

Learning How to Search for Plans with Exponentially Less Space

Dominik Drexler
Linköping University
Linköping, Sweden
dominik.drexler@liu.se

Markus Fritzsche
Linköping University
Linköping, Sweden
markus.fritzsche@liu.se

Simon Ståhlberg
RWTH Aachen University
Aachen, Germany
simon.stahlberg@gmail.com

Blai Bonet
Universidad Simón Bolívar
Caracas, Venezuela
bonetblai@gmail.com

Abstract

Heuristic search for a plan can store exponentially many states, even when its heuristic is almost perfect. We instead learn *search control*, one specification per domain, written as an indexical policy: a generalized policy with registers that hold objects and modes that sequence its rules. We add the *choose rule*, which loads an object into a register and marks a backtracking point, where one candidate suffices; every other rule must work for all of its outcomes and needs no search. Our main result is that structural termination, which rules out infinite executions, also bounds every execution by a polynomial in the number of objects. A depth-first procedure then finds a plan in polynomial space, however large the state space, with no list of visited states. The cost is time, exponential only in the *choice depth*, the number of real choices along an execution. Any class that such a policy solves therefore lies in NP, and in P at constant choice depth. We learn these policies with a language model in a counterexample-guided loop that certifies termination, verifies the training tasks, and keeps the choice depth small. With the learned policies, the procedure solves 1,709 of 1,890 test tasks of the IPC 2023 Learning Track and the Autoscale Agile suite, more than LAMA, BFWS, and Levitron, and most of them within one second and 100 MiB.

Keywords

Generalized Planning, Indexical Policies, Search, Policy Learning

1 Introduction

A classical planning task asks for a plan, a sequence of actions that transforms an initial state into a state that satisfies a goal. Heuristic search is the standard way to find one, but even an almost perfect heuristic can leave an exponential amount of search [28], and planners such as LAMA and BFWS store every state they generate, which can be exponentially many [35, 39].

This paper is about *search control*: a specification that tells a search procedure which paths in the state space are worth exploring. We seek search control for a fixed domain and every task over it, whatever its objects, initial state, and goal, which is the setting of generalized planning [30], and we express it as a policy of exactly that kind. What differs is the job: a generalized policy *decides* a plan, one action at a time and without search, whereas our search control *guides a search* for a plan.

Our rule language is that of indexical policies [7], which extend the Boolean and numerical features of standard policies with

registers that store objects selected during execution and with operating modes that regulate the control flow. In the Ferry domain, for example, a register holds the next car to be moved, while the modes cycle through phases for selecting, boarding, crossing, and unloading; a register thus names an object by its role, as in deictic representations [2, 12]. Such a policy has a strong solution concept: every transition that it allows must keep the goal reachable, so an agent may take any of them and never needs to backtrack [8, 9, 25]. Recent work learns such policies with language models and certifies that they are *structurally terminating*, which rules out infinite executions [3, 4]. Deciding without search has a price: such a policy evaluates cheap features along a short execution, so it decides plan existence in polynomial time, and it cannot solve all solvable tasks of an NP-hard class, unless $P = NP$ (Section 6.1).

We remove this obstacle with one new rule, the *choose rule*. An ordinary *load* rule stores an object in a register and, like every ordinary rule, must work for every object that it may store; a choose rule stores one too, but it is a backtracking point, where only one candidate has to work and the search finds it. In Sokoban, a legal push can leave a box permanently misplaced, so a load rule over two pushable boxes is wrong while a choose rule over the same two is correct: the second box can be tried once the first fails. Every other rule keeps its ordinary meaning, so a policy states, rule by rule, where search is needed and stays search-free everywhere else. A policy and a task thus define an AND/OR graph over augmented states: ordinary rules give AND vertices, which need every outcome to succeed, and choose rules give OR vertices, which need one. The policy solves the task when its initial vertex is solved, a least fixed point of this graph, and the *scout*, the procedure that executes the policy, is a depth-first search that tries the candidates of each choose rule in turn and backtracks on failure.

Structural termination makes the graph acyclic, even across all candidates of the choose rules, so the scout needs no set of visited states, yet the path that it stores could still be long. Our main result is that structural termination also bounds the length of every trajectory by a polynomial in the number of objects of the task. The scout therefore finds a plan in polynomial space, however large the state space is, whereas explicit state-space search may need exponential space. The price is paid in time, as the scout may try up to B^k sequences of choices, where the *choice depth* k counts the choices with more than one candidate along a trajectory and the *choice width* B bounds how many candidates each one has.

The same bound delimits which classes these policies can cover. A policy that solves a whole class turns the scout into a decision procedure, and since its plans have polynomial length, the class lies in NP, and in P if the choice depth is constant. Towers of Hanoi is thus out of reach, since its shortest plans have exponential length [10], and so is Sokoban, unless $\text{NP} = \text{PSPACE}$ [20].

We learn these policies in the counterexample-guided loop of previous work [3, 4], in which a language model proposes policies, an adversary proposes tasks on which they fail, and the model is asked to keep the choice depth small. We also extend the feature language with queries of relational algebra, and hence with predicates of any arity, which description-logic features cannot express, so previous work reformulated such domains [4], and with arithmetic on numerical features. We evaluate the learned policies on the IPC 2023 Learning Track test set and the Autoscale Agile suite. With them, the scout solves 1,709 of 1,890 test tasks, more than LAMA, BFWS, and Levitron, and most of them within one second and 100 MiB, without storing any visited states.

2 Related Work

Search control. Control knowledge has long been used to prune a planner’s search: TLPlan and TALPlanner check hand-written temporal-logic formulas along the paths of a forward search and discard every path that violates one [6, 33]. Hierarchical task network planning instead restricts the search to the decompositions that its methods allow [24, 38], and domain-specific solvers add knowledge of their own, as Rolling Stone does for Sokoban with deadlock tables [31]. In all of these, a standard procedure still carries out the pruned search, with its open and closed lists, so the space it needs can grow exponentially. Our policies instead define the search space itself, and the bound on its depth follows from a property of the policy checked once, before any task is seen.

Choice and AND/OR search. Golog and other agent programming languages provide nondeterministic choice with backtracking [34], and backtracking over variable values is the basic method for constraint satisfaction [21]. Strong solutions in nondeterministic planning are least fixed points of an AND/OR graph whose OR vertices are the actions of the agent and whose AND vertices are the outcomes that the environment picks [14]. Our search space has the same shape, but its AND vertices come from the policy rather than the environment: an ordinary rule must work for every transition it admits, which keeps the rest of the policy search-free.

Relational policies and generalized plans. Description-logic concepts and roles were introduced as a feature language for generalized policies [37] and later used for rule-based policies whose features are independent of a task’s objects [8, 9, 25]. Sketches use the same rules to decompose tasks into subproblems of bounded width [10, 22], and indexical policies add registers and operating modes [7]. General policies have been learned as logical generalized plans [29, 46], as programs with pointers over objects [41, 42], and as neural networks [48, 49, 54]. All of these select actions without search and so run into the limit of Section 6.1; related arguments show that, unless $\text{P} = \text{NP}$, Blocksworld has unbounded width [10] and no general optimal policy [48].

Language models for planning. Language models have been used to synthesize generalized plans and policies, often with automated feedback or soundness checks [13, 40, 44, 50, 51], and have been paired with planners or external validators [32, 36], since direct plan generation was long found unreliable [55]. Recent frontier models, however, rival state-of-the-art planners on tasks based on the IPC 2023 Learning Track [19]. They can also synthesize domain-dependent heuristics as Python code, which beat state-of-the-art domain-independent heuristics on test tasks [18]. Such a heuristic leaves every decision to best-first search, which can require exponentially many expansions even when its error is a small additive constant [28], whereas our policies mark the decisions that need search. Our learning loop follows the generator–adversary design of previous work [3, 4], a form of counterexample-guided synthesis [45] and of abstraction refinement [15, 43].

3 Background

We follow the notation of Francès et al. [25] and Bonet et al. [7].

A classical planning task is a pair $P = \langle D, I \rangle$ where the domain $D = \langle \mathcal{P}, \mathcal{A} \rangle$ contains predicates and action schemas, and I gives the objects O , the initial state s_0 , and the goal description $\mathcal{G}$. States are sets of ground atoms, applicable ground actions map states to successor states, and a plan is a sequence of ground actions that maps the initial state to a state that satisfies $\mathcal{G}$. A task is solvable iff there is a plan for it. A class of planning tasks is a set Q of tasks that share a domain but may vary in objects, initial state, and goal.

For features to compare the current state with the goal, the vocabulary is extended with *goal predicates* [3]. For each domain predicate p , the goal predicate p_g has the same arity, and for every tuple $\bar{o}$, the atom $p_g(\bar{o})$ holds in every state of P iff $\mathcal{G}$ contains $p(\bar{o})$. A second goal predicate records the atoms that the goal requires to be false. Every state thus also describes the goal.

Description-logic expressions over the predicates in the vocabulary denote sets of objects (*concepts*) and sets of object pairs (*roles*) [5]. They are built with the usual constructors. A *nominal* denotes a single object by name, such as the kitchen in Childsnack, which every task of the class contains. Emptiness tests over these sets yield Boolean features, and cardinalities yield numerical features. For example, in Blocksworld, the role of pairs in on_g but not in on relates each block to the block that it must be on but is not yet on, and its cardinality counts the unsatisfied goal atoms. Distance features measure the length of the shortest path between two concepts along a role, and they take the value ∞ if no such path exists. Numerical features take non-negative integer values that are bounded in each task, as the termination test below requires. These features refer only to unary and binary predicates.

3.1 Policies and Structural Termination

Let Φ be a vocabulary of concepts, roles, Boolean features, and numerical features, and let $\Psi = \Psi_B \cup \Psi_N$ be the sets of Boolean and numerical features in Φ . A (standard) rule over Φ has the form $C \mapsto E$, where C is the condition and E is the effect. The condition C is a set of expressions b and $\neg b$ for $b \in \Psi_B$, and $n = 0$ and $n > 0$ for $n \in \Psi_N$. The effect E is a set of expressions b , $\neg b$, and $b?$ for $b \in \Psi_B$, and $n \uparrow$, $n \downarrow$, and $n?$ for $n \in \Psi_N$. Features that E does not mention

must keep their values, while $b?$ and $n?$ allow arbitrary changes. A policy Π over Φ is a finite set of such rules.

A state transition (s, s') is *compatible with* (or *satisfies*) rule r if s satisfies its condition and the features change across (s, s') according to its effect. The transition is a Π -transition if it is compatible with a rule r in Π . A Π -trajectory is a sequence of states in which every consecutive pair is a Π -transition. The policy Π solves a task P if every maximal Π -trajectory that starts in the initial state is finite and ends in a goal state. Equivalently, Π is *closed* and *acyclic* in P : no maximal Π -trajectory that starts in the initial state ends in a non-goal state, and no such trajectory is infinite [25]. This strong solution concept permits an agent to take any Π -transition. The policy Π solves the class $\mathcal{Q}$ if it solves every solvable task in $\mathcal{Q}$.

Indexical policies [7] extend standard policies with a finite set M of (*operating*) *modes* and a finite set R of *registers*. The modes implement sequencing and contexts that activate or deactivate rules. The registers focus the policy on specific objects or object pairs. Rules of an indexical policy over Φ are of three sorts [3]:

- *transition rules* “ $m \parallel C \mapsto E \parallel m'$ ”, where $C \mapsto E$ is a standard rule over Φ and $m, m' \in M$ are modes;
- *load rules* “ $m \parallel C \mapsto \text{Load}(C, r) \parallel m'$ ”, where C is a feature condition, $C \in \Phi$ is a concept, and $r \in R$ is a register; and
- *do rules* “ $m \parallel C \mapsto (\text{op}(C_1, \dots, C_k), E) \parallel m'$ ”, where $C \mapsto E$ is a standard rule over Φ , op is the label of a k -ary action schema, and $C_1, \dots, C_k \in \Phi$ are concepts.

An *augmented state* is a tuple $\bar{s} = (s, m, v)$ of a planning state s , a mode m , and a register valuation v . We refer to its components as $\bar{s}_{\text{state}}$, $\bar{s}_{\text{mode}}$, and $\bar{s}_{\text{reg}}$. A transition rule changes the planning state according to its standard rule. A load rule keeps the planning state and stores one object from the denotation of C in r . A do rule applies a ground action with label op whose i -th argument belongs to the denotation of C_i , and whose transition satisfies $C \mapsto E$. All three sorts change the mode from m to m' . A pair $(\bar{s}, \bar{s}')$ of augmented states is *compatible* with a rule if the rule’s mode is $\bar{s}_{\text{mode}}$, its condition holds in $\bar{s}$, and the rule takes $\bar{s}$ to $\bar{s}'$ in one of these three ways [3]. We also say that the rule *admits* this transition. Load rules also have a role version, which stores a pair from the denotation of a role R in a role register. The vocabulary contains, for each register r , a concept or role whose denotation is the content of r , or the empty set if r is empty. Features may therefore change when a register changes, even if the planning state doesn’t.

An indexical policy is a tuple $\Pi = \langle M, R, \Phi, m_0, \mathcal{R} \rangle$, where m_0 is the initial mode and $\mathcal{R}$ is a finite set of indexical rules. Its executions are trajectories of augmented states that start in $\bar{s}_0 = (s_0, m_0, v_0)$, where v_0 leaves every register empty. As for standard policies, Π solves P if Π is closed and acyclic in P . Modules and call rules [7] are left for future work, as in previous work.

A policy is *structurally terminating* if it cannot induce infinite trajectories merely because of the way its features change, independently of their meaning [11, 22, 47]. For example, the policy Π made of just the two rules $m_0 \parallel \{\neg H, n > 0\} \mapsto \{H, n \downarrow\} \parallel m_0$ and $m_0 \parallel \{H\} \mapsto \{\neg H\} \parallel m_0$ is structurally terminating as every infinite Π -trajectory would flip the Boolean H infinitely often, and each flip involves a decrement of the non-negative integer n . Structural termination establishes acyclicity, one of the two conditions for a solution. Previous work [3] extends structural termination

to indexical policies. Its abstraction has one vertex for each pair of a mode and a qualitative valuation of the Boolean and numerical features, and it ignores the contents of the registers. A load rule into register r may change every feature that depends on r arbitrarily. Its hybrid verifier combines rule elimination with the complete Sieve procedure [47] and decides structural termination in this abstraction. A policy that it accepts is acyclic on every task.

4 Query and Arithmetic Features

Description-logic features refer only to unary and binary predicates. Many planning domains use predicates of higher arity. In Sokoban, for example, the ternary predicate $\text{adj}(l, l', d)$ states that location l' follows location l in direction d . Previous work compiles such predicates away before learning [4]. We instead extend the feature language with queries of relational algebra [1, 16].

A query denotes a relation over the objects of a task, with one named column for each position. Atomic queries denote the state atoms or the goal atoms of a predicate of any arity. Every concept and every role can also be used as a query with one or two columns. Queries are combined with natural join, projection, renaming of columns, selection by the equality of two columns or with a constant, union, and difference. The results connect back to the rest of the vocabulary in two ways. The projection of a query to one or two columns is a concept or a role, and the emptiness and the cardinality of a query are Boolean and numerical features. Evaluating a query takes polynomial time for a fixed query, since each intermediate relation has at most n^c tuples on a task with n objects, where c is its number of columns. A query with many columns therefore raises the degree of the bound in Section 6.

For example, with $\text{agent}(l)$ for the player’s location, $\text{at}(b, l)$ for the location of a box b , and $\text{clear}(l)$ for an empty location, the query

$$\pi_{b,l'}(\text{agent}(r) \bowtie \text{at}(b, l) \bowtie \text{adj}(r, l, d) \bowtie \text{adj}(l, l', d) \bowtie \text{clear}(l'))$$

denotes pairs (b, l') such that a push in some direction d moves b to l' once the player stands at the location r behind the box. Queries can express properties that relate more than two objects at once, which description-logic features cannot express directly [23].

Numerical features are extended with constants, addition, subtraction, multiplication, division, min, and max. Structural termination asks for a feature that some rule decreases and that no rule increases, and a sum can measure progress that no single count captures: in Spanner, adding the number of loose nuts to the number of locations still reachable from the objects gives a feature that decreases when the man picks up a spanner, walks on, or tightens a nut. A constant shifts the point at which a feature reaches zero, which matters because a condition compares a numerical feature only with zero: in Peg Solitaire, the number of occupied positions minus two is zero exactly when two pegs are left. Subtraction is truncated at zero and division rounds down, so numerical features keep non-negative integer values, as the termination test and Theorem 3 require. A value that exceeds a fixed limit, like the distance between two concepts that no path connects, is ∞ .

5 Search Spaces and Choose Rules

A policy, together with a task, defines a search space over augmented states. We first define choose rules and search spaces, and

then when a policy solves a task as a condition on the search space. We describe how the search is performed, illustrate choose rules in Sokoban and Peg Solitaire, and define the choice depth and width.

5.1 Search Spaces

A *choose rule* has the form “ $m \parallel C \mapsto \text{Choose}(C, r) \parallel m'$ ”. In an augmented state $\bar{s}$ in which the condition C holds, the *candidates* of the rule are the objects in the denotation of C in $\bar{s}$. Storing a candidate in the register r and changing the mode to m' gives a successor of $\bar{s}$, while the planning state stays the same. Unlike load rules, choose rules have no role version, because a role can have quadratically many object pairs as candidates. We call the indexical rules of Section 3 *ordinary*, in contrast to choose rules.

The search space $G_{\Pi, P}$ induced by policy Π on task P is the *bipartite and directed graph* made of augmented-state vertices $\bar{s}$, and rule vertices $(\bar{s}, r)$ where $\bar{s}$ is an augmented state, and r is a policy rule. A rule “ $m \parallel C \mapsto \dots \parallel m'$ ” is *applicable* in $\bar{s}$ if $m = \bar{s}_{\text{mode}}$ and C holds in $\bar{s}$. For an ordinary rule “ $r = m \parallel C \mapsto E \parallel m'$ ”, edge $(\bar{s}, (\bar{s}, r))$ exists iff there is at least one augmented state $\bar{s}'$ such that $(\bar{s}, \bar{s}')$ is compatible with r . For a choose rule “ $r = m \parallel C \mapsto \text{Choose}(C, r) \parallel m'$ ”, edge $(\bar{s}, (\bar{s}, r))$ exists iff r is applicable in $\bar{s}$, even if r has no candidate in $\bar{s}$. In both cases, edge $((\bar{s}, r), \bar{s}')$ exists iff $(\bar{s}, \bar{s}')$ is compatible with r . Its root is the initial augmented state $\bar{s}_0$.

The two sorts of rules differ here because they place different obligations on the policy. An applicable ordinary rule that admits no transition offers the search nothing, so the policy need not account for it. A choose rule instead marks a point where the search tries candidates, so it must offer a candidate that works. If it offers none, its vertex has no children and is unsolved, as defined next.

We make $G_{\Pi, P}$ an *AND/OR graph* by tagging each vertex as AND or OR: every augmented-state vertex $\bar{s}$ is an AND vertex, every rule vertex $(\bar{s}, r)$ for an ordinary rule r is an AND vertex, and every rule vertex $(\bar{s}, r)$ for a choose rule r is an OR vertex. Tagging every augmented-state vertex as an AND vertex makes the choice among rules universal: if several rules contribute children of $\bar{s}$, the policy must work for each of them, and an ordinary rule must work for each of its successors. The candidate of a choose rule is the only existential choice. This is not the convention of AND/OR search in nondeterministic planning, where the agent chooses the action at an OR vertex and the environment picks the outcome at an AND vertex [14]. Making the choices among rules universal instead is what lets an agent follow any transition that the policy allows.

The *solved* vertices are the least set of vertices that satisfies the conditions below. An augmented-state vertex $\bar{s}$ whose planning state $\bar{s}_{\text{state}}$ satisfies the goal is solved. An OR vertex is solved if at least one of its children is solved. Every other vertex is solved if it has at least one child and all its children are solved. The three conditions are monotone in the set of solved vertices, so the least such set exists, and it is computed as a least fixed point that starts from the goal states and works backwards. A vertex is therefore unsolved exactly when nothing in the graph makes it solved: a non-goal augmented state in which no rule admits a transition, an applicable choose rule without candidates, or a vertex whose only support runs through itself, as on a cycle.

Definition 1. Policy Π solves task P if the root $\bar{s}_0$ of $G_{\Pi, P}$ is solved. It solves a class $\mathcal{Q}$ if it solves every solvable task in $\mathcal{Q}$.

Definition 1 does not need structural termination, which only makes the search terminate. A cycle does not by itself make a choose rule unsolved: a choose rule one of whose candidates leads into a cycle can still be solved through another candidate.

Without choose rules, every rule vertex is an AND vertex, and Definition 1 says that every reachable non-goal augmented state has an applicable rule with a successor, and every successor of every applicable rule again leads to a solved state. These are the closed and acyclic conditions of Section 3 on the augmented states that Π reaches: a maximal trajectory can only end in a goal state, and no reachable cycle consists of non-goal states only, since without choose rules such a cycle is unsolved. Choose rules relax only the OR vertices, where a solution discards every candidate but one.

5.2 Backtracking and Search

We call the procedure that executes a policy the *scout*. It follows the policy from the initial augmented state $\bar{s}_0$, and it returns a plan if it reaches a goal state.

A choose rule turns an ordinary branch into a *backtracking point*: solving no longer needs every candidate to work, only one, and the scout finds that one by search.

In intractable domains, satisfying an AND vertex in place of a choose rule is not possible in general. A policy without choose rules must describe, with features alone, a set of candidates none of which leads to a *dead end*, a state from which the goal cannot be reached. The scout can then follow any allowed transition and never needs to backtrack. Policy features are evaluated in polynomial time, and Theorem 3 keeps the executions short. A policy without choose rules therefore decides in polynomial time whether a task has a plan, so it cannot cover a class for which this decision is NP-hard, unless $P = NP$ (Proposition 5). Planning is NP-hard for some benchmark domains [26], and Sokoban is even PSPACE-complete [20]. A choose rule removes this obstacle: only one of its candidates must avoid a dead end, and its OR vertex is solved once the scout finds it.

To state this formally, we fix the choices. A *choice function* σ selects, for every solved OR vertex, one solved child that supports it, that is, one that enters the least fixed point before it. Fixing σ at every solved OR vertex of $G_{\Pi, P}$, keeping every other AND vertex as it is, removing the children of goal states, since the search stops there, and keeping only the vertices that the root then reaches gives a subgraph $G_{\Pi, P}^\sigma$ in which every vertex has exactly the children it needs to stay solved. We call $G_{\Pi, P}^\sigma$ a *solution graph*: it exists exactly when the root $\bar{s}_0$ is solved, and every maximal trajectory in it is finite and ends in a goal state.

A solution graph tells us that a plan exists, but the scout does not know σ . Instead, it builds $G_{\Pi, P}$ on demand, one vertex at a time, and explores it with depth-first search. At an augmented-state vertex, it descends into any one of its children, since Definition 1 requires all of them to lead to the goal. It does the same at an AND rule vertex, and in neither case does it keep an alternative. At an OR rule vertex, that is, a choose rule, it tries the candidates one by one, in an arbitrary order, and if the rest of the execution fails, it backtracks to this vertex and tries the next candidate. The scout thus searches only at OR vertices.

The search terminates if the policy has no infinite trajectory, that is, if $G_{\Pi, P}$ is acyclic. Since a choose rule has the same denotation

and the same mode change as a load rule with the same arguments, the abstraction of Anonymous [3] can handle choose rules as well. We call a policy with choose rules *structurally terminating* if it is structurally terminating after replacing its choose rules by load rules. Such a policy induces an acyclic $G_{\Pi,P}$ on every task.

PROPOSITION 2. *Let Π be a structurally terminating indexical policy that solves a task P . Then the scout returns a plan for P .*

The converse does not hold. The scout may find a plan even if Π does not solve P , since it examines only one child of each AND vertex. Structural termination alone ensures only that the scout terminates, either with a plan or with a failure; whether it finds a plan depends on whether Π solves P .

EXAMPLE 1 (PEG SOLITAIRE). A policy that we learned for Peg Solitaire searches over jumps with two choose rules, and has five rules in all. Each jump removes one peg, and the tasks ask for one peg to be left, so the numerical feature r , the number of occupied positions minus two, is zero exactly when one jump remains. The concept S collects the occupied positions from which some jump is possible, and D the positions that a jump can reach from the position in the register s . Both are projections of a query over the ternary predicate *in-line*, which relates a position, the position jumped over, and the landing position. Writing P for the occupied positions, G for the positions that the goal requires to be occupied, L for the position visited last, C_s and C_d for the concepts of the two registers, and *jump* and *end* for the two action schemas, the rules are

$$\begin{aligned} m_1 \parallel \{r > 0\} &\mapsto \text{Choose}(S, s) \parallel m_2 \\ m_2 \parallel \emptyset &\mapsto \text{Choose}(D, d) \parallel m_3 \\ m_3 \parallel \{r > 0\} &\mapsto (\text{jump}(C_s, P, C_d), \{r \downarrow\}) \parallel m_4 \\ m_4 \parallel \emptyset &\mapsto (\text{end}(L), \emptyset) \parallel m_1 \\ m_1 \parallel \{r = 0\} &\mapsto (\text{jump}(P, P, G), \{r?\}) \parallel m_5. \end{aligned}$$

The first four rules form a loop: two choose rules select a jump, the third performs it, and the fourth ends the move. The third rule decreases r while the other three leave it unchanged, which is what certifies structural termination. Its middle argument is the broad concept P , yet the two registers already determine the position jumped over, so this AND vertex has a single child. The fifth rule performs the last jump, onto a position that the goal requires to be occupied, and stops in the mode m_5 . The two choose rules are needed because a legal jump can leave the remaining pegs unsolvable, and a candidate concept that excluded every such jump would have to decide solvability. Each iteration makes two choices, so an execution makes two choices per jump. $\square$

5.3 Choice Depth and Choice Width

For a finite trajectory τ , the *choice depth* $d(\tau)$ is the number of OR vertices on τ with at least two children. A choice with a single candidate introduces no alternative and does not count. The *choice width* $w(\tau)$ is the largest number of children of an OR vertex on τ , or zero if τ contains no choose rule. The choice depth and width, not the number of choose rules, determine how much the scout may have to search. If every trajectory has choice depth at most k and choice width at most B , the scout explores at most $\sum_{i=0}^k B^i$

sequences of choices. Since candidates are objects, the choice width is at most the number n of objects of the task, so the scout explores at most $\sum_{i=0}^k n^i$ sequences of choices. Small choice depth and width are thus natural objectives for learning.

6 Bounded Search Depth

Structurally terminating policies induce an acyclic search space $G_{\Pi,P}$ on every task. We now show that the length of every trajectory in $G_{\Pi,P}$ is also bounded. The bound exploits the fact that numerical features have domains of bounded cardinality.

Let κ be the maximum of 1 and the arities of the relations counted by the numerical features of the policy. On a task with n objects, a cardinality is at most $(n+1)^\kappa$, and a finite distance is below n . A numerical feature is a fixed expression combining such numbers with non-negative constants and the arithmetic operations of Section 4, and its values are non-negative integers or ∞ . If an expression contains m occurrences of cardinalities and distances, its finite values are at most $c(n+1)^{\kappa m}$ for a constant c that depends only on the expression. Addition and multiplication respectively add and multiply bounds, while truncated subtraction, integer division, minimum, and maximum cannot produce a finite value above the largest finite operand. Let D be the largest such κm over the numerical features, or 0 if there are none. There are integers $a \geq 1$ and $b \geq 0$, depending only on the policy, such that $U(n) = a(n+1)^D + b$ bounds all finite numerical feature values. Each numerical feature thus takes at most $p(n) = U(n) + 2$ values, including ∞ .

THEOREM 3. *Let Π be a structurally terminating indexical policy with choose rules. There is a polynomial q such that, for every task P with n objects, every trajectory of Π on P has at most $q(n)$ transitions.*

The polynomial $q(\cdot)$ depends only on Π , through its modes and its features. Its degree is $D|\Psi_N|$, so the bound is polynomial for a fixed policy, but not uniformly over all policies.

A trajectory of Π on P is a root-to-leaf path of $G_{\Pi,P}$ with its rule vertices removed, so a trajectory with ℓ transitions corresponds to a path of $G_{\Pi,P}$ with $2\ell + 1$ vertices, alternating augmented-state and rule vertices and ending at a state vertex. Theorem 3 therefore also bounds the depth of the search space itself: every path of $G_{\Pi,P}$ from the root has at most $2q(n) + 2$ vertices.

COROLLARY 4. *Let Π be a structurally terminating indexical policy with choose rules, and let P be a task with n objects that Π solves. Then a depth-first scout that builds $G_{\Pi,P}$ on demand and stores only the current path from the root and the index of the candidate it tries at each OR vertex on it, returns a plan for P in space polynomial in n .*

Explicit state-space search, such as breadth-first or greedy best-first search, keeps a set of visited states to avoid cycles and repeated work, and this set may grow exponentially in n . A structurally terminating policy induces an acyclic $G_{\Pi,P}$, so the scout needs no such set. It stores a single path through $G_{\Pi,P}$, whose length is polynomial by Theorem 3. Its running time can still be exponential in the choice depth. Theorem 3 also shows a limitation: a structurally terminating policy produces only plans of polynomial length, so it cannot solve classes that require longer plans.

6.1 Completeness

A policy that solves a class Q turns the scout into a decision procedure for plan existence on Q : the scout returns a plan for every task of Q that has one (Proposition 2), and it reports failure on all other tasks. Its space is polynomial in the number n of objects (Corollary 4), and its time is polynomial in n for each of the at most $\sum_{i=0}^k B^i$ sequences of choices that it explores. Since Theorem 3 bounds the length of every plan that the scout returns, it also limits which classes such a policy can solve.

PROPOSITION 5. *Let Π be a structurally terminating policy that solves class Q , and let q be the polynomial of Theorem 3. Every task of Q with n objects that has a plan has one with at most $q(n)$ actions, and deciding whether a task of Q has a plan is in NP. If the choice depth of Π is bounded by a constant on Q , this decision is in P.*

Three consequences follow. No structurally terminating policy solves Towers of Hanoi, where the shortest plan for d disks has $2^d - 1$ actions, since q is a polynomial. Unless $\text{NP} = \text{PSPACE}$, none solves Sokoban, which is PSPACE-complete [20], since plans of polynomial length would place it in NP. Unless $\text{P} = \text{NP}$, none with constant choice depth solves a class with NP-hard plan existence, as in several benchmark domains [26]. The choice depth of a policy thus measures how hard its class can be. A policy without choose rules has choice depth zero, so it can only solve classes in which plan existence is in P. Choose rules raise this ceiling from P to NP.

The converse of Proposition 5 is open. We do not know whether every class whose solvable tasks have polynomial-length plans admits a structurally terminating policy that solves it. Bounded action search does not settle this: structural termination needs progress on the features, not just a bound on the number of steps.

7 Learning Policies with Choose Rules

We learn policies in the counterexample-guided loop of previous work [3, 4], shown in Figure 1. Rather than enumerate a pool of features, a language model constructs features and rules on demand, while symbolic tools check each candidate policy and report why it fails. Two language-model agents drive the loop: a generator, which proposes and revises policies, and an adversary, which proposes tasks on which they fail. Both receive the PDDL domain, the source of its parameterized task generator, and the syntax and semantics of the feature and policy languages. The generator source matters because it defines the class of tasks: it shows which initial states and goals can occur, including invariants that the domain description alone does not imply, so a policy may rely on them.

Objectives. A policy is accepted only if it is structurally terminating and solves every task of the training set, and the prompt asks for a general policy, so that it also solves the tasks outside that set. Among the policies that qualify, we prefer those that search little, that is, those with a small choice depth. Replacing a choose rule by a load rule is not a way to get there, since a load rule is an AND vertex and therefore requires every candidate, not just one, to avoid a dead end. Instead, the generator can sharpen the candidate concept of a choose rule, add features, or use ordinary rules for phases in which no candidate leads to a dead end; a choice left with a single candidate no longer counts toward the choice depth. In Sokoban, for example, a feature can recognize a push that

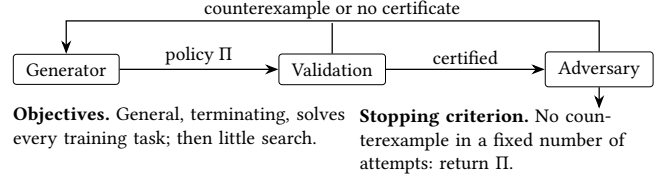


Figure 1: The learning loop. The generator proposes a policy Π and revises it using returned evidence. Validation checks structural termination, then every training task by execution and then universal verification. The adversary proposes task-generator settings and keeps solvable tasks on which Π fails.

moves a box into a corner that is not a goal location, from which the box can never be moved again, and the candidate concept of destinations can then exclude these corners. Only sharpening can cost a solution, since widening a candidate concept never breaks a policy that works. With lower priority, the prompt also asks for simple features and for control flow through modes.

Validation. Each proposal is checked first for structural termination, with choose rules treated like load rules, and only then on the training tasks, by execution and then by universal verification according to Definition 1. Execution needs to find only one plan, whereas verification confirms that every AND vertex of the search space has all its children solved and that every OR vertex has one solved candidate. Execution is cheaper and often finds failures early, so verification is skipped once it fails. The generator then receives the failure: a trajectory to where the policy gets stuck, a choose rule without candidates, a cycle, or the part of the abstraction that the termination test cannot certify, together with the search effort, choice depth, and choice width of successful searches. A missing termination certificate requires a revised policy, while a resource limit in a task check is inconclusive.

Adversary. The adversary proposes task-generator settings intended to expose weaknesses, for example the number of boxes and the size of the layout in Sokoban. A task enters the training set only if a planner finds a plan for it, which establishes solvability independently of the candidate policy, and only if the policy then fails on it or reaches a resource limit. Earlier tasks are rechecked before each new request, so that a revision which breaks one of them is detected. The adversary sees the outcome of every setting that it has tried, which helps it select tasks that challenge the policy while remaining small enough for the symbolic checks.

Returning a policy does not imply that every retained task has been verified, since checks that remain inconclusive are reported as such. Proposal and repair budgets bound the learning process, and exhausting them can end a run without a returned policy.

8 Experiments

We address three questions. Does the learning loop produce structurally terminating policies that use choose rules where plan existence is hard? Do the learned policies generalize to the test tasks, and how do they compare with classical planners in coverage, time, and memory? How much do the learned policies search, and what does storing the explored choice points gain?

Table 1: Learned policies for IPC-learning (top) and Autoscale (bottom). All indexical policies were proven structurally terminating. G and A count generator and adversary proposals; T gives total learning time (hours). P counts distinct learning tasks with proofs of correctness for the retained policy. F, M, R, and Ch count features, modes, rules, and choose rules. d and w give maximum successful choice depth and width during validation of the retained policy. USD estimates token cost at standard short-context API prices (1 October 2026). In each suite, policies in the lower block use choose rules.

Domain	G	A	P	T	F	M	R	Ch	d	w	USD
(a) IPC-learning											
Blocksworld	1	4	360	0.08	5	4	5	0	0	0	1.19
Childsnack	1	4	240	0.78	8	8	9	0	0	0	1.64
Ferry	1	4	338	0.38	6	5	6	0	0	0	1.30
Floortile	1	4	339	0.64	7	5	11	0	0	0	1.96
Miconic	1	4	360	0.35	4	4	8	0	0	0	1.07
Rovers	1	4	303	1.00	19	17	28	0	0	0	2.37
Satellite	1	4	300	1.43	8	7	10	0	0	0	1.97
Spanner	1	4	305	1.82	3	1	3	0	0	0	1.39
Transport	1	4	360	0.50	9	6	8	0	0	0	1.56
Sokoban	15	17	1331	11.05	16	13	18	4	7	7	48.22
(b) Autoscale											
Barman	1	4	350	1.02	8	16	17	0	0	0	1.51
Blocksworld	1	4	356	0.52	4	4	5	0	0	0	1.22
Childsnack	1	4	300	0.89	11	8	10	0	0	0	1.79
Data-network	1	4	276	1.99	11	8	10	0	0	0	2.03
Depots	1	4	330	1.19	9	7	8	0	0	0	1.84
Driverlog	1	4	331	1.12	17	22	31	0	0	0	2.05
Elevators	1	4	335	0.32	10	8	16	0	0	0	1.54
Floortile	1	4	333	0.83	9	7	12	0	0	0	2.38
Grid	1	4	239	1.23	9	11	16	0	0	0	2.06
Gripper	1	4	9	0.06	3	4	4	0	0	0	1.19
Hiking	2	5	404	1.62	9	16	20	0	0	0	3.25
Logistics	1	4	332	0.45	11	15	21	0	0	0	1.74
Miconic	1	4	300	0.32	4	4	8	0	0	0	1.32
Parking	1	4	298	0.64	16	15	29	0	0	0	2.24
Rovers	1	4	304	1.11	13	13	22	0	0	0	2.37
Satellite	1	4	306	0.47	11	8	12	0	0	0	1.65
Scanalyzer	1	4	9	0.27	18	15	25	0	0	0	2.33
Storage	1	4	277	2.03	13	5	9	0	0	0	2.46
Tidybot	6	10	560	11.25	32	38	88	0	0	0	28.66
TPP	1	4	354	0.34	11	8	10	0	0	0	1.59
Transport	1	4	338	0.65	9	6	8	0	0	0	1.46
Visitall	1	4	262	1.00	3	1	2	0	0	0	1.17
Zenotravel	1	4	320	0.46	9	9	14	0	0	0	1.79
Agricola	1	4	235	3.19	40	21	37	2	51	11	4.08
Freecell	12	13	984	4.80	30	21	57	9	1072	10	35.62
Mprime	13	15	848	4.87	36	27	43	10	29	10	46.26
Nomystery	1	4	326	1.24	8	5	9	2	11	5	2.45
Pegsol	1	4	335	0.94	8	5	5	2	29	16	1.32
Snake	11	12	852	6.10	21	20	61	8	689	24	29.78
Sokoban	20	10	892	9.56	64	125	182	50	35	14	72.56
Termes	1	4	335	0.88	13	10	23	3	29	7	2.90
Tetris	2	4	337	1.68	6	4	27	3	34	24	3.99
Woodworking	1	4	296	2.96	15	6	16	1	7	5	2.83

Table 2: Test results for IPC-learning (top) and Autoscale (bottom). Lev. denotes Levitron; IP stores no visited states; IP+Ch stores the explored choice points of the same policies. S counts solved tasks; shared d and w give maximum choice depth and width over tasks solved by both IP variants. Bold marks maximum coverage, including ties. In each suite, policies in the lower block use choose rules.

Domain	LAMA	BFWS	Lev.	IP	IP+Ch	IP / IP+Ch	
	S	S	S	S	S	d	w
(a) IPC-learning (90 tasks/domain)							
Blocksworld	61	36	61	90	90	0	0
Childsnack	35	35	54	90	90	0	0
Ferry	69	66	63	90	90	0	0
Floortile	11	13	38	90	90	0	0
Miconic	90	90	90	90	90	0	0
Rovers	70	65	67	90	90	0	0
Satellite	89	65	90	90	90	0	0
Spanner	30	30	59	90	90	0	0
Transport	66	63	64	90	90	0	0
Subtotal (810)	521	463	586	810	810		
Sokoban	41	46	44	40	41	42	42
Total (900)	562	509	630	850	851		
(b) Autoscale (30 tasks/domain)							
Barman	26	8	28	30	30	0	0
Blocksworld	27	10	29	30	30	0	0
Childsnack	9	7	21	30	30	0	0
Data-network	20	13	22	30	30	0	0
Depots	22	21	17	30	30	0	0
Driverlog	30	30	28	30	30	0	0
Elevators	30	30	30	30	30	0	0
Floortile	2	2	8	30	30	0	0
Grid	20	10	20	16	16	0	0
Gripper	30	30	30	30	30	0	0
Hiking	23	2	21	30	30	0	0
Logistics	18	10	15	30	30	0	0
Miconic	30	30	30	30	30	0	0
Parking	28	25	21	30	30	0	0
Rovers	30	30	30	30	30	0	0
Satellite	24	13	30	30	30	0	0
Scanalyzer	24	21	30	30	30	0	0
Storage	16	26	17	30	30	0	0
Tidybot	17	13	22	30	30	0	0
TPP	25	12	24	30	30	0	0
Transport	12	20	12	30	30	0	0
Visitall	30	25	30	30	30	0	0
Zenotravel	20	16	16	30	30	0	0
Subtotal (690)	513	404	531	676	676		
Agricola	30	30	30	30	30	49	9
Freecell	28	30	29	10	12	582	12
Mprime	9	12	24	27	28	687	25
Nomystery	10	14	22	1	5	28	4
Pegsol	30	30	30	27	30	33	11
Snake	9	18	16	23	23	11530	15
Sokoban	22	24	27	3	3	23	12
Termes	22	12	20	6	10	60	17
Tetris	19	26	23	26	28	46	100
Woodworking	14	5	16	30	30	119	47
Subtotal (300)	193	201	237	183	199		
Total (990)	706	605	768	859	875		

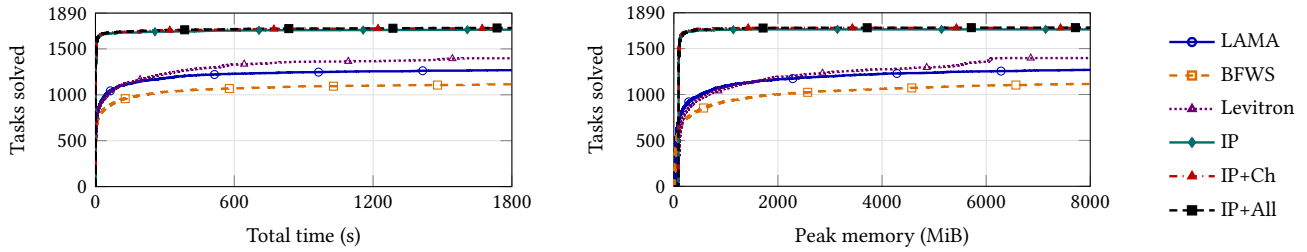


Figure 2: Solved tasks with total wall time (left) or peak memory (right) at most the horizontal-axis value, across 900 IPC-learning and 990 Autoscale tasks (linear axes). IP+Ch stores explored choice points; IP+All stores all explored augmented states.

8.1 Setup

Benchmarks. Our first benchmark, IPC-learning, is the test set of the IPC 2023 Learning Track [52]. Each domain has 30 easy, 30 medium, and 30 hard test tasks, which gives 900 tasks in total. Policies are learned from generated training tasks, and the test tasks are used only for evaluation. The second benchmark, Autoscale, comprises 33 domains from the Autoscale Agile suite [53] for which PDDL task generators are available. It contains 30 tasks per domain and 990 tasks in total. Training tasks are much smaller than test tasks: outside Sokoban and Nomystery, no training task has more than 65 objects, whereas test tasks have up to 19,615 objects. Peg Solitaire is the exception, where every task has 33 objects. The supplementary material lists the sizes per domain.

Learning. Both agents use gpt-6-astra with max reasoning effort. We run the loop once per domain, and all runs finish within 20 outer iterations. Each generator call allows five proposals, each with five syntax-repair attempts; each adversary call allows three proposals. Each adversary proposal specifies three configurations with 30 seeds each. Execution, verification, and solvability checks each allow 60 seconds and 10^6 states per task; execution uses one rollout. Structural termination checks allow 12 relevant features per cyclic component of the mode graph after preprocessing.

Evaluation. Each test task uses one thread, with limits of 30 minutes and 8000 MiB. We compare with LAMA [39], BFWS [35], and Levitron [17]. IP runs the learned policies with the scout of Section 5, which stores only its current path. Two variants of the scout spend space to save time. IP+Ch also stores every *choice point* that it has explored, that is, every augmented state of an OR vertex, so that it never searches a failed subtree again. IP+All stores every explored augmented state. The stored set can grow exponentially, so both variants give up the space bound of Corollary 4, but IP+Ch stores far fewer states than IP+All.

8.2 Results

Learning. Table 1 shows that the learned policies avoid choose rules in domains where plan existence is in P, such as Blocksworld, Logistics, Rovers, and Satellite [26, 27]. They use choose rules where plan existence is NP-complete, as in Freecell and Mprime [26], or PSPACE-complete, as in Sokoban [20]. Most policies without choose rules need one generator proposal (USD 1.07–2.46); Freecell, Mprime, Snake, and Sokoban need 11–20 generator proposals (USD 29.78–72.56). Every policy is proven structurally terminating and

proven to solve its training tasks (column P). The ablation in the supplementary material removes queries and introduces choose rules in Tidybot, Rovers, Data-network, and TPP, all of which are tractable and contain predicates of arity greater than two. Queries provide access to these predicates, avoiding unnecessary search caused by limited feature expressivity.

Testing. Table 2 shows that IP solves 850 of 900 IPC-learning tasks and 859 of 990 Autoscale tasks, compared with 630 and 768 for Levitron, the strongest baseline overall. Without choose rules, the scout never backtracks, and IP solves all 810 IPC-learning tasks and 676 of 690 Autoscale tasks. In the Autoscale domains with choose rules, IP matches or exceeds the best baseline in 5 of 10 domains, despite choice depths up to 11,530, choice widths up to 100, and unordered candidates. It falls far behind in Freecell, Nomystery, Sokoban, and Termes. PSPACE-complete Sokoban is outside our scope: unless NP = PSPACE, no structurally terminating policy solves all its tasks (Section 6.1). In Figure 2, IP solves most tasks within one second and 100 MiB, never exceeding 1,400 MiB; baselines keep gaining coverage up to 30 minutes and 8000 MiB. This reflects the space bound of Corollary 4: the scout stores only its current path. IP+Ch solves 1 and 16 more tasks than IP, with peak memory within 2% of that of IP on 90% of tasks and at most 291 MiB higher. IP+All solves 852 and 876 tasks, one more per benchmark than IP+Ch, but uses up to 1,819 MiB more than IP.

9 Conclusions

We set out to learn search control for intractable planning domains, rather than policies that decide a plan outright. Choose rules mark exactly the decisions that need search, and structural termination bounds every trajectory by a polynomial in the size of the task. A depth-first scout therefore finds a plan in polynomial space, at a cost in time that is exponential only in the choice depth.

We learned such policies with the counterexample-guided loop of previous work, extended with choose rules and with relational-algebra queries. With them, the scout solves more test tasks than LAMA, BFWS, and Levitron on both benchmarks, most within one second and 100 MiB. Storing explored choice points solves a few more tasks at little extra memory, but gives up the space bound.

Our guarantees have limits. Structural termination bounds the search but does not guarantee that it succeeds: a learned policy is proven to solve only its training tasks, and it can still fail on a task outside them. There is also a hard limit, since the bound that gives polynomial space caps the length of the plans that such

a policy can produce. A class that such a policy solves therefore lies in NP, which excludes Towers of Hanoi outright, and Sokoban unless NP = PSPACE. A natural next step is to learn policies with modules and calls, and to extend the depth bound to them.

References

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
- [2] Philip E. Agre and David Chapman. 1990. What Are Plans For? *Robotics and Autonomous Systems* 6, 1–2 (1990), 17–34.
- [3] Anonymous. 2026. Learning Structurally Terminating Indexical Policies with LLMs. Under review.
- [4] Anonymous. 2026. Prompt, Prove, Patch: The Neuro-Symbolic Loop for General Policy Synthesis. Under review.
- [5] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.
- [6] Fahiem Bacchus and Froduald Kabanza. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 116, 1–2 (2000), 123–191.
- [7] Blai Bonet, Dominik Drexler, and Hector Geffner. 2024. On Policy Reuse: An Expressive Language for Representing and Executing General Policies that Call Other Policies. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. AAAI Press, 31–39.
- [8] Blai Bonet, Guillem Francès, and Hector Geffner. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*. AAAI Press, 2703–2710.
- [9] Blai Bonet and Hector Geffner. 2018. Features, Projections, and Representation Change for Generalized Planning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018)*. IJCAI Organization, 4667–4673.
- [10] Blai Bonet and Hector Geffner. 2024. General Policies, Subgoal Structure, and Planning Width. *Journal of Artificial Intelligence Research* 80 (2024), 475–516.
- [11] Blai Bonet and Hector Geffner. 2025. Learning General Policies from Examples. In *Proceedings of the Twenty-Second International Conference on Principles of Knowledge Representation and Reasoning (KR 2025)*. IJCAI Organization, 740–750.
- [12] David Chapman. 1989. Penguins Can Make Cake. *AI Magazine* 10, 4 (1989), 45–50.
- [13] Dillon Ze Chen, Johannes Zenn, Tristan Cinquin, and Sheila A. McIlraith. 2025. Language Models for Generalised PDDL Planning: Synthesising Sound and Programmatic Policies. In *ICAPS 2025 Workshop on Planning in the Era of LLMs (LM4Plan)*.
- [14] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147, 1–2 (2003), 35–84.
- [15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Proceedings of the Twelfth International Conference on Computer Aided Verification (CAV 2000) (Lecture Notes in Computer Science, Vol. 1855)*. Springer, 154–169.
- [16] Edgar F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.
- [17] Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp. 2023. Levitron: Combining Ground and Lifted Planning. In *Tenth International Planning Competition (IPC-10): Planner Abstracts*.
- [18] Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. In *Proceedings of the Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS 2025)*. Neural Information Processing Systems Foundation, 47040–47078.
- [19] Augusto B. Corrêa, André G. Pereira, and Jendrik Seipp. 2025. Frontier Large Language Models Rival State-of-the-Art Planners. arXiv:2511.09378 [cs.AI].
- [20] Joseph C. Culberson. 1997. *Sokoban is PSPACE-complete*. Technical Report TR 97-02. Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada.
- [21] Rina Dechter. 2003. *Constraint Processing*. Morgan Kaufmann.
- [22] Dominik Drexler, Jendrik Seipp, and Hector Geffner. 2024. Expressing and Exploiting Subgoal Structure in Classical Planning Using Sketches. *Journal of Artificial Intelligence Research* 80 (2024), 171–208.
- [23] Dominik Drexler, Simon Ståhlberg, Blai Bonet, and Hector Geffner. 2024. Symmetries and Expressive Requirements for Learning General Policies. In *Proceedings of the Twenty-First International Conference on Principles of Knowledge Representation and Reasoning (KR 2024)*. IJCAI Organization, 845–855.
- [24] Kutluhan Erol, James A. Hendler, and Dana S. Nau. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18, 1 (1996), 69–93.
- [25] Guillem Francès, Blai Bonet, and Hector Geffner. 2021. Learning General Planning Policies from Small Examples Without Supervision. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*. AAAI Press, 11801–11808.
- [26] Malte Helmert. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence* 143, 2 (2003), 219–262.
- [27] Malte Helmert. 2006. New Complexity Results for Classical Planning Benchmarks. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*. AAAI Press, 52–61.
- [28] Malte Helmert and Gabriele Röger. 2008. How Good is Almost Perfect?. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*. AAAI Press, 944–949.
- [29] Yuxiao Hu and Giuseppe De Giacomo. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011)*. AAAI Press, 918–923.
- [30] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. 2019. A Review of Generalized Planning. *The Knowledge Engineering Review* 34 (2019), e5.
- [31] Andreas Junghanns and Jonathan Schaeffer. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129, 1–2 (2001), 219–251.
- [32] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B. Murthy. 2024. Position: LLMs Can’t Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Proceedings of the Forty-First International Conference on Machine Learning (ICML 2024)*. PMLR, 22895–22907.
- [33] Jonas Kvarnström and Patrick Doherty. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30, 1–4 (2000), 119–169.
- [34] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. 1997. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31, 1–3 (1997), 59–83.
- [35] Nir Lipovetzky and Hector Geffner. 2017. Best-First Width Search: Exploration and Exploitation in Classical Planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*. AAAI Press, 3590–3596.
- [36] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. arXiv:2304.11477 [cs.AI].
- [37] Mario Martín and Hector Geffner. 2004. Learning Generalized Policies from Planning Examples Using Concept Languages. *Applied Intelligence* 20, 1 (2004), 9–19.
- [38] Dana S. Nau, Tsz-Chiu Au, Oktay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20 (2003), 379–404.
- [39] Silvia Richter and Matthias Westphal. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39 (2010), 127–177.
- [40] Nicholas Rossetti, Massimiliano Tummolo, Alfonso Emilio Gerevini, Luca Putelli, Ivan Serina, Mattia Chiari, and Matteo Olivato. 2024. Learning General Policies for Planning through GPT Models. In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. AAAI Press, 500–508.
- [41] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. 2019. Computing programs for generalized planning using a classical planner. *Artificial Intelligence* 272 (2019), 52–85.
- [42] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. 2024. Generalized planning as heuristic search: A new planning search-space that leverages pointers over objects. *Artificial Intelligence* 330 (2024), 104097.
- [43] Jendrik Seipp and Malte Helmert. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research* 62 (2018), 535–577.
- [44] Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Michael Katz. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*. AAAI Press, 20256–20264.
- [45] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*. ACM, 404–415.
- [46] Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* 175, 2 (2011), 615–647.
- [47] Siddharth Srivastava, Shlomo Zilberstein, Neil Immerman, and Hector Geffner. 2011. Qualitative Numeric Planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*. AAAI Press, 1010–1016.

- [48] Simon Ståhlberg, Blai Bonet, and Hector Geffner. 2022. Learning Generalized Policies without Supervision Using GNNs. In *Proceedings of the Nineteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2022)*. IJCAI Organization, 474–483.
- [49] Simon Ståhlberg, Blai Bonet, and Hector Geffner. 2023. Learning General Policies with Policy Gradient Methods. In *Proceedings of the Twentieth International Conference on Principles of Knowledge Representation and Reasoning (KR 2023)*. IJCAI Organization, 647–657.
- [50] Katharina Stein, Nils Hodel, Daniel Fišer, Jörg Hoffmann, Michael Katz, and Alexander Koller. 2026. Improved Generalized Planning with LLMs through Strategy Refinement and Reflection. In *Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)*. AAAI Press, 677–686.
- [51] Katharina Stein, Chaahat Jain, Jörg Hoffmann, and Alexander Koller. 2026. Provably Complete Generalized Planning with LLMs. arXiv:2609.27105 [cs.AI].
- [52] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. 2024. The 2023 International Planning Competition. *AI Magazine* 45, 2 (2024), 280–296.
- [53] Álvaro Torralba, Jendrik Seipp, and Silvan Sievers. 2021. Automatic Instance Generation for Classical Planning. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. AAAI Press, 376–384.
- [54] Sam Toyer, Sylvie Thiébaux, Felipe Trevizan, and Lexing Xie. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research* 68 (2020), 1–68.
- [55] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023. On the Planning Abilities of Large Language Models - A Critical Investigation. In *Proceedings of the Thirty-Seventh Annual Conference on Neural Information Processing Systems (NeurIPS 2023)*. Neural Information Processing Systems Foundation, 75993–76005.

A Supplement to Section 5 (Search Spaces and Choose Rules)

PROPOSITION 2. *Let Π be a structurally terminating indexical policy that solves a task P . Then the scout returns a plan for P .*

PROOF. Since Π is structurally terminating, $G_{\Pi,P}$ is acyclic. Let σ be a choice function for which the root of $G_{\Pi,P}$ is solved, and let $G_{\Pi,P}^\sigma$ be the corresponding solution graph. Since $G_{\Pi,P}^\sigma$ is acyclic and the set of augmented states is finite, every trajectory of $G_{\Pi,P}^\sigma$ is finite, and since every leaf of $G_{\Pi,P}^\sigma$ is a goal state, every maximal one ends in a goal state. For every augmented state $\bar{s}$ that $G_{\Pi,P}^\sigma$ reaches from $\bar{s}_0$, let $\ell(\bar{s})$ be the length of the longest trajectory of $G_{\Pi,P}^\sigma$ from $\bar{s}$ to a goal state. We show by induction on $\ell(\bar{s})$ that the scout finds a plan from every such state $\bar{s}$. If $\bar{s}$ is a goal state, the scout stops. Otherwise, $\bar{s}$ is solved in $G_{\Pi,P}$, so every applicable choose rule in $\bar{s}$ has a candidate: an applicable choose rule without one would itself be an unsolved OR vertex, contradicting that $\bar{s}$ is solved. If the scout takes an ordinary transition, it picks one applicable rule and, if the rule allows several outcomes, one of them; both choices are among children of solved AND vertices, so the resulting state is itself solved, is reached by $G_{\Pi,P}^\sigma$, and has a smaller value of ℓ , so the scout finds a plan from it by induction. If the scout applies a choose rule, it tries the candidates one by one. Each candidate for which the rest of the execution fails is abandoned after finitely many steps, since $G_{\Pi,P}$ is acyclic and every vertex has finitely many children. The scout therefore eventually tries the candidate selected by σ , unless an earlier candidate succeeds, and it finds a plan from the successor of this candidate by induction. $\square$

B Supplement to Section 6 (Bounded Search Depth)

LEMMA 6. *There are integers $a \geq 1$ and $b \geq 0$, depending only on Π , such that on every task with n objects, every finite value of a numerical feature of Π is at most $U(n) = a(n+1)^D + b$. Each numerical feature thus takes at most $p(n) = U(n) + 2$ values, including ∞ .*

PROOF. Fix a task P with n objects. For each numerical subexpression e , we obtain an integer-valued polynomial $U_e(n) \geq 1$ that bounds every finite value of e on P . A constant c has the bound $\max\{1, c\}$, and cardinalities and finite distances have the bound $(n+1)^\kappa$. Given bounds U_1 and U_2 for two operands, their sum has the bound $U_1 + U_2$ and their product has the bound $U_1 U_2$. Truncated subtraction cannot exceed its finite left operand, and division by a positive finite integer cannot exceed its finite numerator. A finite minimum or maximum is one of its finite operands. If an operand is ∞ , a finite result of any of these operations is either 0 or a finite operand, so the same bounds still apply. Thus subtraction, division, minimum, and maximum may all use the bound $U_1 + U_2$. Induction on the expression gives a bound $c(n+1)^{\kappa m}$ when it contains m occurrences of cardinalities and distances: addition takes the larger exponent, multiplication adds exponents, and the other operations need no larger exponent than addition. Since Π has finitely many fixed expressions, $U(n) = a(n+1)^D + b$ bounds all finite numerical feature values for suitable $a \geq 1$ and $b \geq 0$, including those of expressions containing only constants. Including ∞ , each numerical feature has at most $p(n) = U(n) + 2$ values. $\square$

THEOREM 3. *Let Π be a structurally terminating indexical policy with choose rules. There is a polynomial q such that, for every task P with n objects, every trajectory of Π on P has at most $q(n)$ transitions.*

PROOF. Fix a task P with n objects. For a feature $f \in \Psi$ and an augmented state $\bar{s}$, write $f(\bar{s})$ for its actual value, and define

$$\alpha(\bar{s}) = (\bar{s}_{\text{mode}}, (f(\bar{s}))_{f \in \Psi}).$$

This tuple records numerical values, rather than only whether they are zero or positive. Each Boolean feature has at most two values, and each numerical feature has at most $p(n)$ values by Lemma 6, so there are at most

$$q(n) = |M| 2^{|\Psi_B|} p(n)^{|\Psi_N|}$$

possible tuples $\alpha(\bar{s})$.

Let G be the abstract graph of the termination test [3], in which choose rules are treated like load rules. Its vertices are the pairs (m, ν) of a mode m and a Boolean valuation ν of the features in Ψ , and its edges are labeled with rules. Every transition of Π on P satisfies the condition and the effects of a rule that induces it, where load and choose rules allow arbitrary changes to the features that depend on their register and preserve the other features. The transition therefore corresponds to an edge of G labeled with this rule. If the effect of the rule contains $f \downarrow$ (resp. $f \uparrow$), the value of f strictly decreases (resp. increases), and if the effect does not mention f , the value of f does not change.

Since Π is structurally terminating, the hybrid verifier accepts Π , and the hybrid verifier accepts exactly the policies that Sieve accepts [3]. Sieve repeatedly removes an edge from a strongly connected component K of the current graph if the edge makes progress on a feature f that no edge of K undoes, until the graph is acyclic. If f is numerical, the rule of the edge has the effect $f \downarrow$, and no edge of K has the effect $f \uparrow$ or $f?$, or symmetrically with $f \uparrow$. If f is Boolean, the rule of the edge flips f : its condition requires one value of f and its effect sets the other. The removal also requires that no other rule of K may flip f in the opposite direction, where a rule with the effect $f?$ may flip f in both directions and so opposes both. The condition of a rule that flips f fixes the value of f before the transition, so all edges of that rule flip f in the same direction, and every edge of K that changes f changes it in the direction of the removed edge.

Suppose a trajectory repeats a tuple, so that $\alpha(\bar{s}_i) = \alpha(\bar{s}_j)$ for some $i < j$. For each transition of the segment from $\bar{s}_i$ to $\bar{s}_j$, fix a rule that induces it. The corresponding edges of G form a closed walk, since the segment starts and ends with the same mode and the same feature values. Sieve makes G acyclic, so it removes some edge of this walk; let e be the first one that it removes. Until then, no edge of the walk has been removed, so the whole walk lies in the strongly connected component K from which e is removed, and e makes progress on a feature f that no edge of K undoes. If f is numerical and e has the effect $f \downarrow$, no edge of the walk has the effect $f \uparrow$ or $f?$, so the value of f never increases along the segment and strictly decreases at e , which gives $f(\bar{s}_j) < f(\bar{s}_i)$; the case $f \uparrow$ is symmetric. If f is Boolean, the segment changes f at e and never in the opposite direction, which gives $f(\bar{s}_j) \neq f(\bar{s}_i)$. Both cases contradict $\alpha(\bar{s}_i) = \alpha(\bar{s}_j)$.

Hence all tuples along a trajectory are distinct. A trajectory with ℓ transitions contains $\ell + 1$ tuples, so $\ell \leq q(n) - 1$. The modes,

the feature sets, and the polynomial p depend only on Π , so q is a polynomial in n . $\square$

COROLLARY 4. *Let Π be a structurally terminating indexical policy with choose rules, and let P be a task with n objects that Π solves. Then a depth-first scout that builds $G_{\Pi,P}$ on demand and stores only the current path from the root and the index of the candidate it tries at each OR vertex on it, returns a plan for P in space polynomial in n .*

PROOF. The scout enumerates the candidates of each choose rule in a fixed order, so that the index of a candidate determines the candidate. It stores the current path of $G_{\Pi,P}$ from the root: an augmented state at each state vertex, and, at each OR vertex on the path, the index of the candidate it is currently trying. At an AND vertex, whether an augmented state or an ordinary rule, it descends into the first child and stores nothing beyond the child itself. If the path reaches a state vertex that is a leaf but not a goal, or an OR vertex with no candidate left to try, the scout backtracks: it removes the path back to the most recent OR vertex with an untried candidate, advances its index, and continues from there, or backtracks further if no candidate is left. By the bound above, the path never exceeds $2q(n) + 2$ vertices, and since $G_{\Pi,P}$ is acyclic, the search terminates. Since Π solves P , Proposition 2 shows that it returns a plan, namely the actions along its final trajectory. For a fixed domain, a planning state has polynomially many atoms in n , a mode has constant size, a register valuation consists of a constant number of objects or object pairs, and a candidate index is a number below $n + 1$. Each vertex on the stored path thus has polynomial size, and so does the path itself. Finally, the features, and hence the successors and the candidates of a state, can be computed in polynomial time and therefore in polynomial space. $\square$

PROPOSITION 5. *Let Π be a structurally terminating policy that solves class $\mathcal{Q}$, and let q be the polynomial of Theorem 3. Every task of $\mathcal{Q}$ with n objects that has a plan has one with at most $q(n)$ actions, and deciding whether a task of $\mathcal{Q}$ has a plan is in NP. If the choice depth of Π is bounded by a constant on $\mathcal{Q}$, this decision is in P.*

PROOF. Let $P \in \mathcal{Q}$ have n objects and a plan. Then Π solves P , so the root of $G_{\Pi,P}$ is solved and a solution graph $G_{\Pi,P}^\sigma$ exists. A maximal trajectory of $G_{\Pi,P}^\sigma$ that starts at $\bar{s}_0$ ends in a goal state, and it has at most $q(n)$ transitions by Theorem 3. The actions that its transitions apply thus form a plan for P with at most $q(n)$ actions. Deciding plan existence for $\mathcal{Q}$ is therefore in NP, since such a plan is a certificate of polynomial size that can be checked in polynomial time for a fixed domain.

Let now the choice depth be at most a constant k on $\mathcal{Q}$. The scout terminates on every task, because $G_{\Pi,P}$ is acyclic. If P has a plan, then Π solves P and the scout returns a plan by Proposition 2. Every sequence of actions that the scout returns is a plan, so it reports failure if P has none. The scout explores at most $\sum_{i=0}^k B^i$ sequences of choices (Section 5.3), which is polynomial in n , because the choice width B is at most n . It follows each of them for at most $q(n)$ transitions by Theorem 3, and each transition costs polynomial time, since the features are evaluated in polynomial time and a state has polynomially many successors. The scout thus decides plan existence for $\mathcal{Q}$ in polynomial time. $\square$

C Supplement to Section 8 (Experiments)

Table 3 compares the number of objects of the training and test tasks. Table 4 reports learning effort and policy sizes for the ablation without queries. Table 5 compares test coverage for the main and ablation policies under IP, IP+Ch, and IP+All, together with LAMA, BFWS, and Levitron.

Without queries, the learned policies use choose rules in four additional domains: Rovers in both suites, and Data-network, Tidybot, and TPP in Autoscale. All four domains are tractable, and the policies learned with queries avoid choose rules in each of them. The additional choose rules thus introduce unnecessary search, indicating insufficient feature expressivity without queries. The ablation also has two unsuccessful learning runs, both in Autoscale: Mprime reaches the iteration limit, and Sokoban is cancelled after roughly 24 hours. In both cases, we test the last available policy before learning stops.

Table 3: Number of objects of the training and test tasks. Training tasks are all tasks that the adversary retained during learning, and n counts them. Med and max give the median and the maximum number of objects per task.

Domain	Training			Test	
	n	Med	Max	Med	Max
(a) IPC-learning					
Blocksworld	360	11	27	90	488
Childsnack	360	25	28	126	1326
Ferry	360	19	26	88	1461
Floortile	339	16	22	234	1022
Miconic	360	14	24	94	681
Rovers	360	21	30	149	596
Satellite	360	18	23	100	402
Sokoban	1331	127	578	1210	9880
Spanner	335	20	27	124	833
Transport	360	18	26	74	354
(b) Autoscale					
Agricola	328	29	48	54	76
Barman	350	21	31	84	158
Blocksworld	359	16	40	80	151
Childsnack	360	24	28	104	143
Data-network	276	31	36	192	262
Depots	353	21	25	256	477
Driverlog	360	23	42	294	542
Elevators	360	20	32	120	214
Floortile	344	17	27	500	1726
Freecell	1041	38	65	80	113
Grid	360	16	22	5418	19615
Gripper	12	13	21	96	169
Hiking	418	18	43	30	44
Logistics	360	18	28	1014	2376
Miconic	360	17	26	154	279
Mprime	848	18	24	62	98
Nomystery	356	24	78	640	981
Parking	346	13	19	74	127
Pegsol	352	33	33	33	33
Rovers	360	24	34	48	148
Satellite	360	19	32	264	507
Scanalyzer	10	16	20	74	132
Snake	852	20	27	89	182
Sokoban	898	59	91	113	170
Storage	341	24	32	132	300
Termes	335	11	17	53	107
Tetris	341	31	56	124	212
Tidybot	560	29	35	40	56
TPP	360	16	20	69	124
Transport	338	23	32	122	255
Visitall	360	26	36	2552	8836
Woodworking	345	36	46	172	238
Zenotravel	360	19	22	125	217

Table 4: Learning without queries on IPC-learning (top) and Autoscale (bottom). All finalized policies were proven structurally terminating. Columns follow Table 1, with T in hours. † marks interrupted learning: Mprime reached the iteration limit, and Autoscale Sokoban was cancelled after roughly 24 hours. Available partial statistics are shown; – denotes unavailable statistics. In each suite, policies in the lower block use choose rules.

Domain	G	A	P	T	F	M	R	Ch	d	w	USD
(a) IPC-learning											
Blocksworld	1	4	359	0.29	5	4	5	0	0	0	1.08
Childsnack	1	4	240	0.83	10	7	9	0	0	0	1.52
Ferry	1	4	336	0.35	4	4	5	0	0	0	1.12
Floortile	1	4	329	0.77	7	5	11	0	0	0	1.49
Miconic	1	4	360	0.32	4	4	8	0	0	0	1.29
Satellite	1	4	288	0.74	11	9	13	0	0	0	1.41
Spanner	1	4	330	0.88	4	2	4	0	0	0	1.22
Transport	1	4	360	0.39	9	6	8	0	0	0	1.28
Rovers	4	7	612	2.28	37	51	91	15	160	19	13.72
Sokoban	5	8	643	5.82	59	62	94	12	138	6	15.04
(b) Autoscale											
Barman	1	4	352	0.90	9	17	18	0	0	0	1.56
Blocksworld	1	4	359	0.12	5	4	5	0	0	0	1.40
Childsnack	1	4	270	0.52	11	8	10	0	0	0	1.45
Depots	1	4	292	1.12	8	7	8	0	0	0	1.41
Driverlog	1	4	355	0.70	13	21	32	0	0	0	2.91
Elevators	1	4	356	0.29	10	7	15	0	0	0	1.76
Floortile	1	4	345	0.65	8	5	10	0	0	0	1.73
Grid	1	4	274	1.25	8	8	13	0	0	0	2.11
Gripper	1	4	8	0.09	3	5	5	0	0	0	1.13
Hiking	2	5	450	0.85	16	21	28	0	0	0	3.49
Logistics	1	4	334	0.54	9	14	23	0	0	0	1.97
Miconic	1	4	360	0.19	4	4	8	0	0	0	1.32
Mprime†	20	4	–	5.32	–	–	–	–	–	–	49.48
Parking	1	4	218	0.92	19	17	33	0	0	0	2.06
Satellite	1	4	298	0.58	11	8	11	0	0	0	2.14
Scanalyzer	1	4	7	0.40	13	34	49	0	0	0	2.37
Storage	1	4	300	1.42	14	9	15	0	0	0	2.53
Transport	1	4	331	0.84	10	6	8	0	0	0	1.34
Visitall	1	4	278	0.83	3	1	2	0	0	0	1.00
Zenotravel	1	4	353	0.39	11	11	17	0	0	0	1.57
Agricola	1	4	319	2.22	26	12	33	3	45	14	4.70
Data-network	1	4	293	1.97	19	15	25	2	8	7	3.47
Freecell	13	14	1097	4.04	24	26	112	14	134	9	39.98
Nomystery	4	6	508	2.20	9	95	187	23	17	23	10.88
Pegsol	1	4	308	1.04	10	6	7	3	43	24	1.92
Rovers	4	6	540	1.20	22	45	102	18	46	383	8.92
Snake	5	6	501	1.82	7	5	10	4	163	23	8.96
Sokoban†	17	10	–	–	–	–	–	–	–	–	–
Termes	1	4	331	1.31	15	10	23	2	45	8	3.48
Tetris	19	10	786	17.45	41	17	71	1	65	30	52.17
Tidybot	6	9	480	12.45	56	79	157	4	29	11	24.53
TPP	1	4	360	0.36	14	10	15	1	15	25	2.45
Woodworking	1	4	295	2.83	16	9	27	2	8	8	3.33

Table 5: Testing the main policies and the ablation without queries on IPC-learning (top) and Autoscale (bottom). For ablation runs ending in failure or timeout (Autoscale Mprime and Sokoban), we test the last available policy. S counts solved tasks; bold marks maximum coverage, including ties. IP stores no visited states, IP+Ch stores explored choice points, and IP+All stores every explored augmented state. Both d, w pairs give maxima over tasks solved by all six policy configurations, independently of the baselines. Within each family, all three modes have identical maxima on these tasks; – denotes an empty common solved set. The domain blocks follow the main policies: policies in the lower block use choose rules.

Domain	Baselines			Main					Ablation				
	LAMA	BFWS	Levitron	IP	IP+Ch	IP+All	Shared		IP	IP+Ch	IP+All	Shared	
	S	S	S	S	S	S	d	w	S	S	S	d	w
(a) IPC-learning (90 tasks/domain)													
Blocksworld	61	36	61	90	90	90	0	0	90	90	90	0	0
Childsnack	35	35	54	90	90	90	0	0	90	90	90	0	0
Ferry	69	66	63	90	90	90	0	0	90	90	90	0	0
Floortile	11	13	38	90	90	90	0	0	90	90	90	0	0
Miconic	90	90	90	90	90	90	0	0	90	90	90	0	0
Rovers	70	65	67	90	90	90	0	0	90	90	90	2309	313
Satellite	89	65	90	90	90	90	0	0	90	90	90	0	0
Spanner	30	30	59	90	90	90	0	0	90	90	90	0	0
Transport	66	63	64	90	90	90	0	0	90	90	90	0	0
Subtotal (810)	521	463	586	810	810	810			810	810	810		
Sokoban	41	46	44	40	41	42	7	7	30	41	43	215	7
Total (900)	562	509	630	850	851	852			840	851	853		
(b) Autoscale (30 tasks/domain)													
Barman	26	8	28	30	30	30	0	0	30	30	30	0	0
Blocksworld	27	10	29	30	30	30	0	0	30	30	30	0	0
Childsnack	9	7	21	30	30	30	0	0	30	30	30	0	0
Data-network	20	13	22	30	30	30	0	0	30	30	30	74	47
Depots	22	21	17	30	30	30	0	0	30	30	30	0	0
Driverlog	30	30	28	30	30	30	0	0	30	30	30	0	0
Elevators	30	30	30	30	30	30	0	0	30	30	30	0	0
Floortile	2	2	8	30	30	30	0	0	30	30	30	0	0
Grid	20	10	20	16	16	15	0	0	13	13	13	0	0
Gripper	30	30	30	30	30	30	0	0	30	30	30	0	0
Hiking	23	2	21	30	30	30	0	0	30	30	30	0	0
Logistics	18	10	15	30	30	30	0	0	30	30	30	0	0
Miconic	30	30	30	30	30	30	0	0	30	30	30	0	0
Parking	28	25	21	30	30	30	0	0	30	30	30	0	0
Rovers	30	30	30	30	30	30	0	0	30	30	30	140	29
Satellite	24	13	30	30	30	30	0	0	30	30	30	0	0
Scanalyzer	24	21	30	30	30	30	0	0	30	30	30	0	0
Storage	16	26	17	30	30	30	0	0	30	30	30	0	0
Tidybot	17	13	22	30	30	30	0	0	30	30	30	65	13
TPP	25	12	24	30	30	30	0	0	30	30	30	28	55
Transport	12	20	12	30	30	30	0	0	30	30	30	0	0
Visitall	30	25	30	30	30	30	0	0	30	30	30	0	0
Zenotravel	20	16	16	30	30	30	0	0	30	30	30	0	0
Subtotal (690)	513	404	531	676	676	675			673	673	673		
Agricola	30	30	30	30	30	30	49	9	30	30	30	72	12
Freecell	28	30	29	10	12	14	582	12	14	22	24	78	10
Mprime	9	12	24	27	28	28	687	25	21	21	21	0	0
Nomystery	10	14	22	1	5	5	28	4	1	2	2	29	4
Pegsol	30	30	30	27	30	30	17	11	27	29	29	29	17
Snake	9	18	16	23	23	23	11530	15	28	29	29	1398	63
Sokoban	22	24	27	3	3	3	–	–	0	3	3	–	–
Termes	22	12	20	6	10	10	60	17	7	10	10	62	17
Tetris	19	26	23	26	28	28	28	96	15	17	17	90	33
Woodworking	14	5	16	30	30	30	119	47	19	20	20	55	59
Subtotal (300)	193	201	237	183	199	201			162	183	185		
Total (990)	706	605	768	859	875	876			835	856	858		