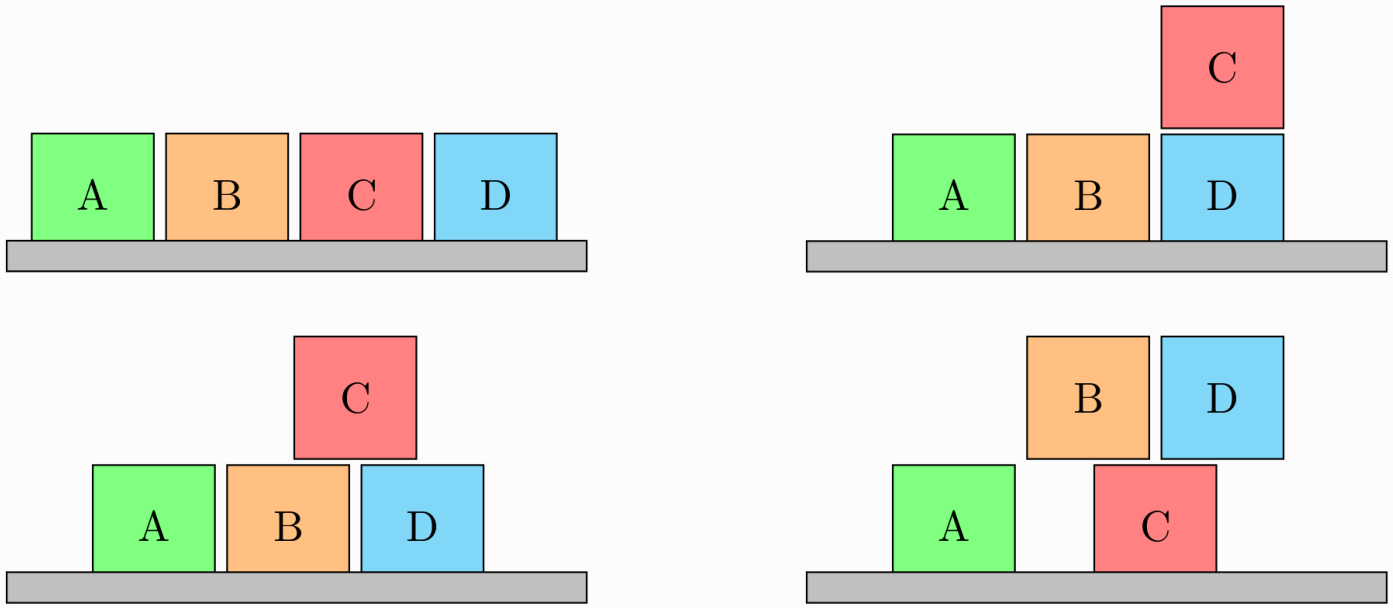


Is it Blocksworld?



All *syntactically* valid

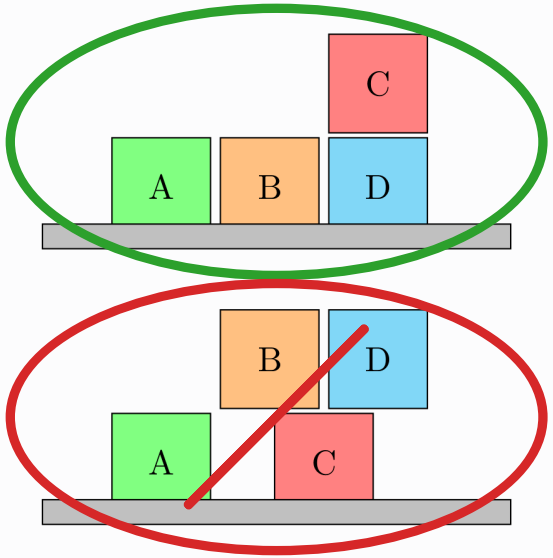
Is it Blocksworld?



All *syntactically* valid

Not all *semantically* valid

Formalizing initial constraints

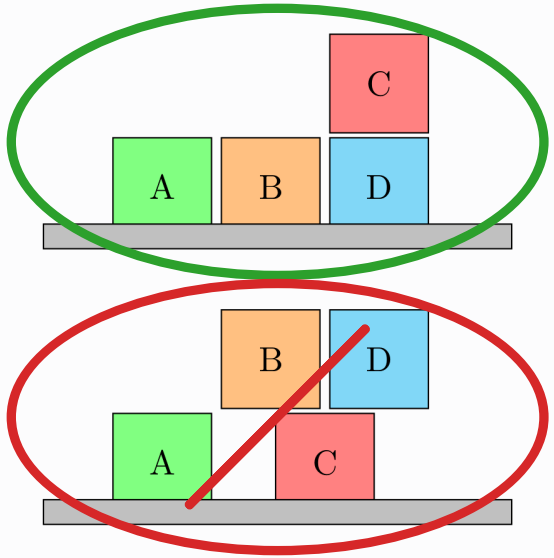


[1] Grundke, Röger, Helmert. Formal Representations of Classical Planning Domains. ICAPS 2024.

[2] Shleyfman, Karpas. Position Paper: Reasoning About Domains with PDDL. AAAI Spring Symposium (SS-18), 2018.

Formalizing initial constraints

- Intent exists

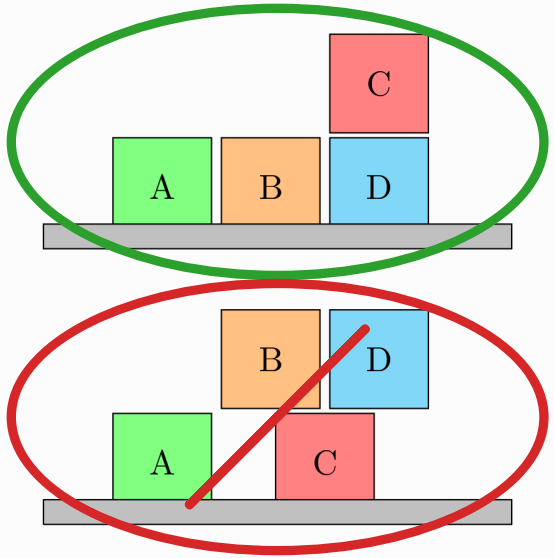


[1] Grundke, Röger, Helmert. Formal Representations of Classical Planning Domains. ICAPS 2024.

[2] Shleyfman, Karpas. Position Paper: Reasoning About Domains with PDDL. AAAI Spring Symposium (SS-18), 2018.

Formalizing initial constraints

- Intent exists
- Provided implicitly

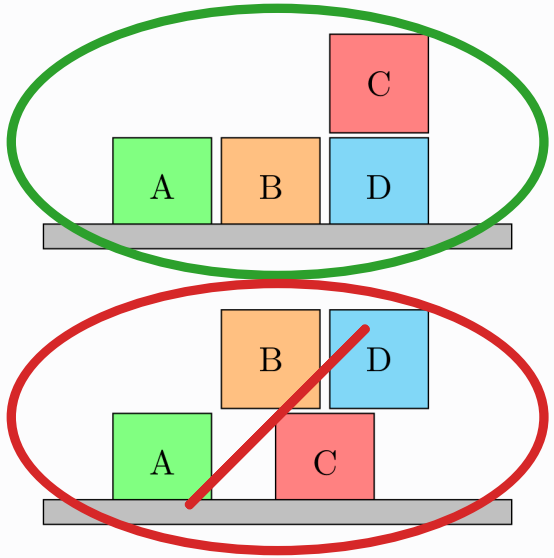


[1] Grundke, Röger, Helmert. Formal Representations of Classical Planning Domains. ICAPS 2024.

[2] Shleyfman, Karpas. Position Paper: Reasoning About Domains with PDDL. AAAI Spring Symposium (SS-18), 2018.

Formalizing initial constraints

- Intent exists
- Provided implicitly
- Axioms[1]

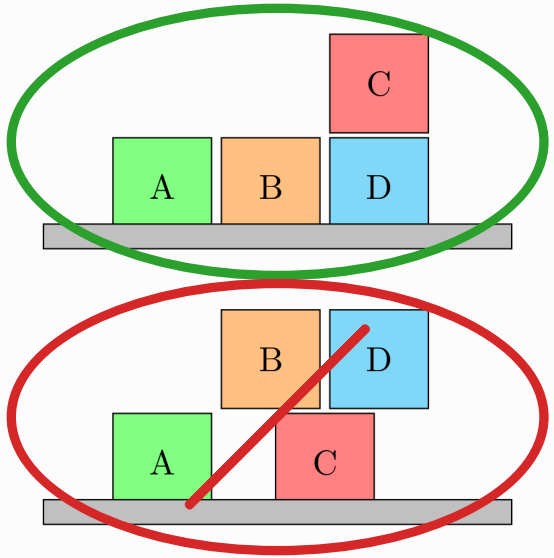


[1] Grundke, Röger, Helmert. Formal Representations of Classical Planning Domains. ICAPS 2024.

[2] Shleyfman, Karpas. Position Paper: Reasoning About Domains with PDDL. AAAI Spring Symposium (SS-18), 2018.

Formalizing initial constraints

- Intent exists
- Provided implicitly
- Axioms[1]
- (PDDL3) Constraints[2]

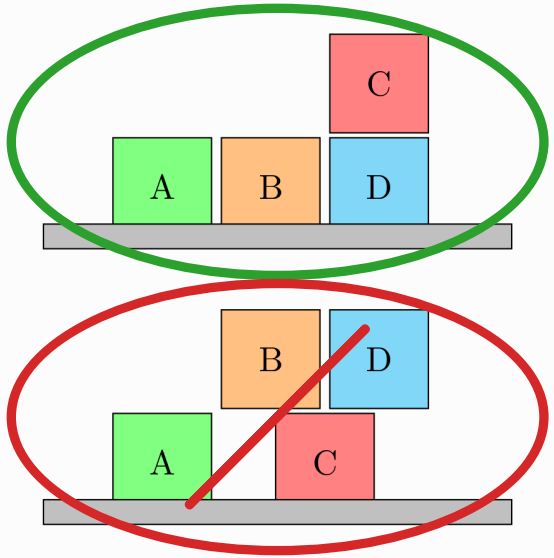


[1] Grundke, Röger, Helmert. Formal Representations of Classical Planning Domains. ICAPS 2024.

[2] Shleyfman, Karpas. Position Paper: Reasoning About Domains with PDDL. AAAI Spring Symposium (SS-18), 2018.

Formalizing initial constraints

- Intent exists
- Provided implicitly
- Axioms[1]
- (PDDL3) Constraints[2]
- Manual effort



[1] Grundke, Röger, Helmert. Formal Representations of Classical Planning Domains. ICAPS 2024.

[2] Shleyfman, Karpas. Position Paper: Reasoning About Domains with PDDL. AAAI Spring Symposium (SS-18), 2018.

Learning intended constraints

Learning intended constraints

- What's available?

Learning intended constraints

- What's available?
  - The domain

Learning intended constraints

- What's available?
  - The domain
  - Legal tasks

Learning intended constraints

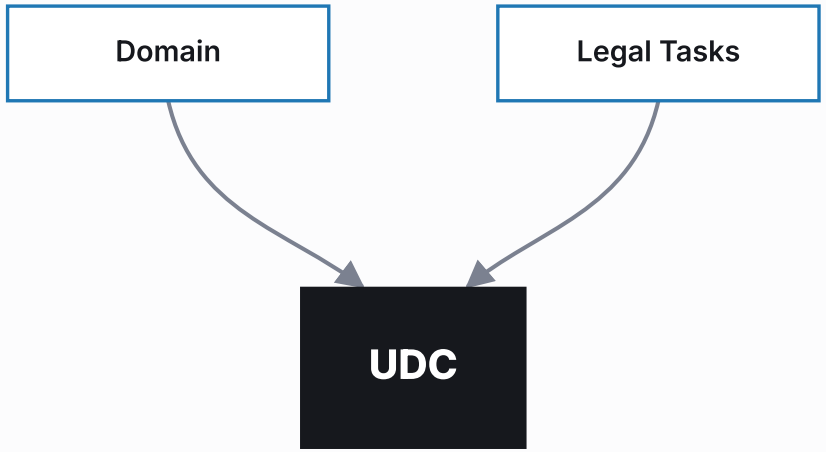
- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks

Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?

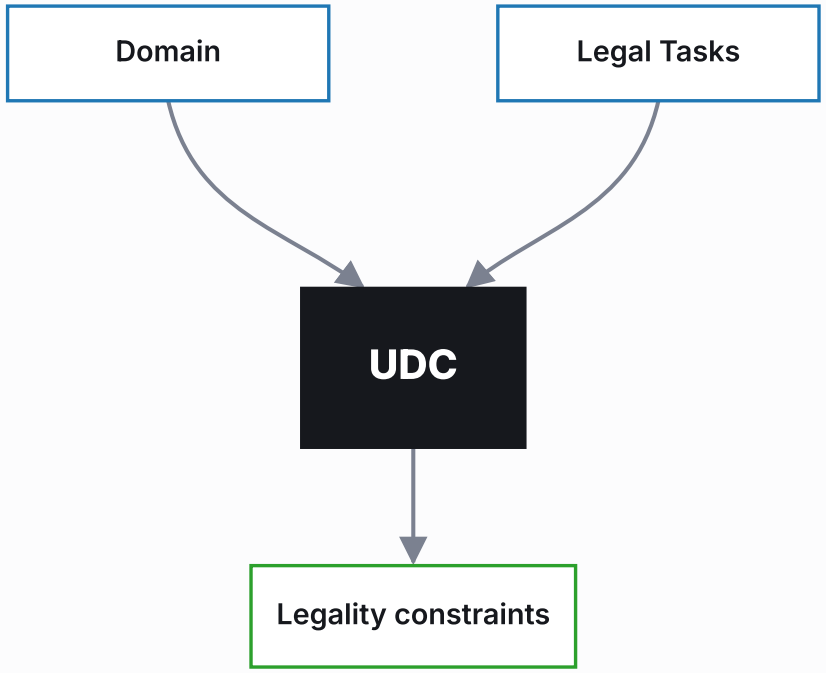
Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?
  - Uncover intended Domain Constraints (UDC)



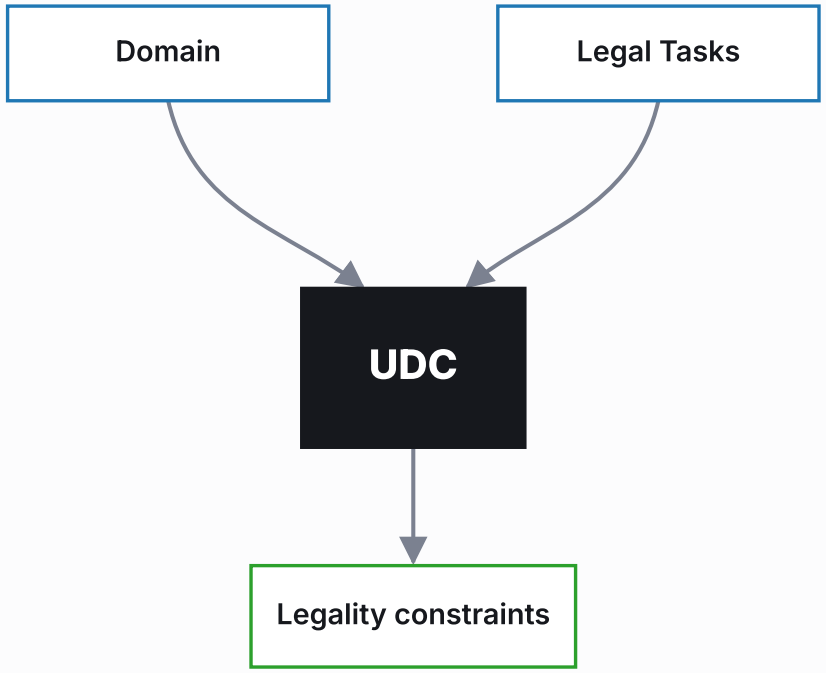
Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?
  - Uncover intended Domain Constraints (UDC)
  - Legality constraints



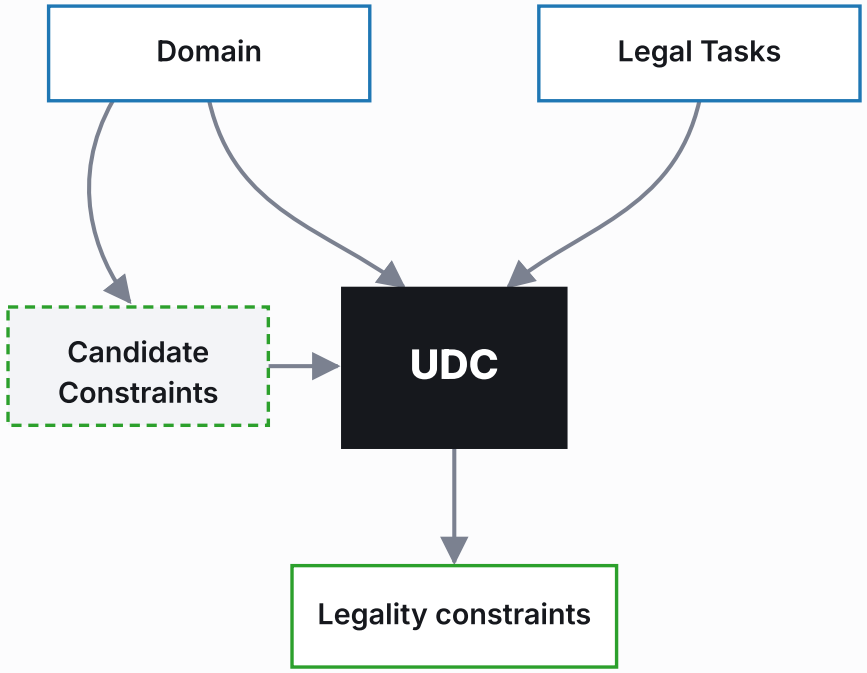
Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?
  - Uncover intended Domain Constraints (UDC)
  - Legality constraints
- How do we do it?



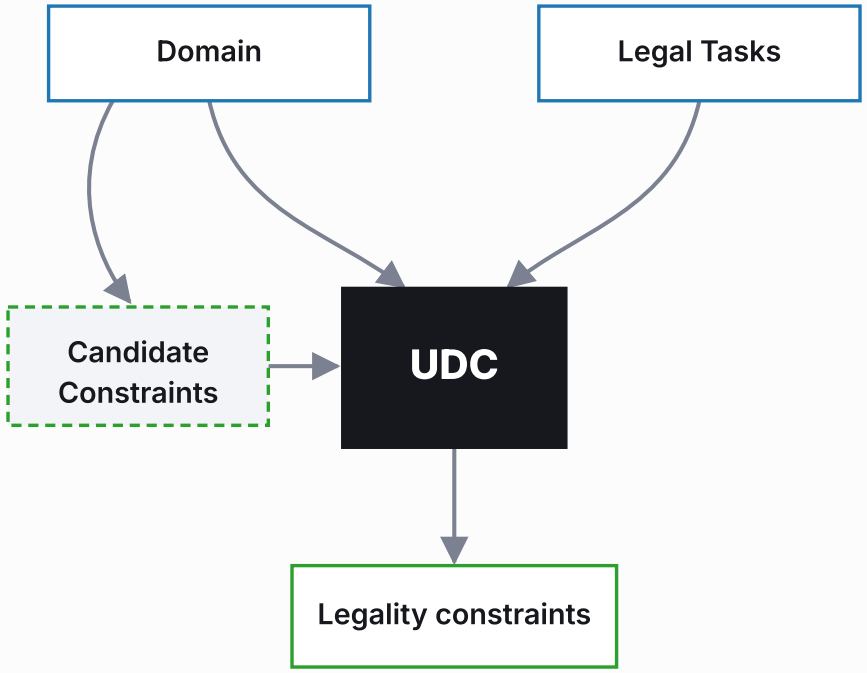
Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?
  - Uncover intended Domain Constraints (UDC)
  - Legality constraints
- How do we do it?
  - Candidate constraints



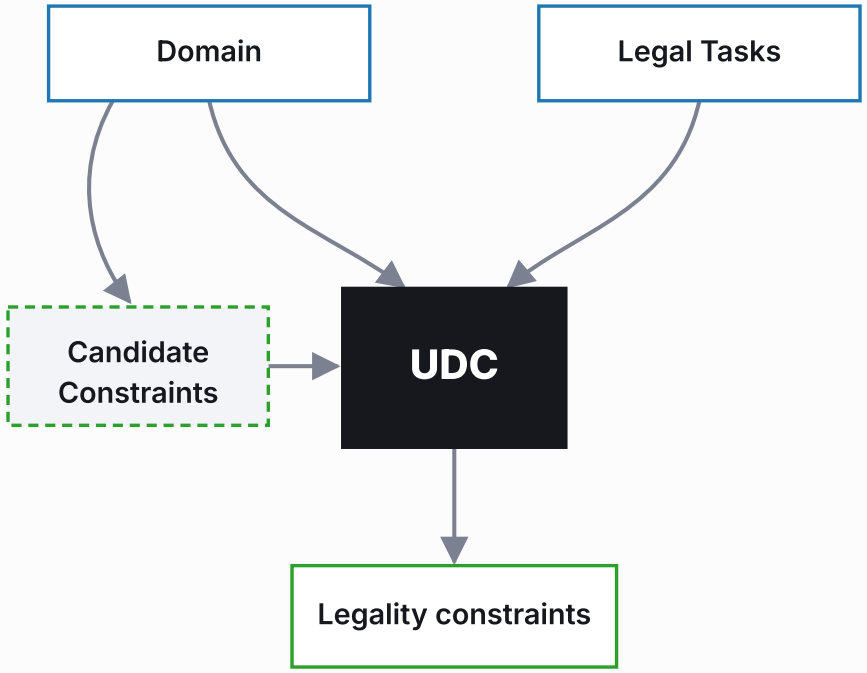
Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?
  - Uncover intended Domain Constraints (UDC)
  - Legality constraints
- How do we do it?
  - Candidate constraints
  - Choose the best



Learning intended constraints

- What's available?
  - The domain
  - Legal tasks
  - *Only* legal tasks
- What's the goal?
  - Uncover intended Domain Constraints (UDC)
  - Legality constraints
- How do we do it?
  - Candidate constraints
  - Choose the best
  - Do it efficiently



Candidate Constraints

$$\forall_{x,y}\ \mathrm{on}(x,y) \rightarrow \neg\exists_{z \neq y}\ \mathrm{on}(x,z)$$

a block is on at most one other

$$\forall_{x,y}\ \mathrm{on}(x,y) \rightarrow \neg\exists_{z \neq x}\ \mathrm{on}(z,y)$$

a block has at most one block on it

Candidate Constraints

- Any boolean sentences

$$\forall_{x,y}\ on(x,y) \rightarrow \neg\exists_{z \neq y}\ on(x,z)$$

a block is on at most one other

$$\forall_{x,y}\ on(x,y) \rightarrow \neg\exists_{z \neq x}\ on(z,y)$$

a block has at most one block on it

Candidate Constraints

- Any boolean sentences
- Presented as a set

$$\forall_{x,y}\ on(x,y) \rightarrow \neg\exists_{z \neq y}\ on(x,z)$$

a block is on at most one other

$$\forall_{x,y}\ on(x,y) \rightarrow \neg\exists_{z \neq x}\ on(z,y)$$

a block has at most one block on it

Candidate Constraints

- Any boolean sentences
- Presented as a set
- We consider two:

$$\forall_{x,y}\ on(x,y) \rightarrow \neg\exists_{z \neq y}\ on(x,z)$$

a block is on at most one other

$$\forall_{x,y}\ on(x,y) \rightarrow \neg\exists_{z \neq x}\ on(z,y)$$

a block has at most one block on it

Candidate Constraints

- Any boolean sentences
- Presented as a set
- We consider two:
  - Typed Implications

$$\forall_{x,y}\ \mathrm{on}(x,y) \rightarrow \neg\exists_{z \neq y}\ \mathrm{on}(x,z)$$

a block is on at most one other

$$\forall_{x,y}\ \mathrm{on}(x,y) \rightarrow \neg\exists_{z \neq x}\ \mathrm{on}(z,y)$$

a block has at most one block on it

Candidate Constraints

- Any boolean sentences
- Presented as a set
- We consider two:
  - Typed Implications
  - LLM-generated

$$\forall_{x,y}\ \mathrm{on}(x,y) \rightarrow \neg\exists_{z \neq y}\ \mathrm{on}(x,z)$$

a block is on at most one other

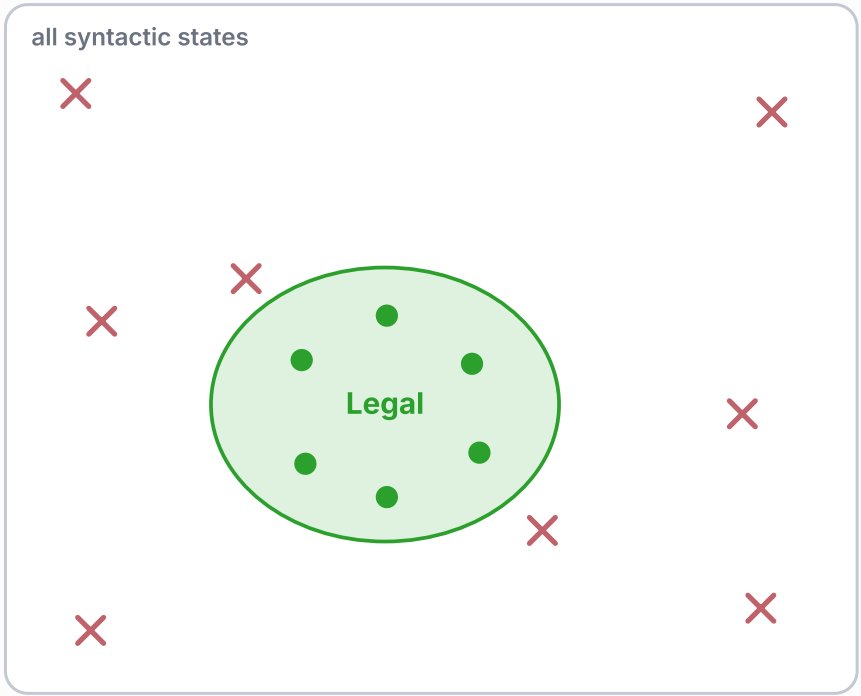
$$\forall_{x,y}\ \mathrm{on}(x,y) \rightarrow \neg\exists_{z \neq x}\ \mathrm{on}(z,y)$$

a block has at most one block on it

Best Candidate Constraint Subset

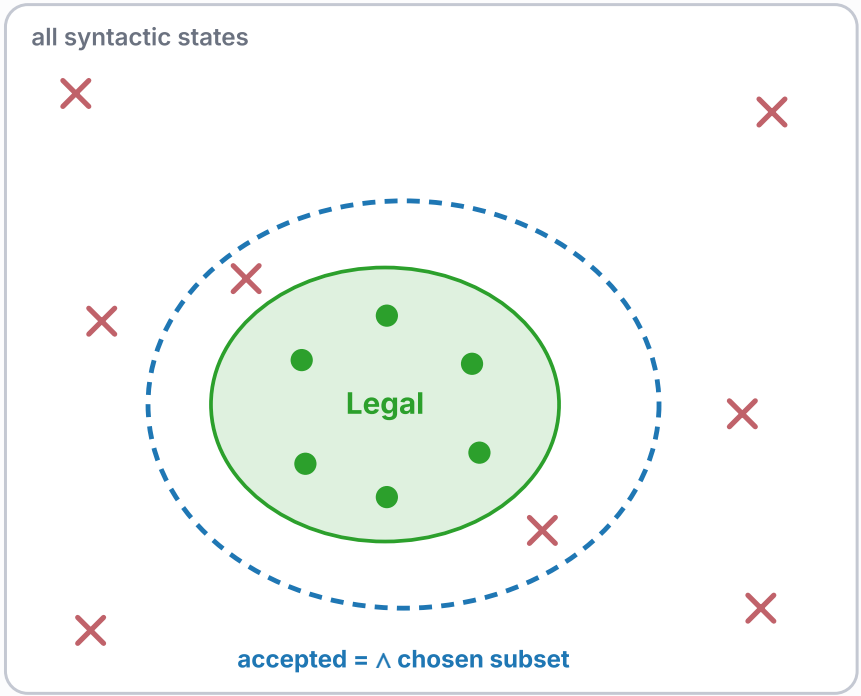
Best Candidate Constraint Subset

- The subset that:



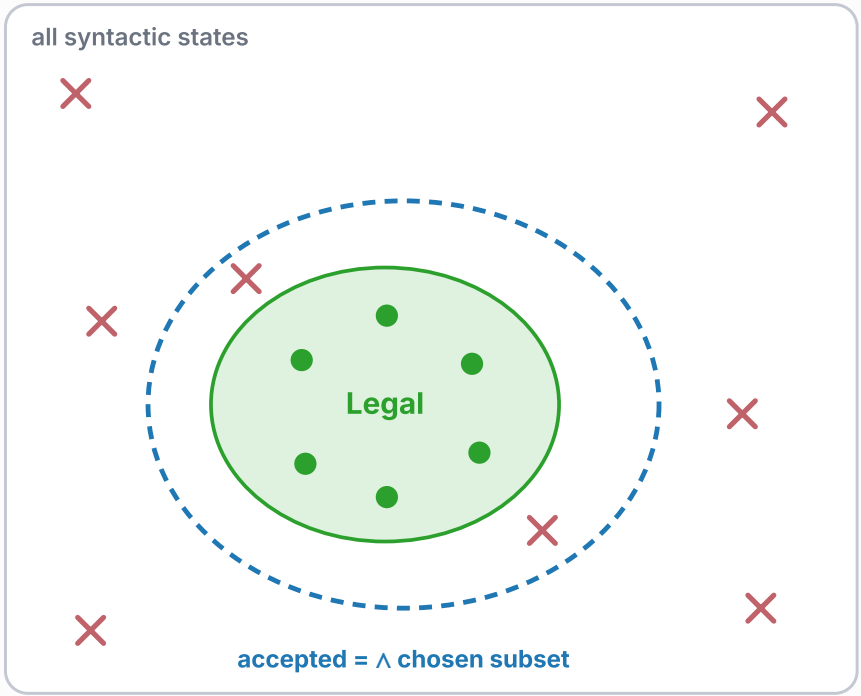
Best Candidate Constraint Subset

- The subset that:
  - Allows all legal tasks



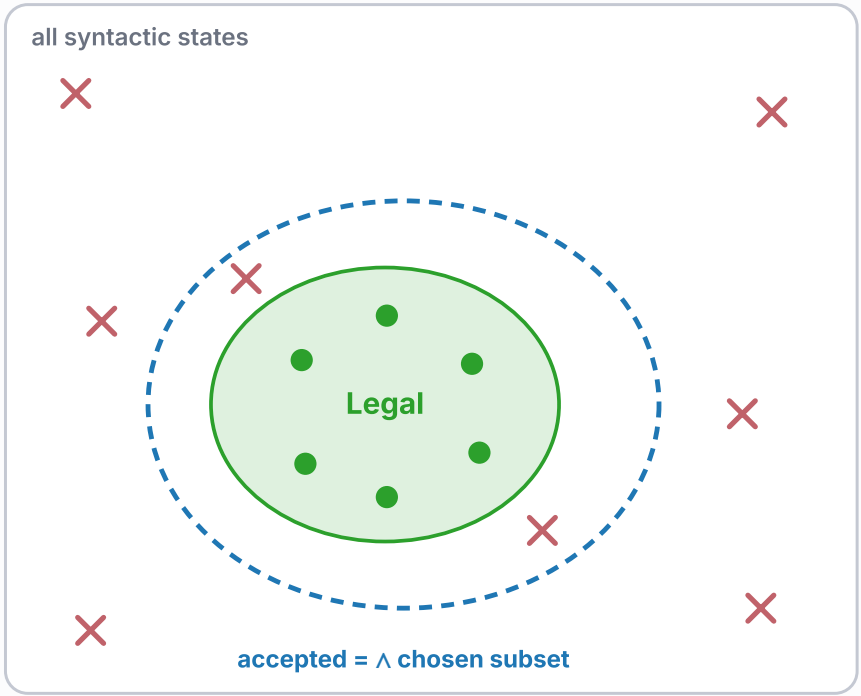
Best Candidate Constraint Subset

- The subset that:
  - Allows all legal tasks
  - Disallows the most illegal tasks



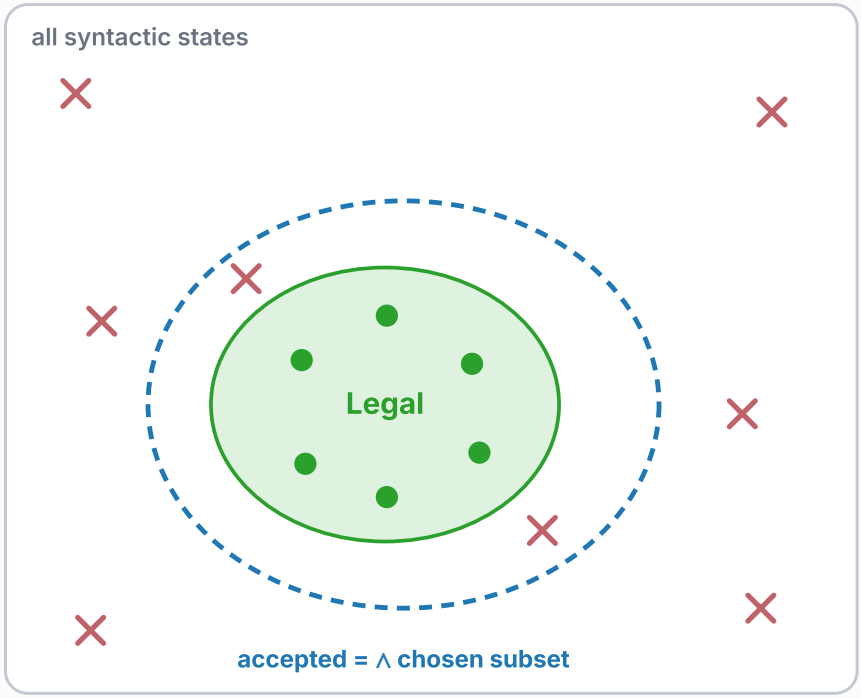
Best Candidate Constraint Subset

- The subset that:
  - Allows all legal tasks
  - Disallows the most illegal tasks
- Always exists



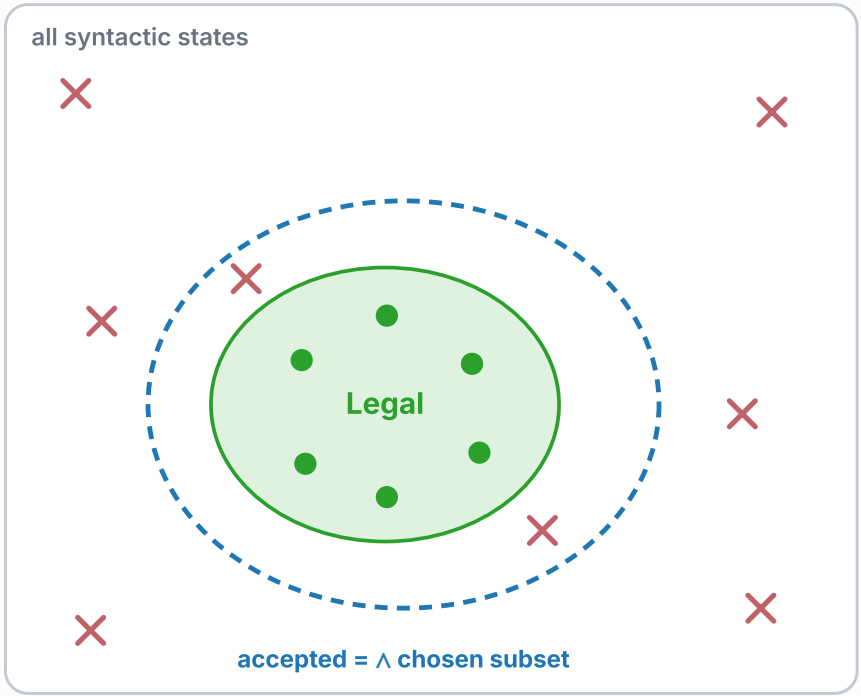
Best Candidate Constraint Subset

- The subset that:
  - Allows all legal tasks
  - Disallows the most illegal tasks
- Always exists
- Many equivalent ones



Best Candidate Constraint Subset

- The subset that:
  - Allows all legal tasks
  - Disallows the most illegal tasks
- Always exists
- Many equivalent ones
- Different candidates, different discrimination



The Exhaustive UDC and Motivation

[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

The Exhaustive UDC and Motivation

1. Initialize with all constraints

```
UDC(C, E)
Č ← C                                  # all candidates
```

[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

The Exhaustive UDC and Motivation

1. Initialize with all constraints
2. Keep the ones that always hold

```
UDC(C, E)
Č ← C                          # all candidates
for ⟨·, I, ·⟩ in E:            # each legal task
    for c in Č:
        if I ⊭ c:              # c fails here
            remove c from Č
```

[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

The Exhaustive UDC and Motivation

1. Initialize with all constraints
2. Keep the ones that always hold
3. Optionally reduce

```
UDC(C, E)
Č ← C                          # all candidates
for ⟨·, I, ·⟩ in E:            # each legal task
    for c in Č:
        if I ⊭ c:              # c fails here
            remove c from Č
Č ← reduce(Č)                  # optional
return Č
```

[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

The Exhaustive UDC and Motivation

1. Initialize with all constraints
2. Keep the ones that always hold
3. Optionally reduce
- PAC to best subset

```
UDC(C, E)
Č ← C                              # all candidates
for ⟨·, I, ·⟩ in E:                # each legal task
    for c in Č:
        if I ⊭ c:                  # c fails here
            remove c from Č
Č ← reduce(Č)                      # optional
return Č
```

[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

The Exhaustive UDC and Motivation

1. Initialize with all constraints
2. Keep the ones that always hold
3. Optionally reduce

- PAC to best subset
- Valiant's CNF algorithm[1]

```
UDC(C, E)
Č ← C                          # all candidates
for ⟨·, I, ·⟩ in E:            # each legal task
    for c in Č:
        if I ⊭ c:              # c fails here
            remove c from Č
Č ← reduce(Č)                  # optional
return Č
```

[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

The Exhaustive UDC and Motivation

1. Initialize with all constraints
2. Keep the ones that always hold
3. Optionally reduce

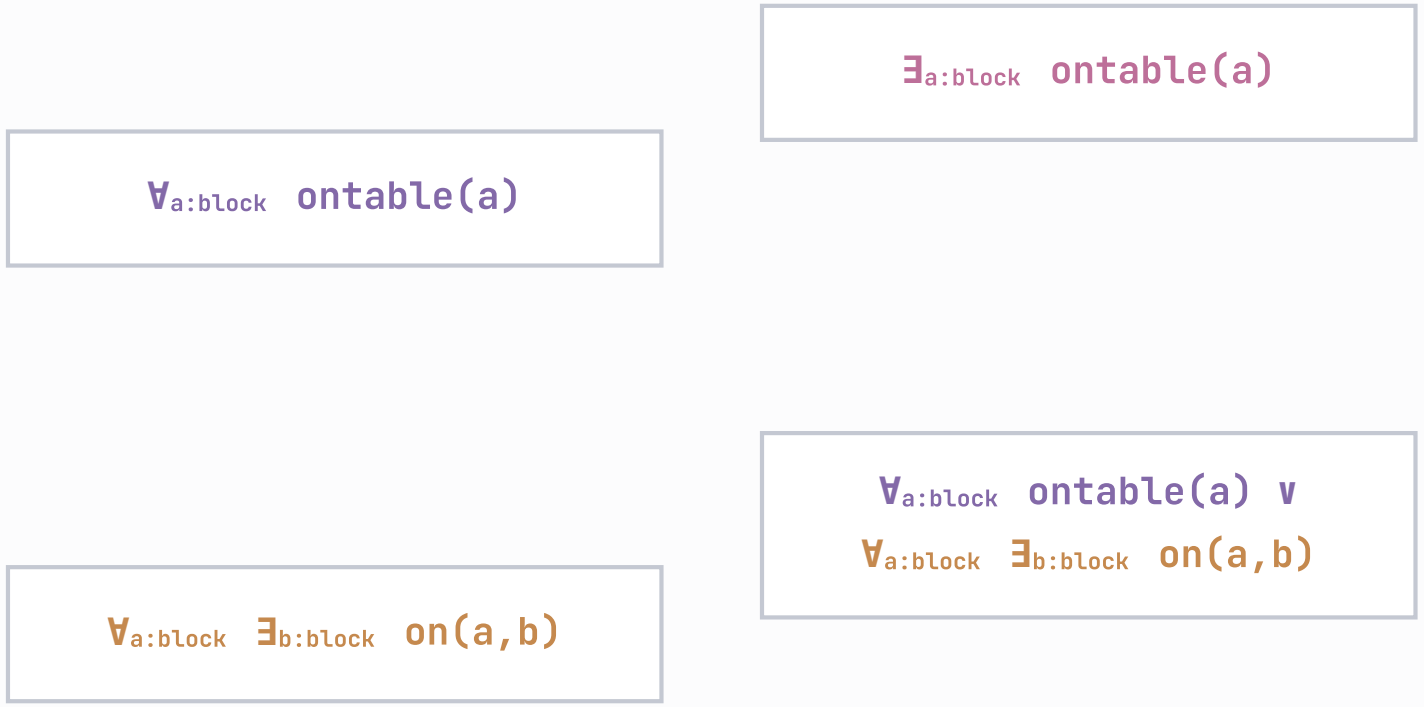
- PAC to best subset
- Valiant's CNF algorithm[1]
- Inefficient!

```
UDC(C, E)
Č ← C                          # all candidates
for ⟨·, I, ·⟩ in E:            # each legal task
    for c in Č:
        if I ⊭ c:              # c fails here
            remove c from Č
Č ← reduce(Č)                  # optional
return Č
```

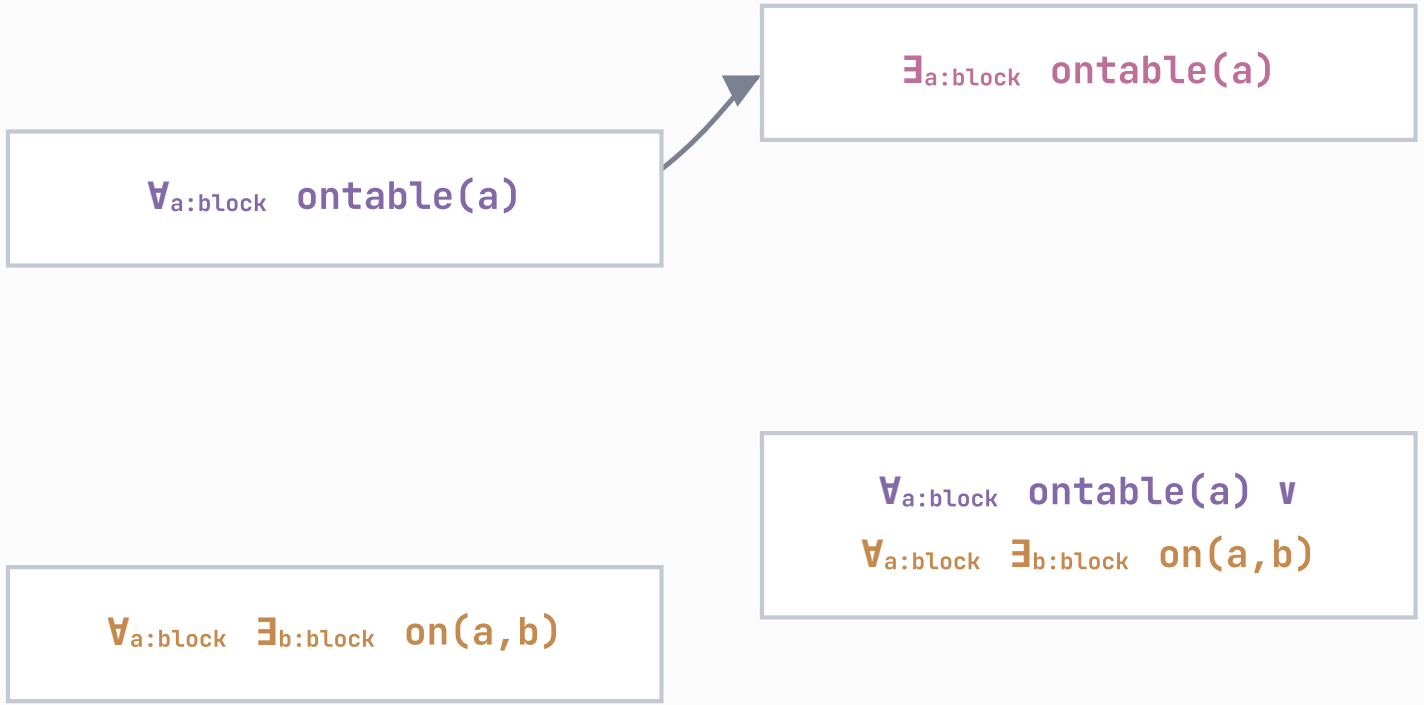
[1] Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.

Implication DAG

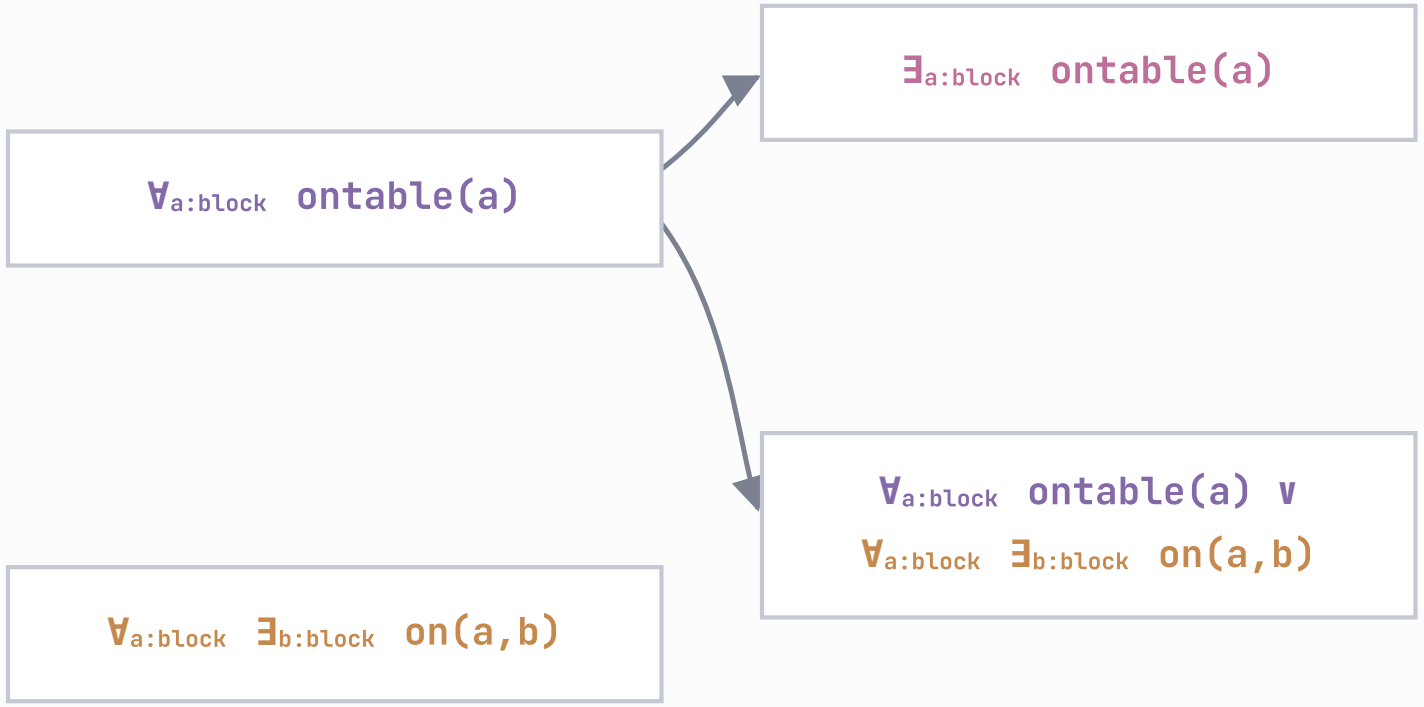
Implication DAG



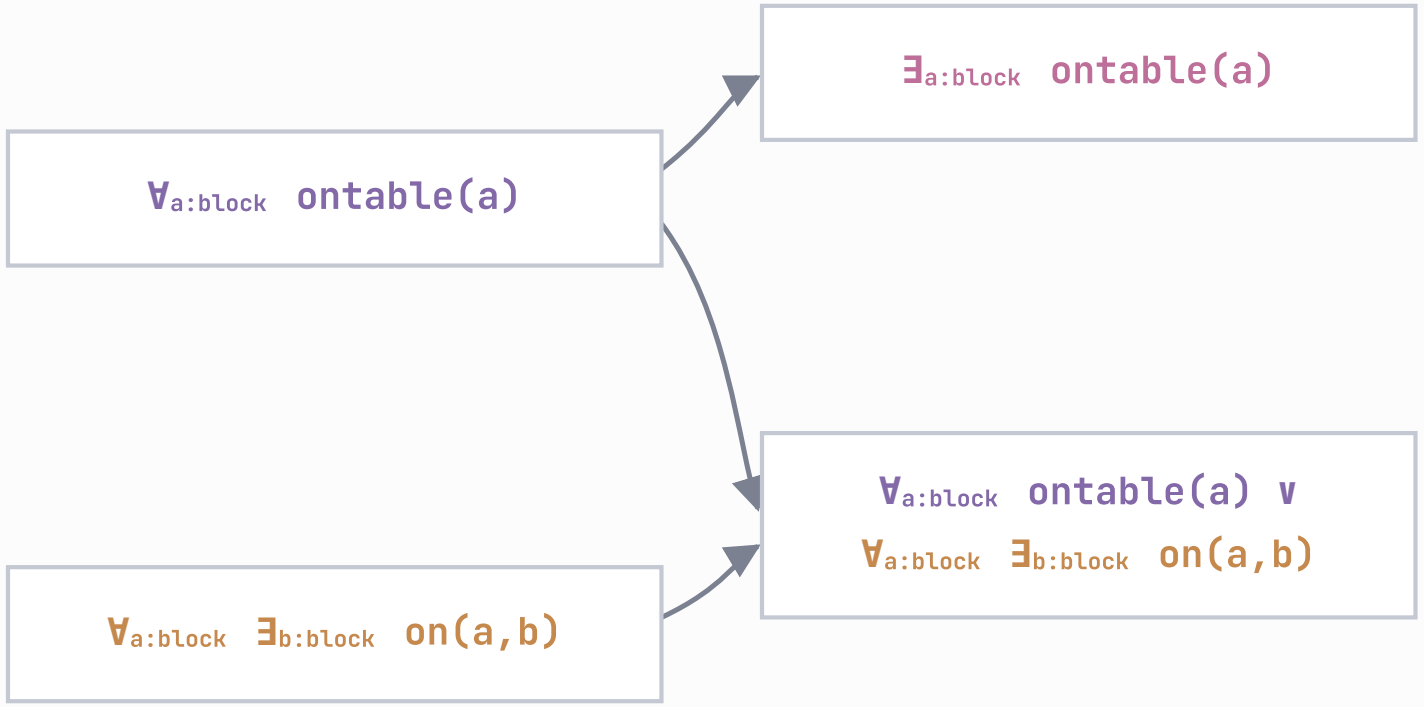
Implication DAG



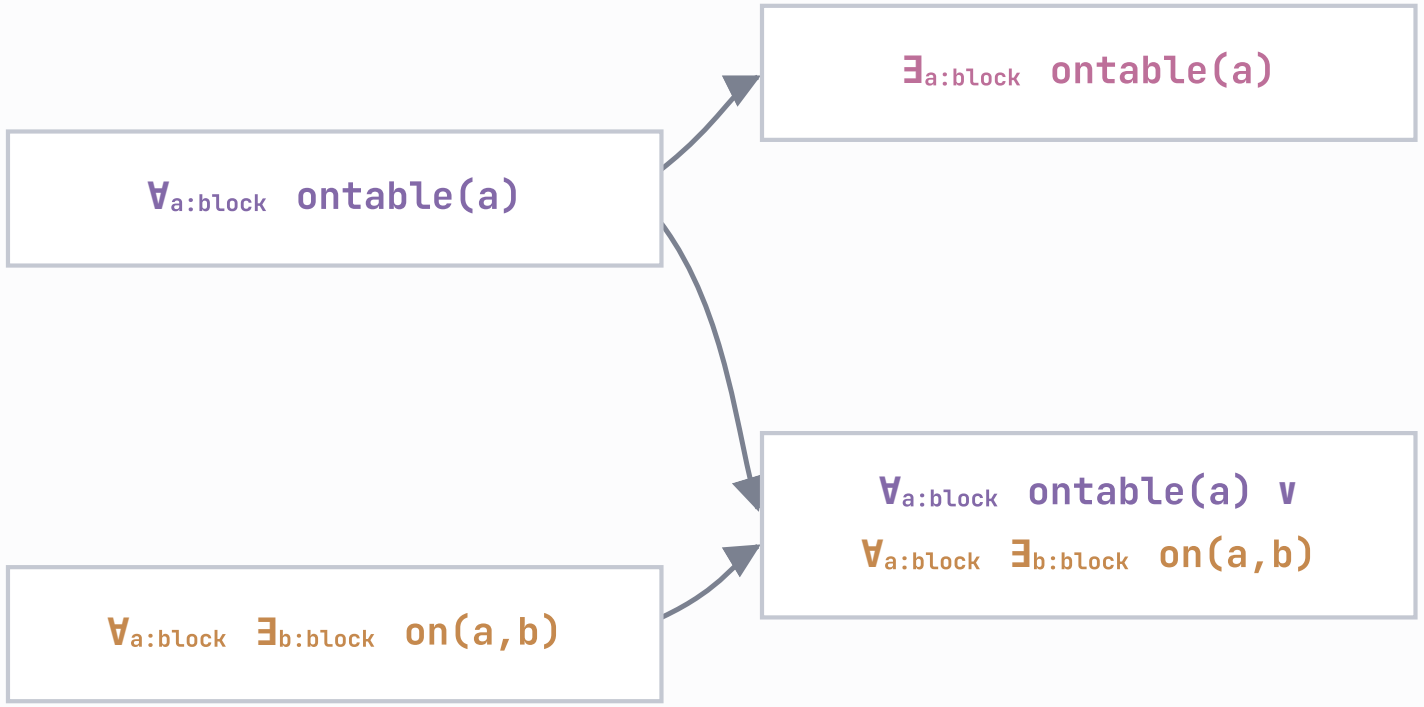
Implication DAG



Implication DAG

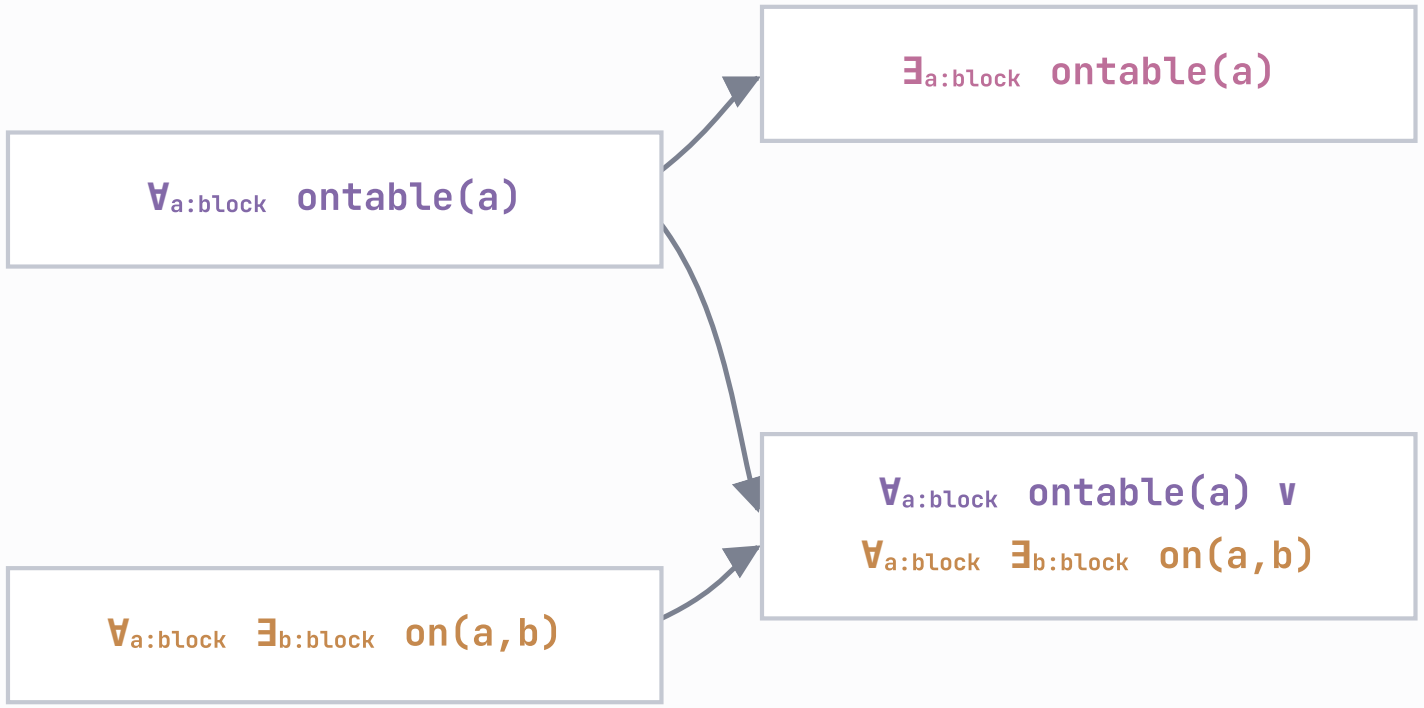


Implication DAG



Roots and weakening functions

Implication DAG



Roots and weakening functions

Common implicit invariant definition

The UDC Algorithm

The UDC Algorithm

1. Initialize with the DAG-roots

```
UDC(C, E, G)
Č ← roots(G)
```

The UDC Algorithm

1. Initialize with the DAG-roots
2. Sometimes weaken failures

```
UDC(C, E, G)
Č ← roots(G)
for ⟨·, I, ·⟩ in E:
    for c in Č:
        if I ⊭ c:
            Č ← Č \ c
            W ← maybe-weaken(c)
            Č ← Č ∪ W
```

The UDC Algorithm

1. Initialize with the DAG-roots
2. Sometimes weaken failures
3. Optionally reduce

```
UDC(C, E, G)
Č ← roots(G)
for ⟨·, I, ·⟩ in E:
    for c in Č:
        if I ⊭ c:
            Č ← Č \ c
            W ← maybe-weaken(c)
            Č ← Č ∪ W
Č ← reduce(Č)
return Č
```

The UDC Algorithm

1. Initialize with the DAG-roots
2. Sometimes weaken failures
3. Optionally reduce

- Same guarantees

```
UDC(C, E, G)
Č ← roots(G)
for ⟨·, I, ·⟩ in E:
    for c in Č:
        if I ⊭ c:
            Č ← Č \ c
            W ← maybe-weaken(c)
            Č ← Č ∪ W
Č ← reduce(Č)
return Č
```

The UDC Algorithm

1. Initialize with the DAG-roots
2. Sometimes weaken failures
3. Optionally reduce

- Same guarantees
- More efficient

```
UDC(C, E, G)
Č ← roots(G)
for ⟨·, I, ·⟩ in E:
    for c in Č:
        if I ⊭ c:
            Č ← Č \ c
            W ← maybe-weaken(c)
            Č ← Č ∪ W
Č ← reduce(Č)
return Č
```

The UDC Algorithm

1. Initialize with the DAG-roots
2. Sometimes weaken failures
3. Optionally reduce

- Same guarantees
- More efficient
- Matches existing works

```
UDC(C, E, G)
Č ← roots(G)
for ⟨·, I, ·⟩ in E:
    for c in Č:
        if I ⊭ c:
            Č ← Č \ c
            W ← maybe-weaken(c)
            Č ← Č ∪ W
Č ← reduce(Č)
return Č
```

When to weaken?

When to weaken?

- Exhaustive: always evaluate everything

When to weaken?

- Exhaustive: always evaluate everything
- Eager: weaken when *one* parent fails

```
#eval(Eager) ≤ #eval(Exhaustive)
```

When to weaken?

- Exhaustive: always evaluate everything
- Eager: weaken when *one* parent fails
- Lazy: weaken when *all* parents fail

```
#eval(Lazy) ≤ #eval(Eager) ≤ #eval(Exhaustive)
```

When to weaken?

- Exhaustive: always evaluate everything
- Eager: weaken when *one* parent fails
- Lazy: weaken when *all* parents fail
- All invariant work eager (or exhaustive)

```
#eval(Lazy) ≤ #eval(Eager) ≤ #eval(Exhaustive)
```

When to weaken?

- Exhaustive: always evaluate everything
- Eager: weaken when *one* parent fails
- Lazy: weaken when *all* parents fail
- All invariant work eager (or exhaustive)
- K-Step Lookahead: weaken when all *known* parents fail

```
#eval(Lazy) ≤ #eval(K-Step) ≤ #eval(Eager) ≤ #eval(Exhaustive)
```

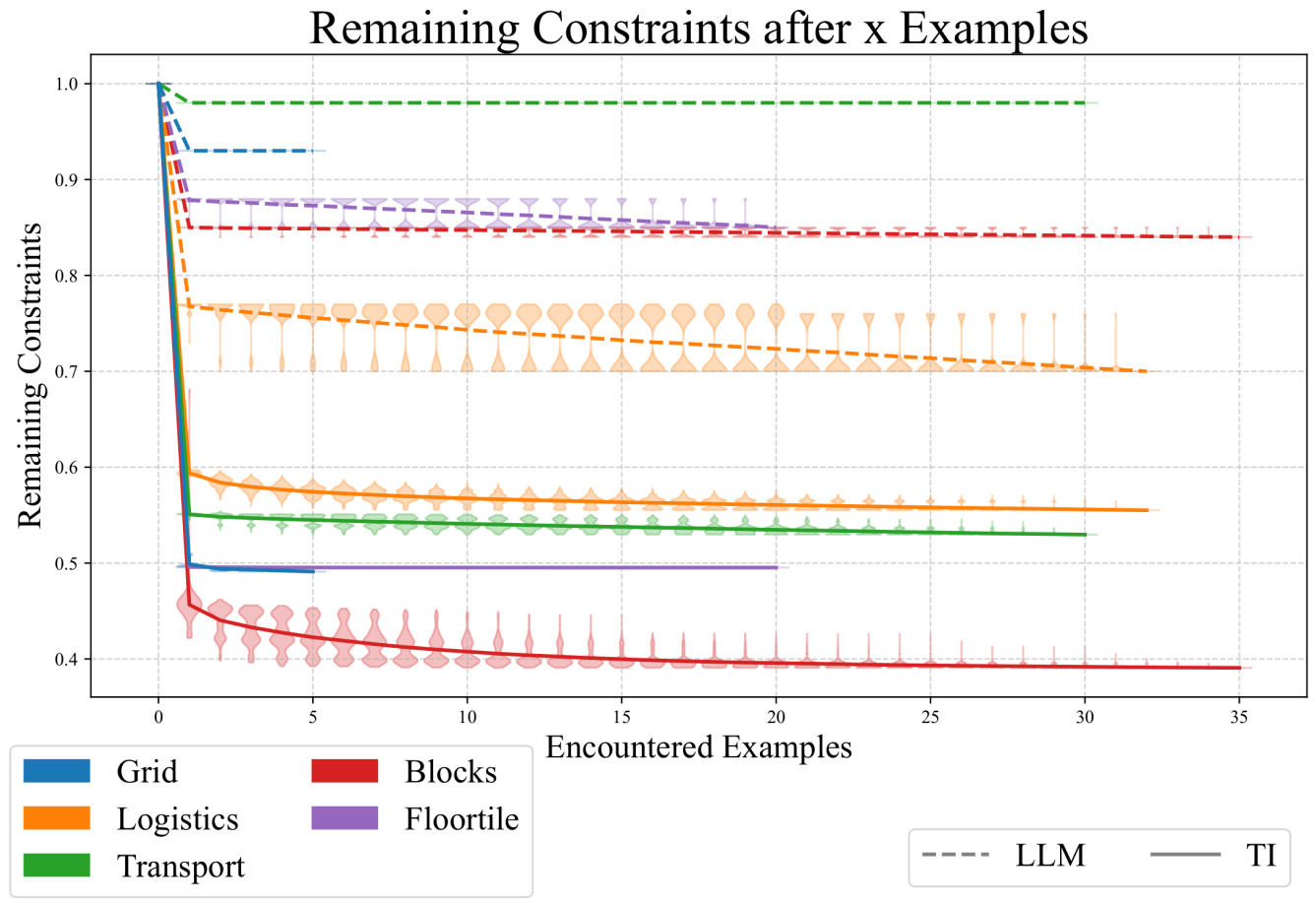
When to weaken?

- Exhaustive: always evaluate everything
- Eager: weaken when *one* parent fails
- Lazy: weaken when *all* parents fail
- All invariant work eager (or exhaustive)
- K-Step Lookahead: weaken when all *known* parents fail
  - Configurable memory–eval

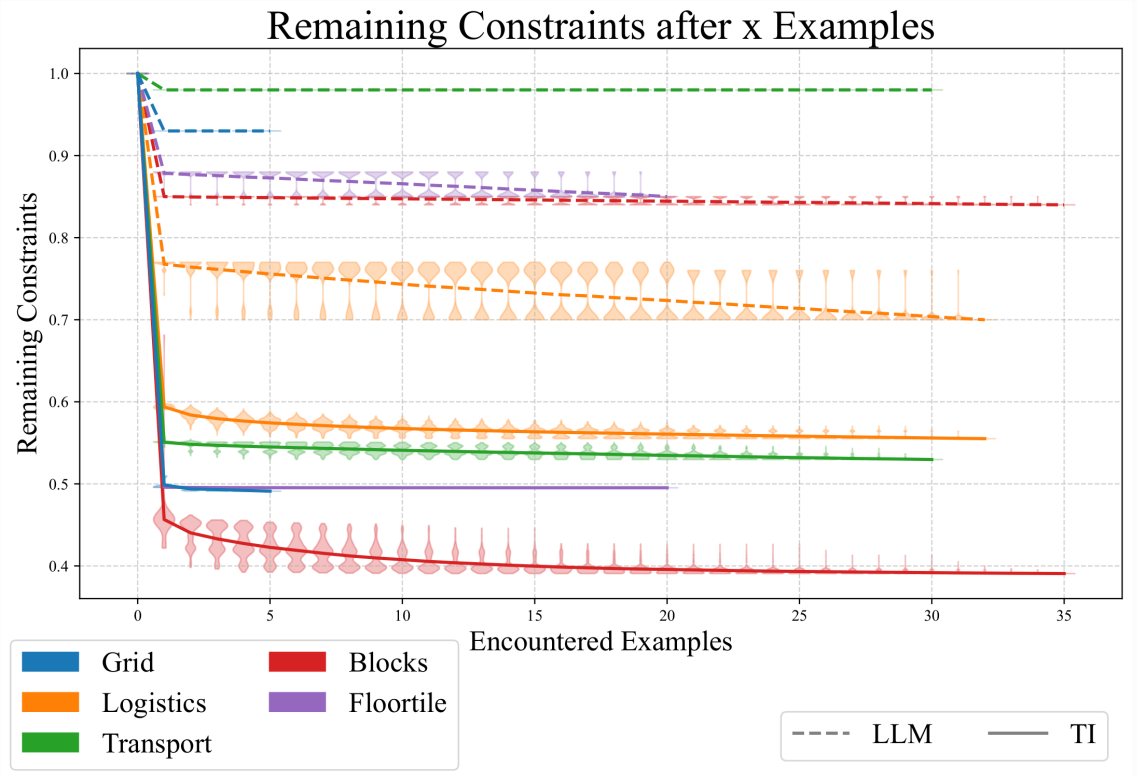
```
#eval(Lazy) ≤ #eval(K-Step) ≤ #eval(Eager) ≤ #eval(Exhaustive)
```

Experiments: Quantitative

Experiments: Quantitative

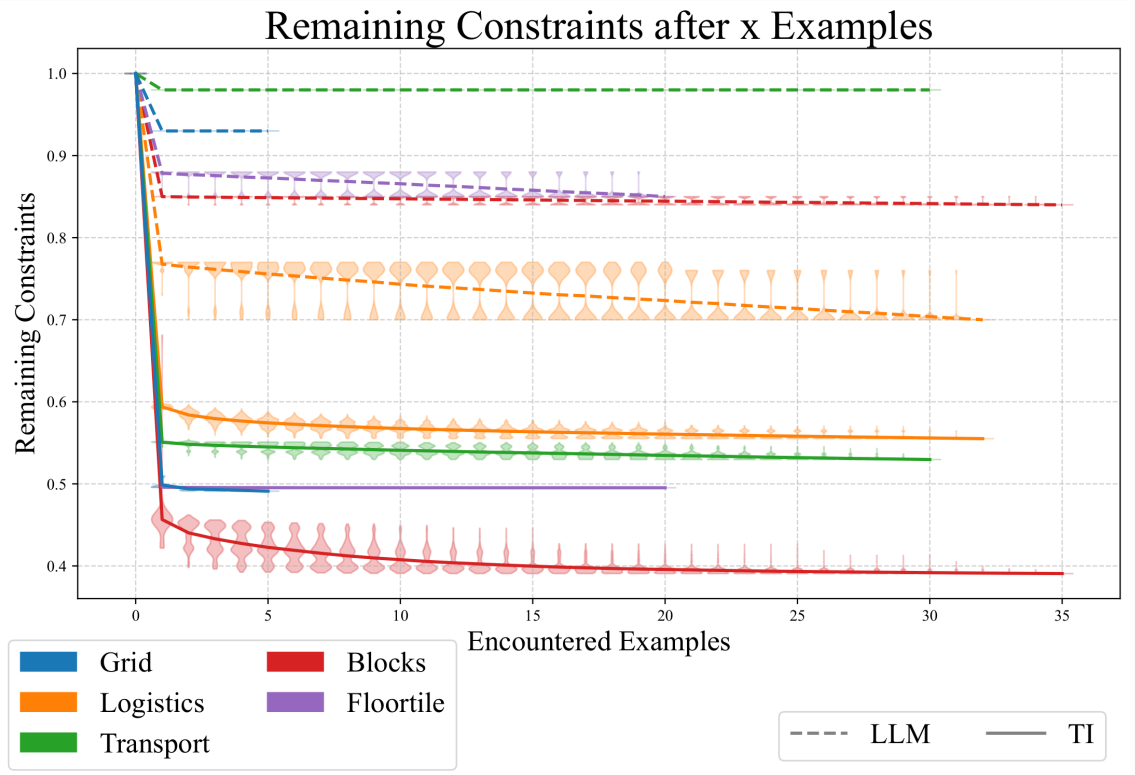


Experiments: Qualitative



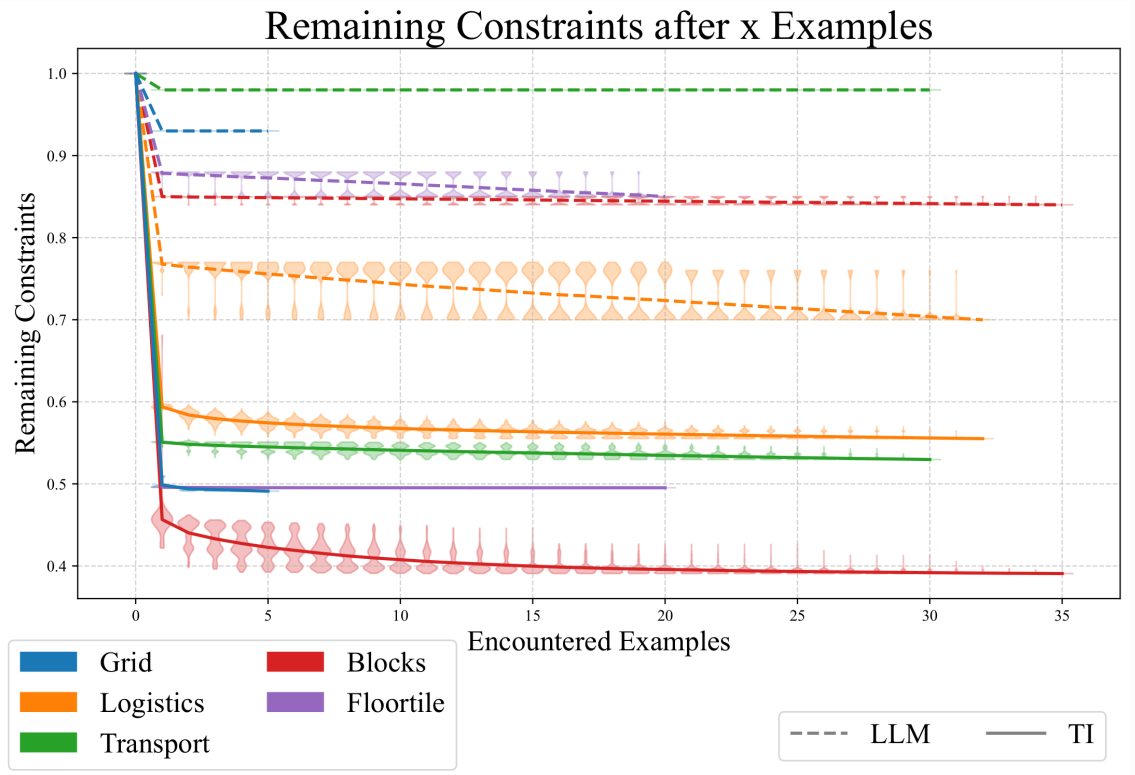
Experiments: Qualitative

- Qualitatively good



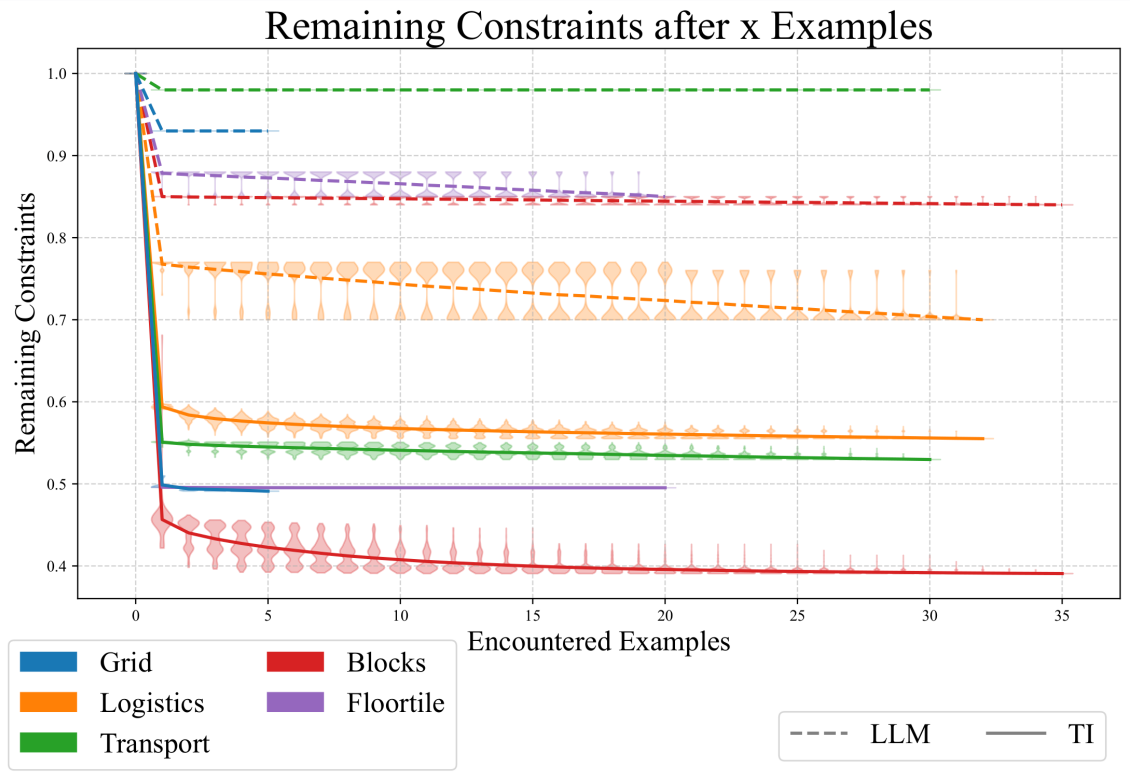
Experiments: Qualitative

- Qualitatively good
- Perfect initial Blocksworld



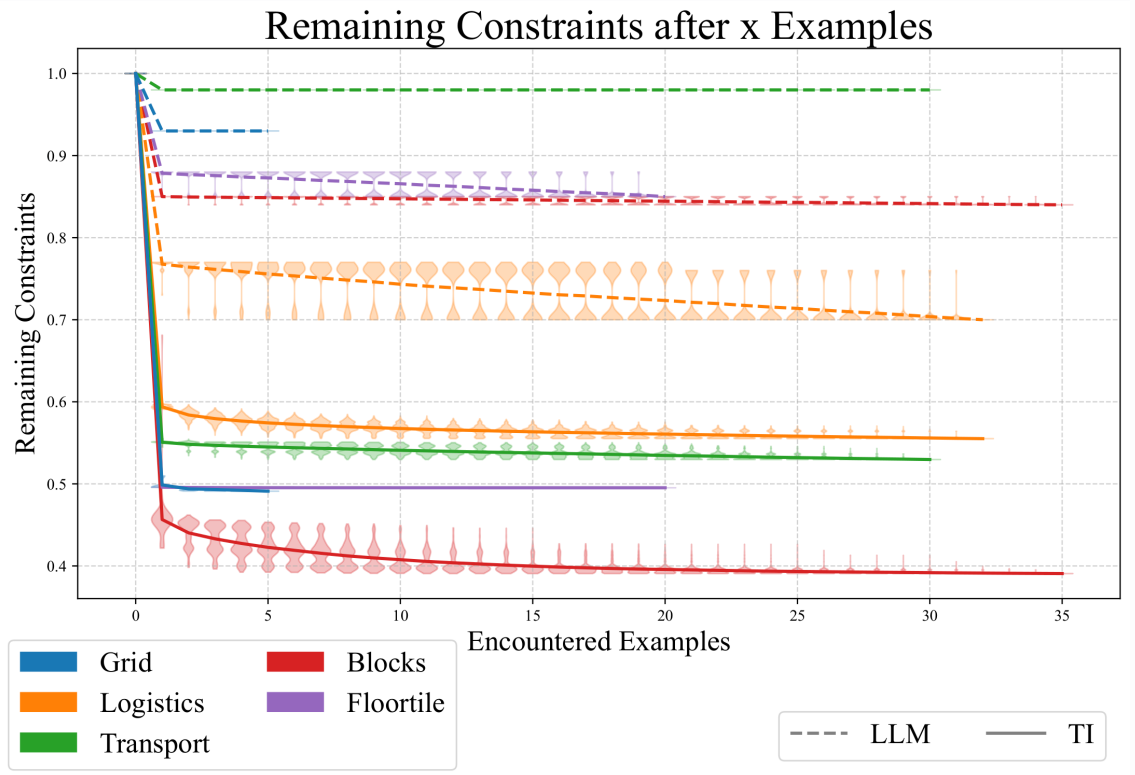
Experiments: Qualitative

- Qualitatively good
- Perfect initial Blocksworld
- LLM is more general



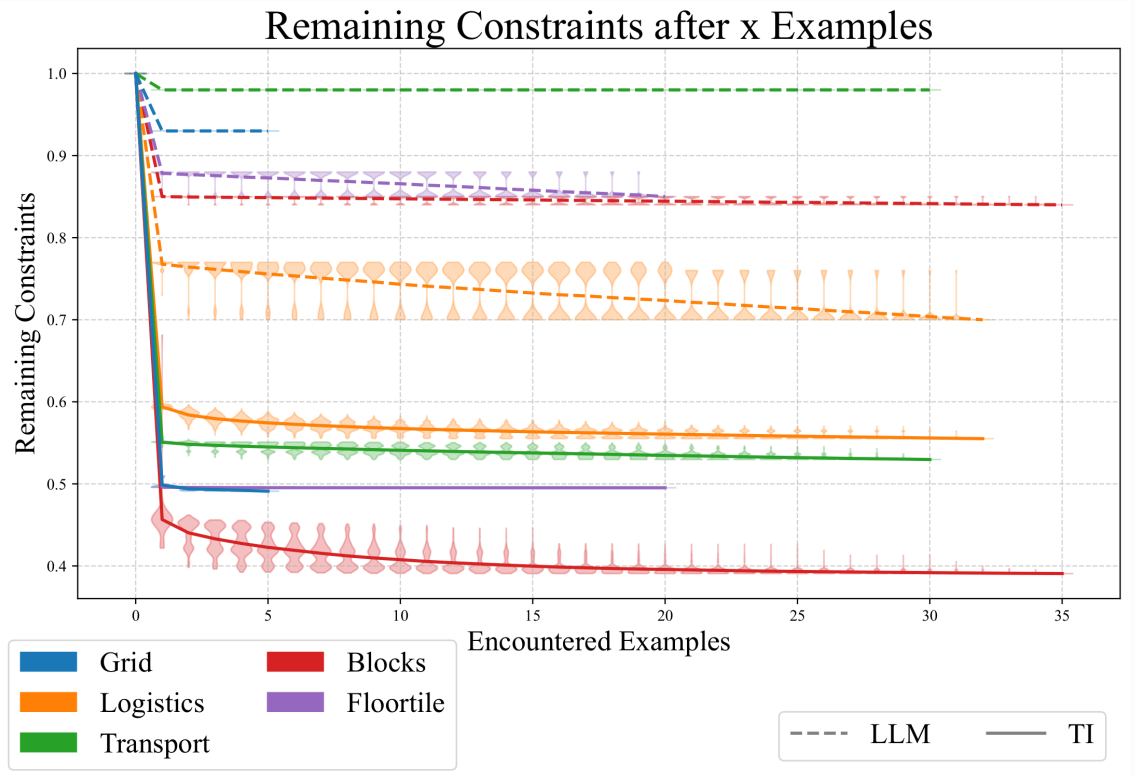
Experiments: Qualitative

- Qualitatively good
- Perfect initial Blocksworld
- LLM is more general
- Some overfitting



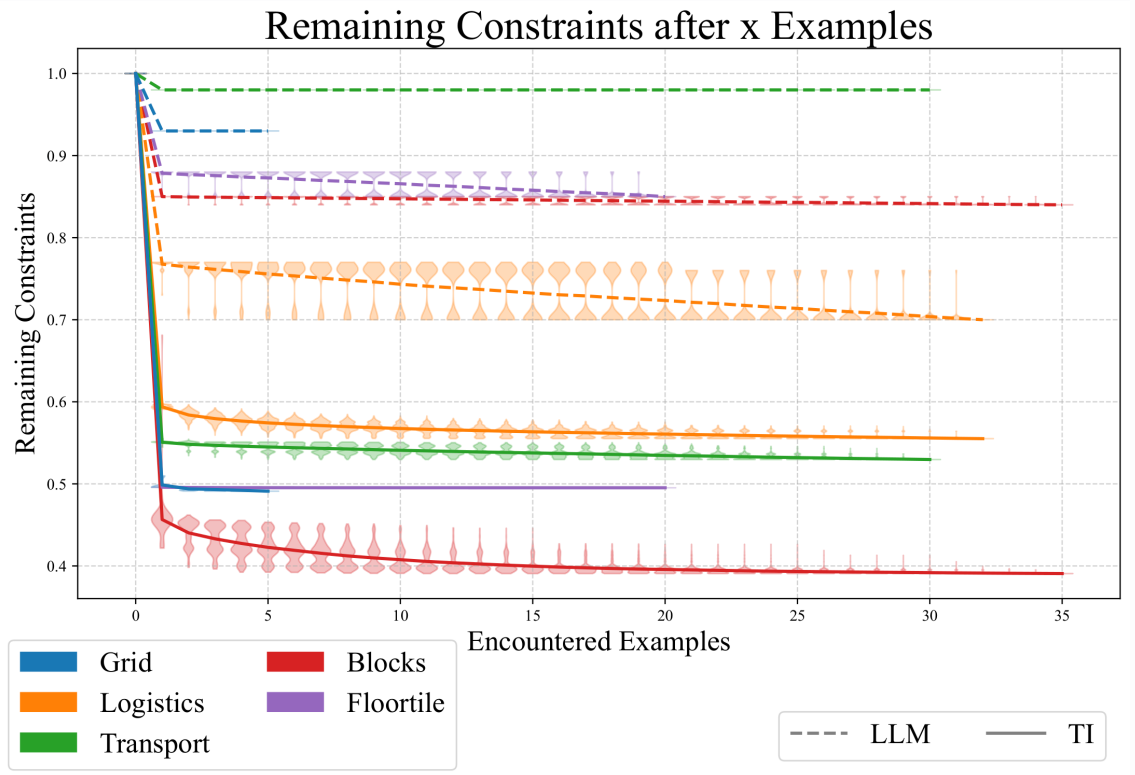
Experiments: Qualitative

- Qualitatively good
- Perfect initial Blocksworld
- LLM is more general
- Some overfitting
  - Floortile robot can't start top row



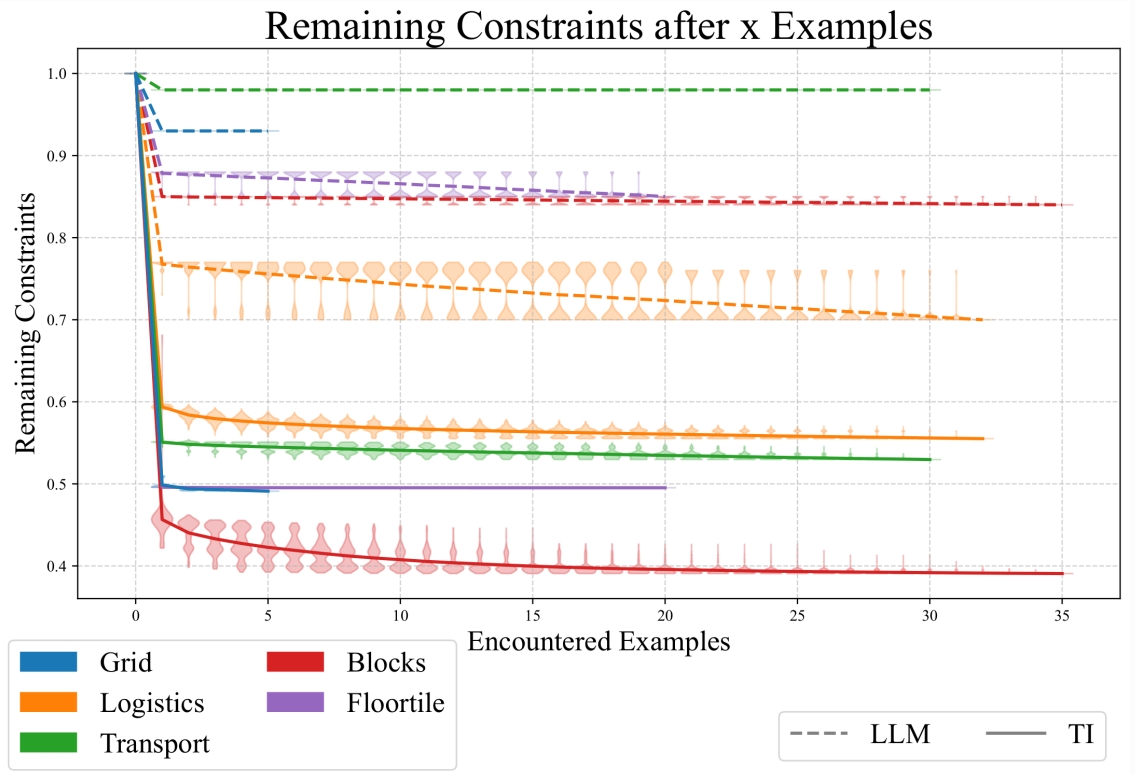
Experiments: Qualitative

- Qualitatively good
- Perfect initial Blocksworld
- LLM is more general
- Some overfitting
  - Floortile robot can't start top row
- TI “Debugging”:



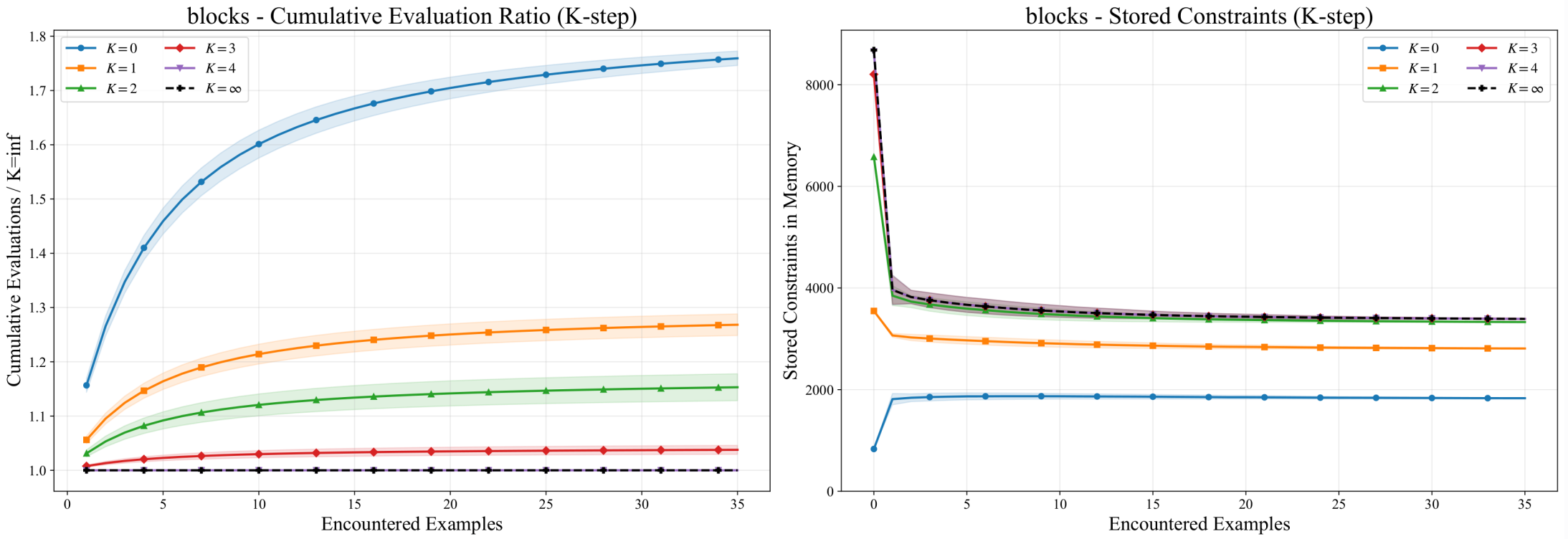
Experiments: Qualitative

- Qualitatively good
- Perfect initial Blocksworld
- LLM is more general
- Some overfitting
  - Floortile robot can't start top row
- TI “Debugging”:
  - Logistics: airplanes only *sometimes* start somewhere



Experiments: Lookahead

Experiments: Lookahead



Blocksworld · K = 0 (eager) … K = ∞ (lazy) · left: evaluations relative to lazy · right: constraints held in memory

Conclusions & Future Work

Conclusions & Future Work

CONCLUSIONS

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains
- Recover guarantees from any set

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains
- Recover guarantees from any set
- Widely applicable DAG-traversal methods

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains
- Recover guarantees from any set
- Widely applicable DAG-traversal methods

FUTURE WORK

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains
- Recover guarantees from any set
- Widely applicable DAG-traversal methods

FUTURE WORK

- Porting other invariants

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains
- Recover guarantees from any set
- Widely applicable DAG-traversal methods

FUTURE WORK

- Porting other invariants
- Quantitative quality

Conclusions & Future Work

CONCLUSIONS

- Automatically formalize domains
- Recover guarantees from any set
- Widely applicable DAG-traversal methods

FUTURE WORK

- Porting other invariants
- Quantitative quality
- K-step details

APPENDIX

Backup slides

Formal definition: Typed Implications

$$
\begin{aligned}
&\mathrm{TI}(X, Y) = \text{all constraints of the form} \\
&\quad \forall x_1 : \tau(x_1) \leq \tau(\mathrm{par}(X)_1) \ \cdots\ \forall x_n : \ldots \ \ X'(x_1, \ldots, x_n) \rightarrow \\
&\qquad Q_1 \tilde{y}_1 : \tau(\tilde{y}_1) \leq t_1 \wedge (\tilde{y}_1 \circ_1 S_1) \ \cdots\ Q_= \tilde{y}_= : \ldots \ \ Y'(y_1, \ldots, y_=) \\
&\quad \text{with} \quad X' \in \{X, \neg X\}, \quad Y' \in \{Y, \neg Y\} \\
&\qquad\qquad Q_i \in \{\forall, \exists\} \qquad \text{(universal / existential)} \\
&\qquad\qquad \circ_i \in \{\in, \notin\} \qquad \text{(restrict to / exclude X's args)} \\
&\qquad\qquad S_i \subseteq \{x_1, \ldots, x_n\} \qquad \text{(which antecedent args to tie)}
\end{aligned}
$$

- Single-predicate implications, $\top$ allowed as antecedent
- Extended predicate set: base $\cup$ goal-compiled $\cup$ transitive closures of arity-2 predicates
- Canonicalized: one representative per equivalence class, tautologies / unsatisfiable removed

Formal goal compilation

```
# compile a goal conjunction into the initial state,
# so one framework learns over initial and goal states alike

T' ≡ T    𝔸' ≡ 𝔸    𝒪' ≡ 𝒪    𝒢' ≡ 𝒢
𝒫' ≡ 𝒫 ∪ { p_g | p ∈ 𝒫 }                          # goal variant per predicate
ℐ' ≡ ℐ ∪ { p_g(ō) | ∃ ψ s.t. p(ō) ∧ ψ ≡ 𝒢 }       # add goal atoms
```

- For each goal atom $p(\bar{o})$, add a fresh atom $p_g(\bar{o})$ to the initial state
- Conjunctive goals: sound and complete
- Non-conjunctive goals: still sound, but incomplete (disjunctions not captured)

LLM prompt structure (C_{LLM})

- 1. Expertise priming
  - “a highly-skilled professor in AI planning and PDDL...”
- 2. Constraints + syntax description
  - first-order formulas over initial & (partial) goal predicates; `forall`, `exists`, `and/or/not/`$\rightarrow$, $\neq$
- 3. The PDDL domain
- 4. Smallest and largest example problems
  - two problems fit context; effective in prior work
- 5. Instruction: up to 10 derived predicates, exactly 100 candidates
  - structured output; GPT-5 · ~9.8k in / 55.7k out tokens · ~$0.57 per domain

Greedy reduction algorithm

```
# GreedyReduce(C): drop any constraint implied by the rest
C' ← C
for c in C':
    if Unsat( {¬c} ∪ (C' \ {c}) ):   # negation impossible ⇒ implied
        C' ← C' \ {c}
return C'
```

- Sound: removes only constraints implied by the others ($\wedge C' \leftrightarrow \wedge C$)
- Any sound reduction preserves all guarantees
- We use Z3 with a 30 s timeout, so the reduced set is an upper bound on minimal
- Optional: omitting it makes UDC additive (compose sets and examples)

Blocksworld: 22 dominant constraints (C_{TI})

#	Constraint	Meaning
B1	$\top \to \text{handempty}()$	hand starts empty
B2	$\top \to \neg\text{handempty}_g()$	goal never has the hand empty
B3	$\top \to \neg\exists a\ \text{clear}_g(a)$	goal never has a clear block
B4	$\top \to \neg\exists a\ \text{holding}(a)$	no block held at the start
B5	$\top \to \neg\exists a\ \text{holding}_g(a)$	goal never holds a block
B6	$\top \to \neg\exists a\ \text{ontable}_g(a)$	goal never has a block on the table
B7	$\top \to \exists a\ \exists b\ \text{on}_g(a,b)$	some block is on a block in the goal
B8	$\top \to \neg\forall b\ \exists a\ \text{on}_g(a,b)$	some block has nothing on it (goal)
B9	$\top \to \neg\forall a\ \exists b\ \text{on}_g(a,b)$	some block has nothing below it (goal)
B12	$\forall a\ \neg\text{clear}(a) \to \exists b\ \text{on}_{tc}(b,a)$	not clear ⇒ something is above it
B13	$\forall a\ \neg\text{ontable}(a) \to \exists c\ \text{on}_{tc}(a,c)$	not on table ⇒ above some block
B14	$\forall a,b\ \text{on}(a,b) \to \neg\exists_{d\neq b}\ \text{on}(a,d)$	a is on at most one block
B15	$\forall a,b\ \text{on}(a,b) \to \neg\exists_{c\neq a}\ \text{on}(c,b)$	at most one block on b
B19	$\forall a,b\ \text{on}_g(a,b) \to \neg\exists_{d\neq b}\ \text{on}_g(a,d)$	goal: a on at most one
B20	$\forall a,b\ \text{on}_g(a,b) \to \neg\exists_{c\neq a}\ \text{on}_g(c,b)$	goal: at most one on b
B21	$\forall a,b\ \text{on}_g(a,b) \to \neg\exists_{d\in\{a,b\}}\ \text{on}_g(b,d)$	goal: no height-2 cycles
B22	$\forall a,b\ \text{on}_{tc}(a,b) \to \neg\text{clear}(b)$	above(a,b) ⇒ b not clear
B23	$\forall a,b\ \text{on}_{tc}(a,b) \to \neg\text{ontable}(a)$	above(a,b) ⇒ a not on table
B26	$\forall a,b\ \text{on}_{tc}(a,b) \to \exists_{c\in\{a,b\}}\ \exists d\ \text{on}_g(c,d)$	≤ one goal-stack bottom
B27	$\forall a,b\ \text{on}_{tc}(a,b) \to \exists_{d\in\{a,b\}}\ \exists c\ \text{on}_g(c,d)$	≤ one goal-stack top
B30	$\forall a,b\ \text{on}_{tc}(a,b) \to \exists_{c\neq b}\ \exists d\ \text{on}_{tc}(c,d)$	no lone height-2 stack
B31	$\forall a,b\ \text{on}_{tc}(a,b) \to \neg\exists_{d\in\{a,b\}}\ \text{on}_{tc}(b,d)$	no cycles (above is acyclic)

22 constraints (hand-reduced from the 36 greedily-reduced) that together imply all others · on_{tc} = transitive closure of on (“above”) · subscript g = goal-compiled

Results: C_{TI} and C_{LLM} on IPC domains

					C_{TI}			C_{LLM}		
Domain	|T|	|P|	Arities	|E|	Cand.	Valid	Red.	Cand.	Valid	Red.
Blocksworld	1	5	0 / 1.0 / 2	35	8681	39.1%	36	100	84%	26
Logistics	9	3	2 / 2.0 / 2	32	19618	55.5%	36	100	70%	22
Transport	6	5	2 / 2.0 / 2	30	21358	53.0%	42	100	98%	32
Floortile	3	10	1 / 1.7 / 2	20	106358	49.5%	364	100	85%	35
Grid	1	12	0 / 1.2 / 2	5	111976	49.2%	174	100	93%	43

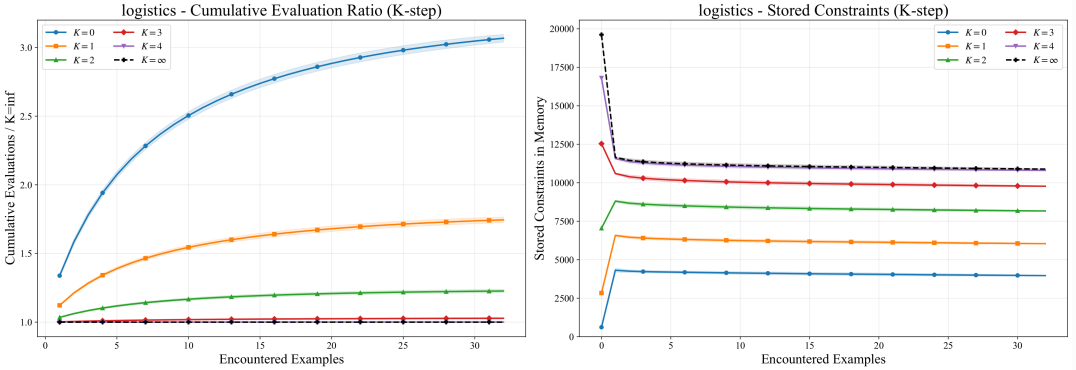
|T| types · |P| predicates · |E| examples · Valid = holds on all examples (% of candidates) · Red. = constraints after reduction

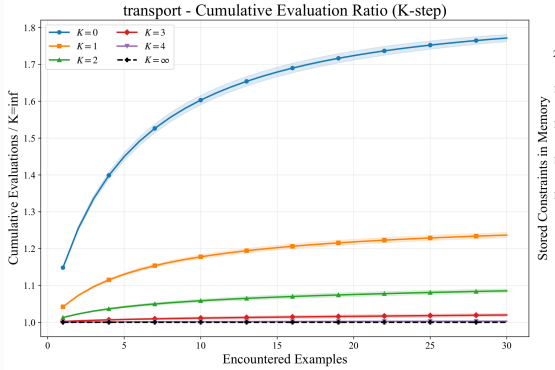
Runtime decomposition for C_{TI} (minutes)

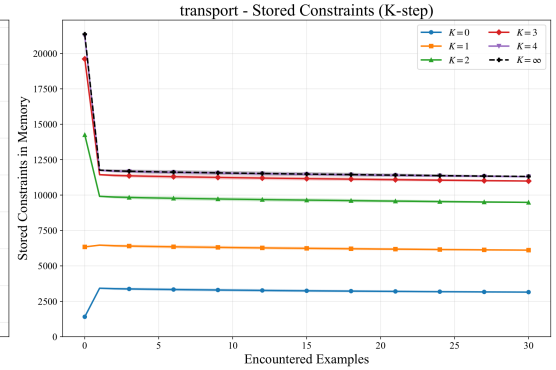
Domain	Generate	Evaluate	Sim. Exhaustive	Sim. Eager	Sim. Lazy	Reduce	Total
Blocksworld	2.86	0.67	0.15	0.20	0.16	22.57	26.61
Logistics	1.91	2.08	0.37	0.56	0.28	30.55	35.75
Transport	4.72	63.67	0.36	0.42	0.32	32.25	101.74
Floortile	29.57	332.85	1.19	3.94	2.62	262.84	633.01
Grid	63.68	803.25	0.04	0.31	0.29	189.35	1056.92

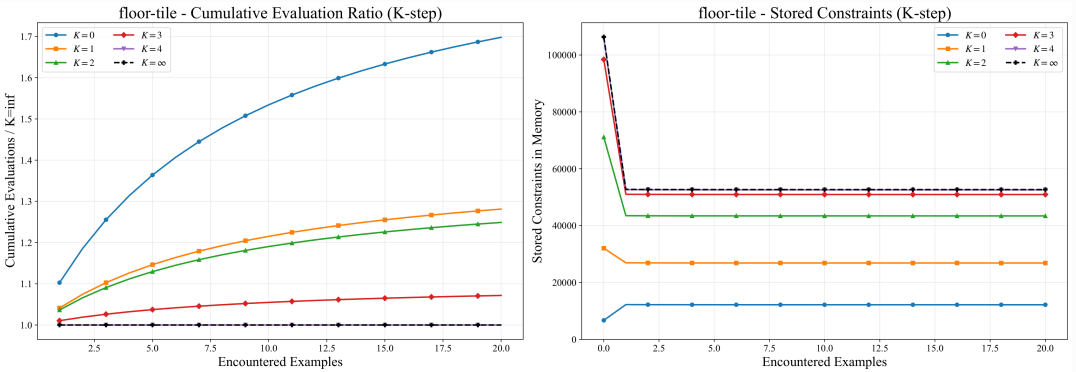
One full run per domain · Ryzen 7 7800X3D, 32 GiB · “Sim.” = simulated traversal over the same evaluated constraint-task outcomes

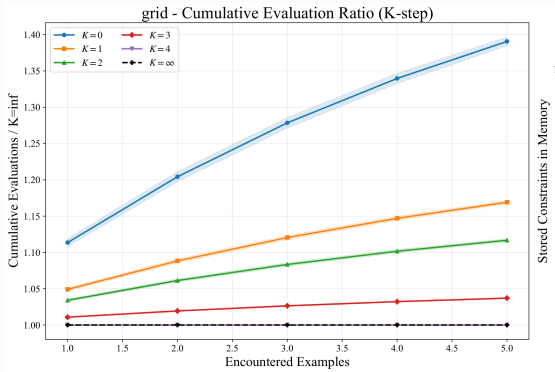
K-step lookahead: other domains

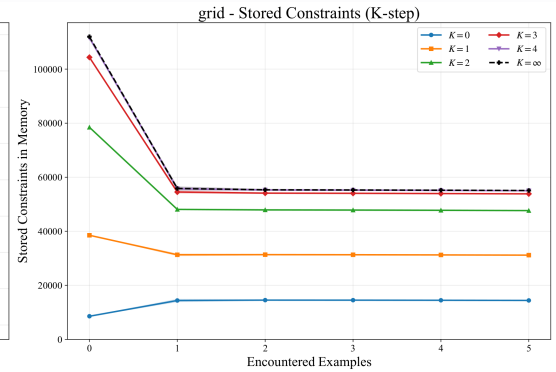












Cumulative evaluation ratio (vs lazy) and stored constraints, K = 0 (eager) … K = ∞ (lazy) · Logistics, Transport, Floortile, Grid

Traversal evaluation bound

$$1 \leq \frac{\#eval(K\text{-}Step)}{\#eval(Lazy)} \leq \frac{\#eval(Eager)}{\#eval(Lazy)} < |Candidates| - 1$$

- Eager evaluates 1× to nearly $|C|-1$ times as much as lazy
- K-step sits between lazy and eager, tunable via k
- Exhaustive is another 2.7–4.3× eager in the limit