Learning Hierarchical Task Networks from Problem Solving

Pat Langley

Computational Learning Laboratory

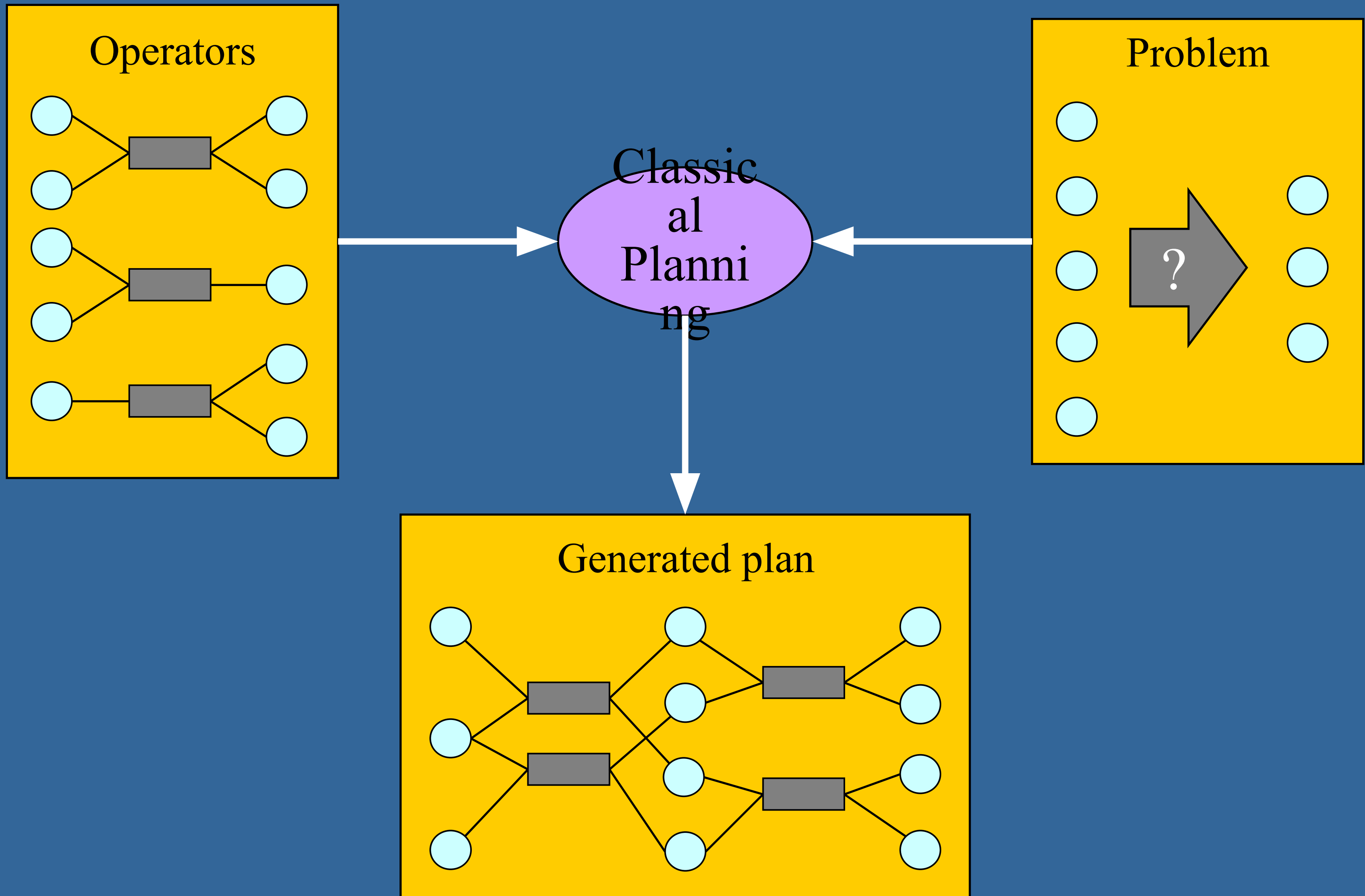
Center for the Study of Language and Information

Stanford University, Stanford, California

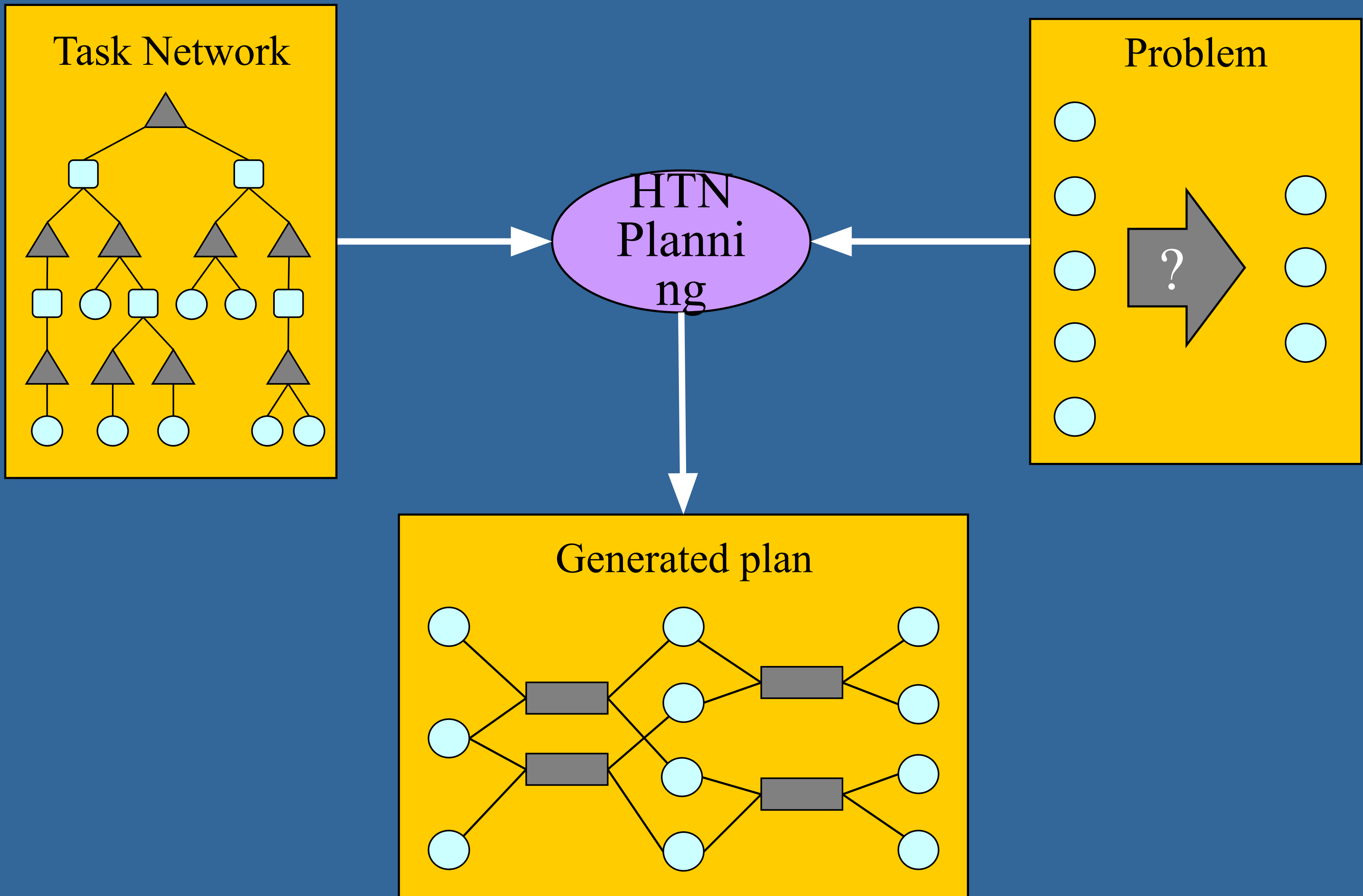
http://cll.stanford.edu/

Thanks to Dongkyu Choi, Kirstin Cummings, Seth Rogers, and Daniel Shapiro for contributions to this research, which was funded by Grant HR0011-04-1-0008 from DARPA IPTO and by Grant IIS-0335353 from NSF.

Classical Approaches to Planning



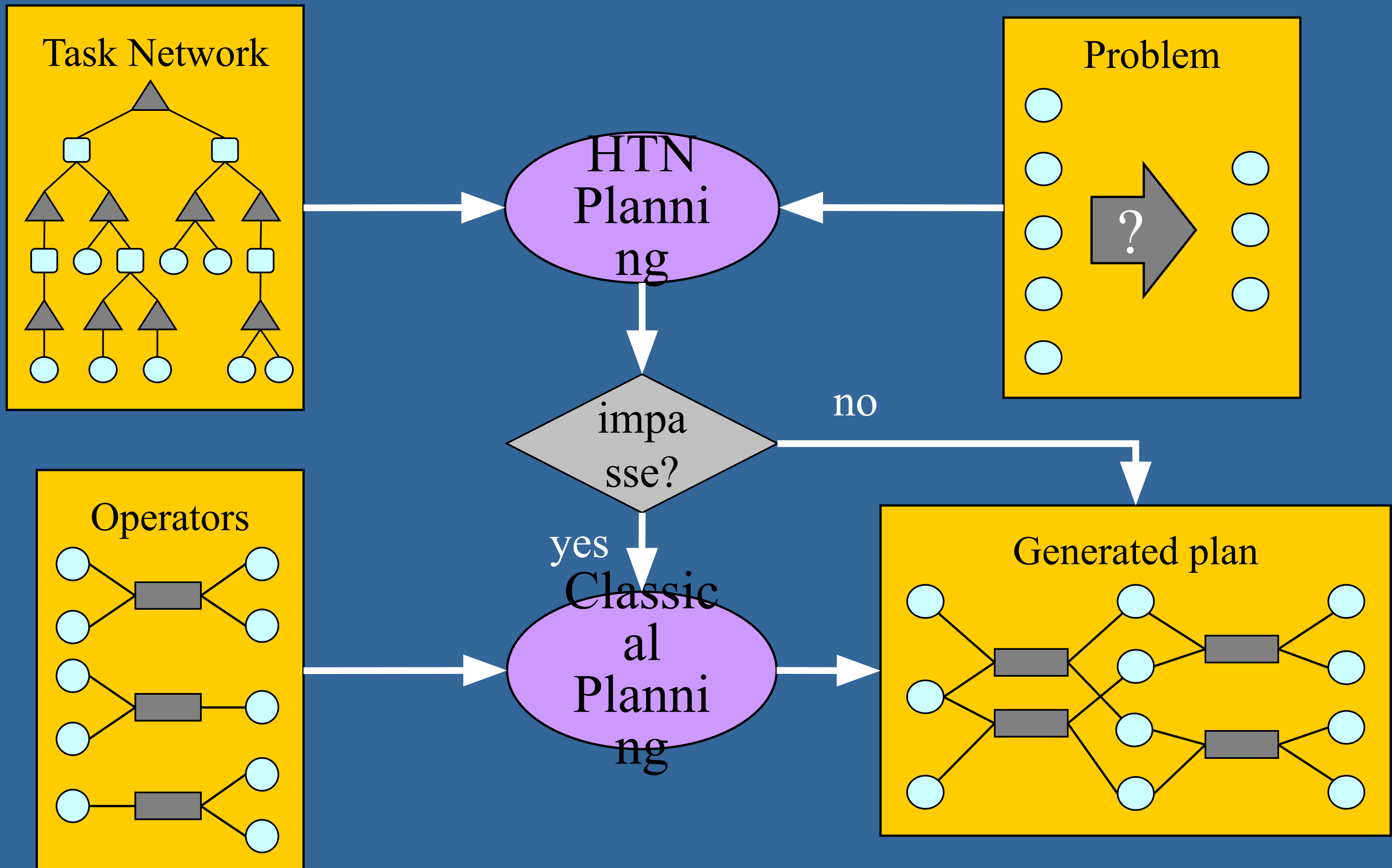
Planning with Hierarchical Task Networks



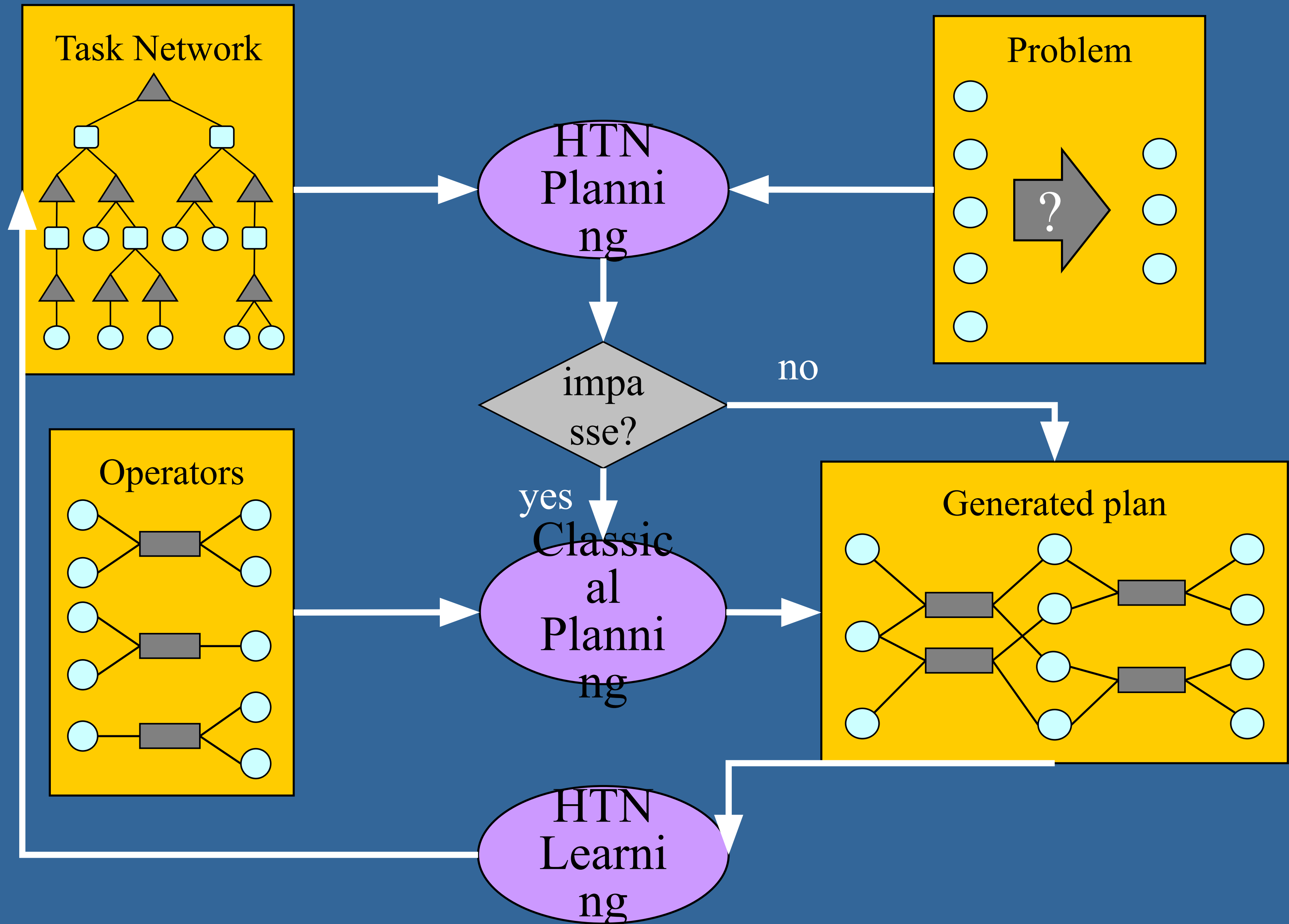
Classical and HTN Planning

- Challenge: *Can we unify classical and HTN planning in a single framework?*
- Challenge: *Can we use learning to gain the advantage of HTNs while avoiding the cost of manual construction?*
- Hypothesis: *The responses to these two challenges are closely intertwined.*

Mixed Classical / HTN Planning



Learning HTNs from Classical Planning



Four Contributions of the Research

- *Representation:* A specialized class of hierarchical task nets.
- *Execution:* A reactive controller that utilizes these structures.
- *Planning:* A method for interleaving HTN execution with problem solving when impasses are encountered.
- *Learning:* A method for creating new HTN methods from successful solutions to these impasses.

A New Representational Formalism

A *teleoreactive logic program* consists of three components:

- *Concepts:* A set of conjunctive relational inference rules;
- *Primitive skills:* A set of durative STRIPS operators;
- *Nonprimitive skills:* A set of HTN methods which specify:
  - a head that indicates a goal the method achieves;
  - a single (typically defined) precondition;
  - one or more ordered subskills for achieving the goal.

This special class of hierarchical task networks can be executed reactively but in a goal-directed manner (Nilsson, 1994).

Some Defined Concepts (Axioms)

```
(clear (?block)
 :percepts    ((block ?block))
 :negatives   ((on ?other ?block)))

(hand-empty ( )
 :percepts    ((hand ?hand status ?status))
 :tests   ((eq ?status 'empty)))

(unstackable (?block ?from)
 :percepts    ((block ?block) (block ?from))
 :positives   ((on ?block ?from) (clear ?block) (hand-empty)))

(pickupable (?block ?from)
 :percepts    ((block ?block) (table ?from))
 :positives   ((ontable ?block ?from) (clear ?block) (hand-empty)))

(stackable (?block ?to)
 :percepts    ((block ?block) (block ?to))
 :positives   ((clear ?to) (holding ?block)))

(putdownable (?block ?to)
 :percepts    ((block ?block) (table ?to))
 :positives   ((holding ?block)))
```

Some Primitive Skills (Operators)

```
(unstack (?block ?from)
 :percepts      ((block ?block ypos ?ypos) (block ?from))
 :start     (unstackable ?block ?from)
 :actions ((*grasp ?block) (*vertical-move ?block (+ ?ypos 10)))
 :effects  ((clear ?from) (holding ?block)))

(pickup (?block ?from)
 :percepts      ((block ?block) (table ?from height ?height))
 :start     (pickupable ?block ?from)
 :effects  ((holding ?block)))

(stack (?block ?to)
 :percepts      ((block ?block) (block ?to xpos ?xpos ypos ?ypos height
?height))
 :start     (stackable ?block ?to)
 :effects  ((on ?block ?to) (hand-empty)))

(putdown (?block ?to)
 :percepts      ((block ?block) (table ?to xpos ?xpos ypos ?ypos height
?height))
 :start     (putdownable ?block ?to)
 :effects  ((ontable ?block ?to) (hand-empty)))
```

Some NonPrimitive Recursive Skills

```
(clear (?C)
 :percepts    ((block ?D) (block ?C))
 :start    (unstackable ?D ?C)
 :skills    ((unstack ?D ?C)))

(clear (?B)
 :percepts    ((block ?C)  (block ?B))
 :start    [(on ?C ?B)  (hand-empty)]
 :skills    ((unstackable ?C ?B)  (unstack ?C ?B)))

(unstackable (?C ?B)
 :percepts    ((block ?B)  (block ?C))
 :start    [(on ?C ?B)  (hand-empty)]
 :skills    ((clear ?C)  (hand-empty)))

(hand-empty (  )
 :percepts    ((block ?D)  (table ?T1))
 :start    (putdownable ?D ?T1)
 :skills    ((putdown ?D ?T1)))
```

[Expanded for readability]

Teleoreactive logic programs are executed in a top-down, left-to-right manner, much as in Prolog but extended over time, with a single path being selected on each time step.

Interleaving HTN Execution and Classical Planning

```
Solve(G)
  Push the goal literal G onto the empty goal stack GS.
  On each cycle,
    If the top goal G of the goal stack GS is satisfied,
    Then pop GS.
    Else if the goal stack GS does not exceed the depth limit,
        Let S be the skill instances whose heads unify with G.
        If any applicable skill paths start from an instance in S,
        Then select one of these paths and execute it.
        Else let M be the set of primitive skill instances that have not already failed in which G is an effect.
            If the set M is nonempty,
            Then select a skill instance Q from M.
          Push the start condition C of Q onto goal stack GS.
            Else if G is a complex concept with the unsatisfied subconcepts H and with satisfied subconcepts F,
                Then if there is a subconcept I in H that has not yet failed,
                        Then push I onto the goal stack GS.
                        Else pop G from the goal stack GS and store information about failure with G's parent.
                Else pop G from the goal stack GS.
                      Store information about failure with G's parent.
```

This is traditional means-ends analysis, with three exceptions: (1) conjunctive goals must be defined concepts; (2) chaining occurs over both skills/operators and concepts/axioms; and (3) selected skills are executed whenever applicable.

A Successful Planning Trace

initial state

(clear C)

(hand-empty)

(on C B)

(ontable A T)

(on B A)

(unst. C B)

(holding C)

(unstack C B)

(putdown C T)

(clear B)

(hand-empty)

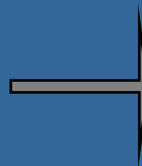
(unst. B A)

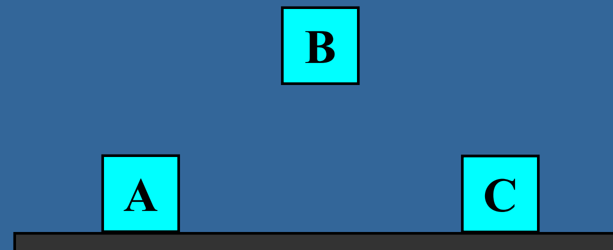
(unstack B A)

goal

(clear A)

(holding B)





Three Questions about HTN Learning

- What is the hierarchical structure of the network?
- What are the heads of the learned clauses/methods?
- What are the conditions on the learned clauses/methods?

The answers follow naturally from our representation and from our approach to plan generation.

Recording Results for Learning

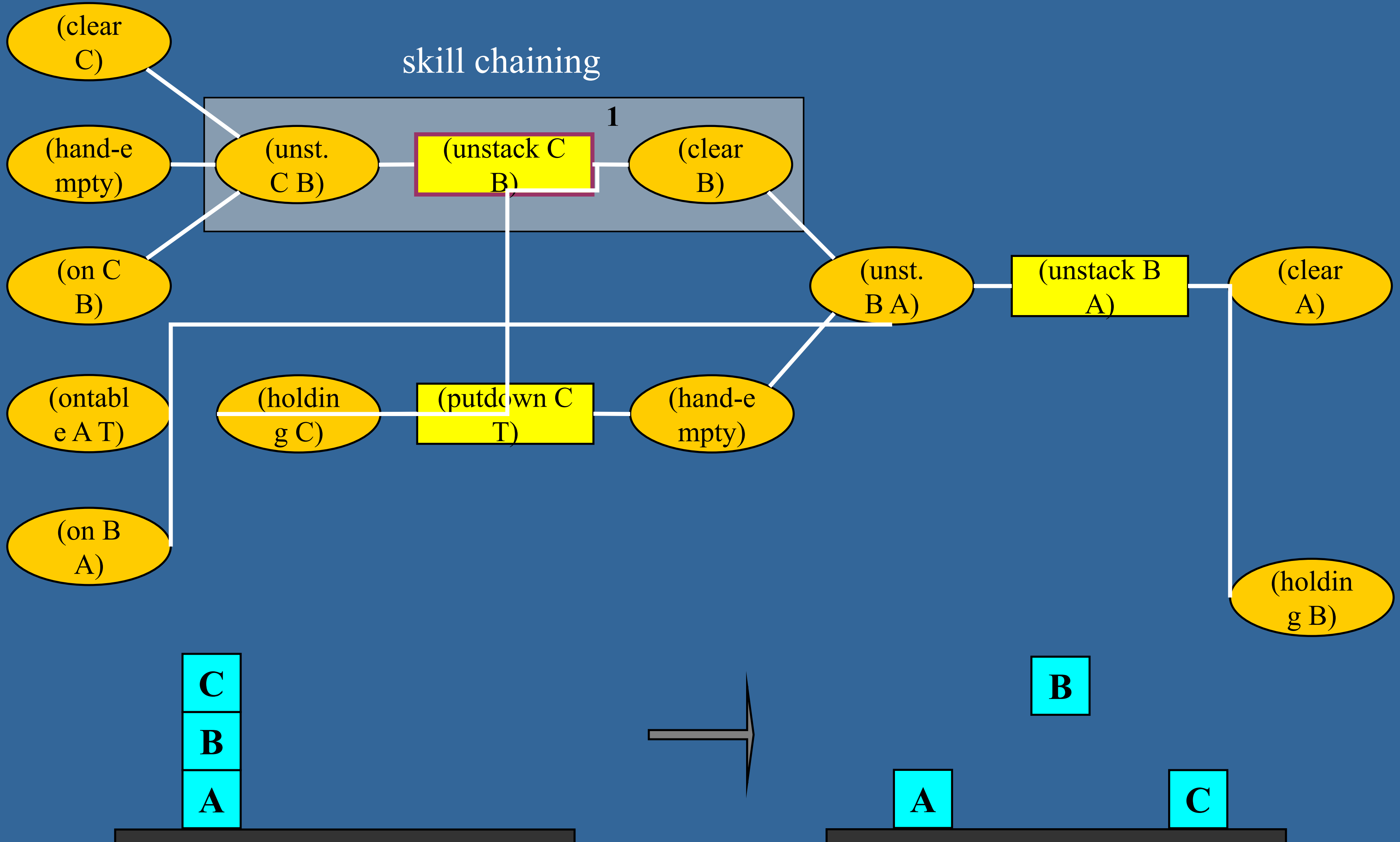
Solve(G)
Push the goal literal G onto the empty goal stack GS.
On each cycle,
If the top goal G of the goal stack GS is satisfied,
Then pop GS *and let New be Learn(G).*
If G's parent P involved skill chaining,
Then store New as P's first subskill.
Else if G's parent P involved concept chaining,
Then store New as P's next subskill.
Else if the goal stack GS does not exceed the depth limit,
Let S be the skill instances whose heads unify with G.
If any applicable skill paths start from an instance in S,
Then select one of these paths and execute it.
Else let M be the set of primitive skill instances that have not already failed in which G is an effect.
If the set M is nonempty,
Then select a skill instance Q from M, *store Q with goal G as its last subskill,*
Push the start condition C of Q onto goal stack GS*, and mark goal G as involving skill chaining.*
Else if G is a complex concept with the unsatisfied subconcepts H and with satisfied subconcepts F,
Then if there is a subconcept I in H that has not yet failed,
Then push I onto the goal stack GS, *store F with G as its initially true subconcepts,*
and mark goal G as involving concept chaining.
Else pop G from the goal stack GS and store information about failure with G's parent.
Else pop G from the goal stack GS.
Store information about failure with G's parent.

The extended problem solver calls on *Learn* to construct a new skill clause and stores the information it needs in the goal stack generated during search.

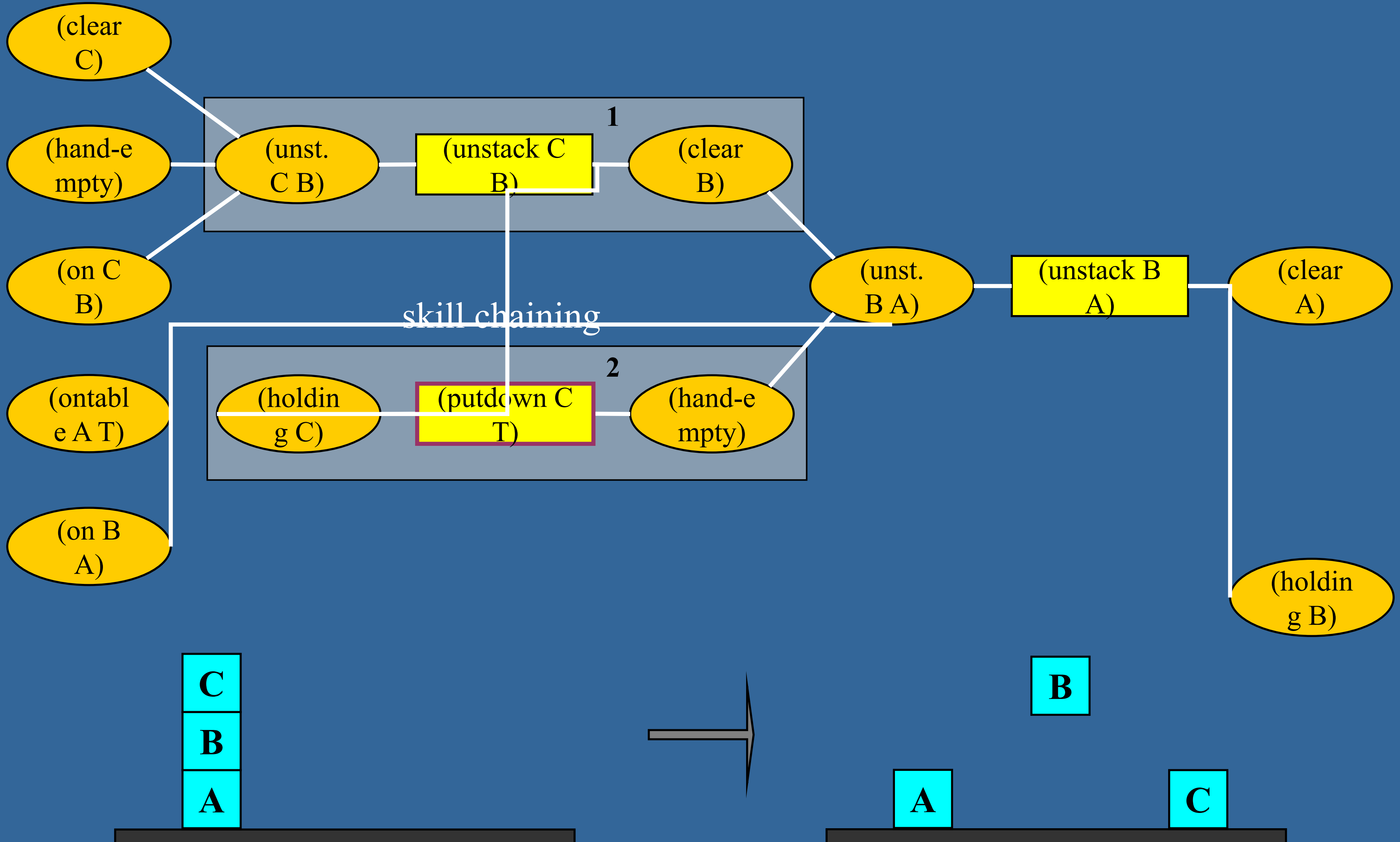
Three Questions about HTN Learning

- What is the hierarchical structure of the network?
  - *The structure is determined by the subproblems solved during planning, which, because both operator conditions and goals are single literals, form a semilattice.*
- What are the heads of the learned clauses/methods?
- What are the conditions on the learned clauses/methods?

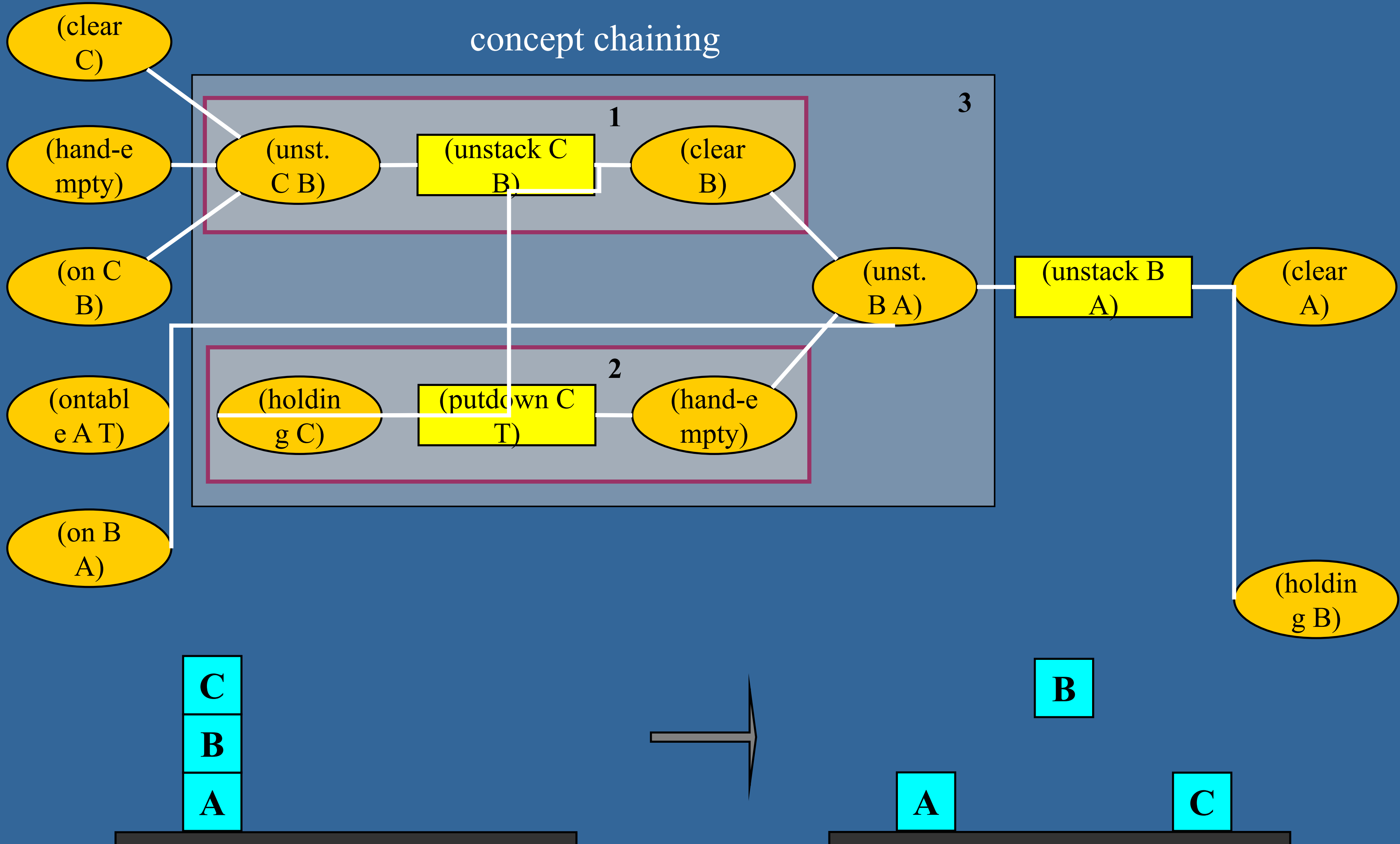
Constructing Skills from a Trace



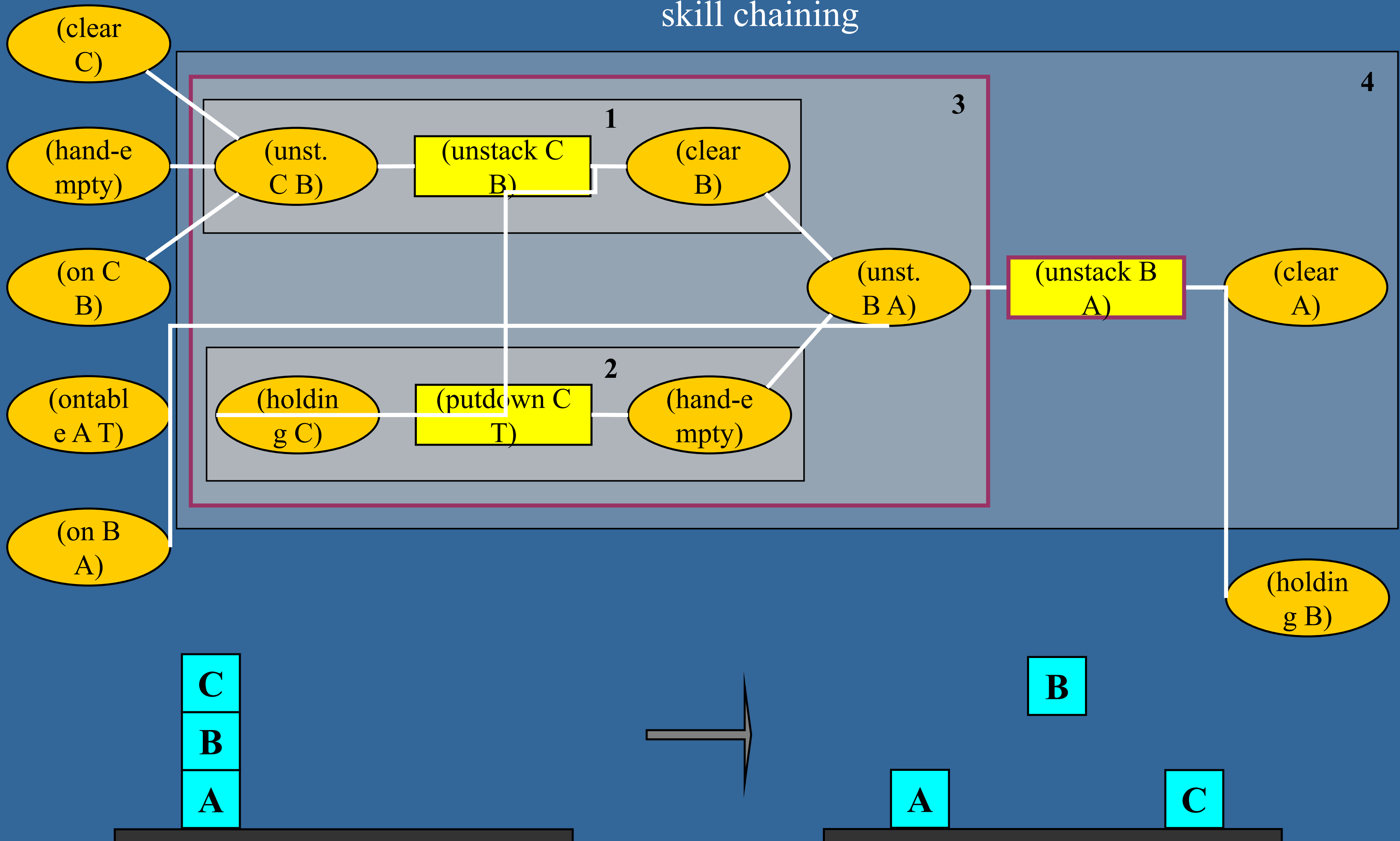
Constructing Skills from a Trace



Constructing Skills from a Trace



Constructing Skills from a Trace



Learned Skills After Structure Determined

```
(<head>  (?C)
 :percepts      ((block ?D) (block ?C))
 :start     <condition>
 :skills    ((unstack ?D ?C)))

(<head>  (?B)
 :percepts      ((block ?C)  (block ?B))
 :start     <condition>
 :skills    ((unstackable ?C ?B)  (unstack ?C ?B)))

(<head>  (?C ?B)
 :percepts      ((block ?B)  (block ?C))
 :start     <condition>
 :skills    ((clear ?C)  (hand-empty)))

(<head>  (  )
 :percepts      ((block ?D)  (table ?T1))
 :start     <condition>
 :skills    ((putdown ?D ?T1)))
```

Three Questions about HTN Learning

- What is the hierarchical structure of the network?
  - The structure is determined by the subproblems solved during planning, which, because both operator conditions and goals are single literals, form a semilattice.
- What are the heads of the learned clauses/methods?
  - *The head of a learned clause is the goal literal that the planner achieved for the subproblem that produced it.*
- What are the conditions on the learned clauses/methods?

Learned Skills After Heads Inserted

```
(clear (?C)
 :percepts    ((block ?D) (block ?C))
 :start       <condition>
 :skills      ((unstack ?D ?C)))

(clear (?B)
 :percepts    ((block ?C)  (block ?B))
 :start       <condition>
 :skills      ((unstackable ?C ?B)  (unstack ?C ?B)))

(unstackable (?C ?B)
 :percepts    ((block ?B)  (block ?C))
 :start       <condition>
 :skills      ((clear ?C)  (hand-empty)))

(hand-empty (  )
 :percepts    ((block ?D)  (table ?T1))
 :start       <condition>
 :skills      ((putdown ?D ?T1)))
```

Three Questions about HTN Learning

- What is the hierarchical structure of the network?
  - The structure is determined by the subproblems solved during planning, which, because both operator conditions and goals are single literals, form a semilattice.
- What are the heads of the learned clauses/methods?
  - The head of a learned clause is the goal literal that the planner achieved for the subproblem that produced it.
- What are the conditions on the learned clauses/methods?
  - *If the subproblem involved skill chaining, they are the conditions of the first subskill clause.*
  - *If the subproblem involved concept chaining, they are the subconcepts that held at the outset of the subproblem.*

Learned Skills After Conditions Inferred

```
(clear (?C)
 :percepts   ((block ?D) (block ?C))
 :start      (unstackable ?D ?C)
 :skills     ((unstack ?D ?C)))

(clear (?B)
 :percepts   ((block ?C)  (block ?B))
 :start      [(on ?C ?B)  (hand-empty)]
 :skills     ((unstackable ?C ?B)  (unstack ?C ?B)))

(unstackable (?C ?B)
 :percepts   ((block ?B)  (block ?C))
 :start      [(on ?C ?B)  (hand-empty)]
 :skills     ((clear ?C)  (hand-empty)))

(hand-empty ( )
 :percepts   ((block ?D)  (table ?T1))
 :start      (putdownable ?D ?T1)
 :skills     ((putdown ?D ?T1)))
```

Learning an HTN Method from a Problem Solution

Learn(G)

If the goal G involves skill chaining,

Then let S_1 and S_2 be G's first and second subskills.

If subskill S_1 is empty,

Then return the literal for clause S_2.

Else create a new skill clause N with head G,

with S_1 and S_2 as ordered subskills, and

with the same start condition as subskill S_1.

Return the literal for skill clause N.

Else if the goal G involves concept chaining,

Then let $\{C_{k+1}, ..., C_n\}$ be G's initially satisfied subconcepts.

Let $\{C_1, ..., C_k\}$ be G's stored subskills.

Create a new skill clause N with head G,

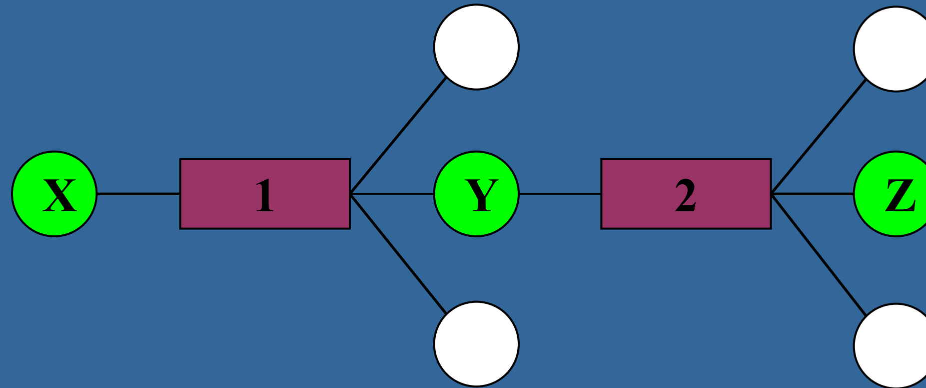
with $\{C_{k+1}, ..., C_n\}$ as ordered subskills, and

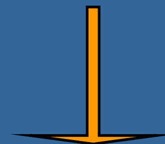
with the conjunction of $\{C_1, ..., C_k\}$ as start condition.

Return the literal for skill clause N.

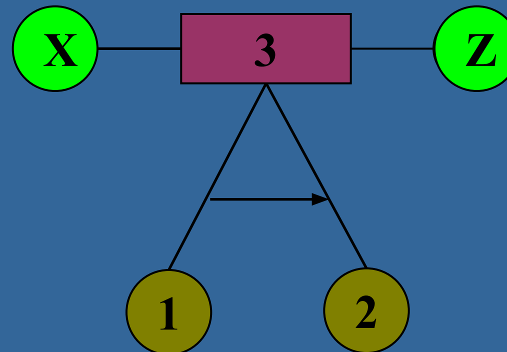
Creating a Clause from Skill Chaining

Problem Solution



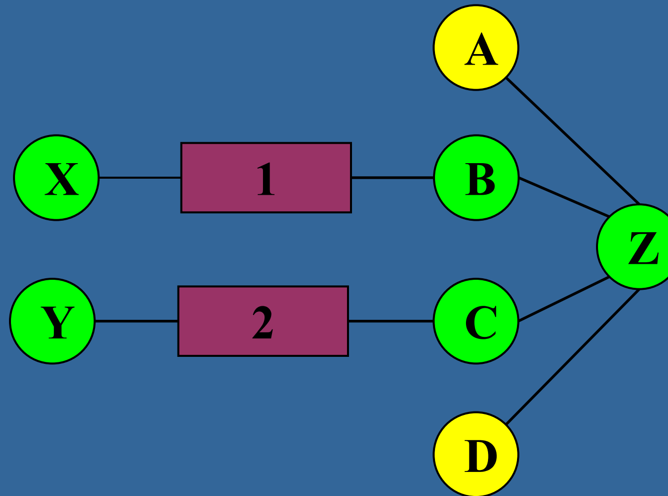


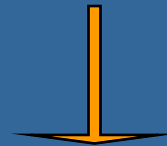
New Method



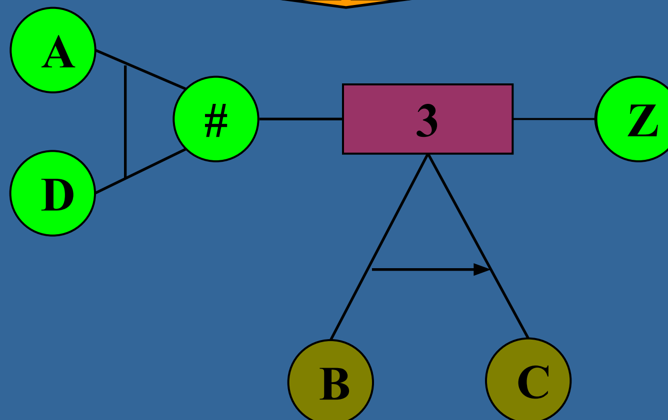
Creating a Clause from Concept Chaining

Problem Solution





New Method



Important Features of Learning Method

Our approach to learning HTNs has some important features:

- it occurs incrementally from one experience at a time;
- it takes advantage of existing background knowledge;
- it constructs the hierarchies in a *cumulative* manner.

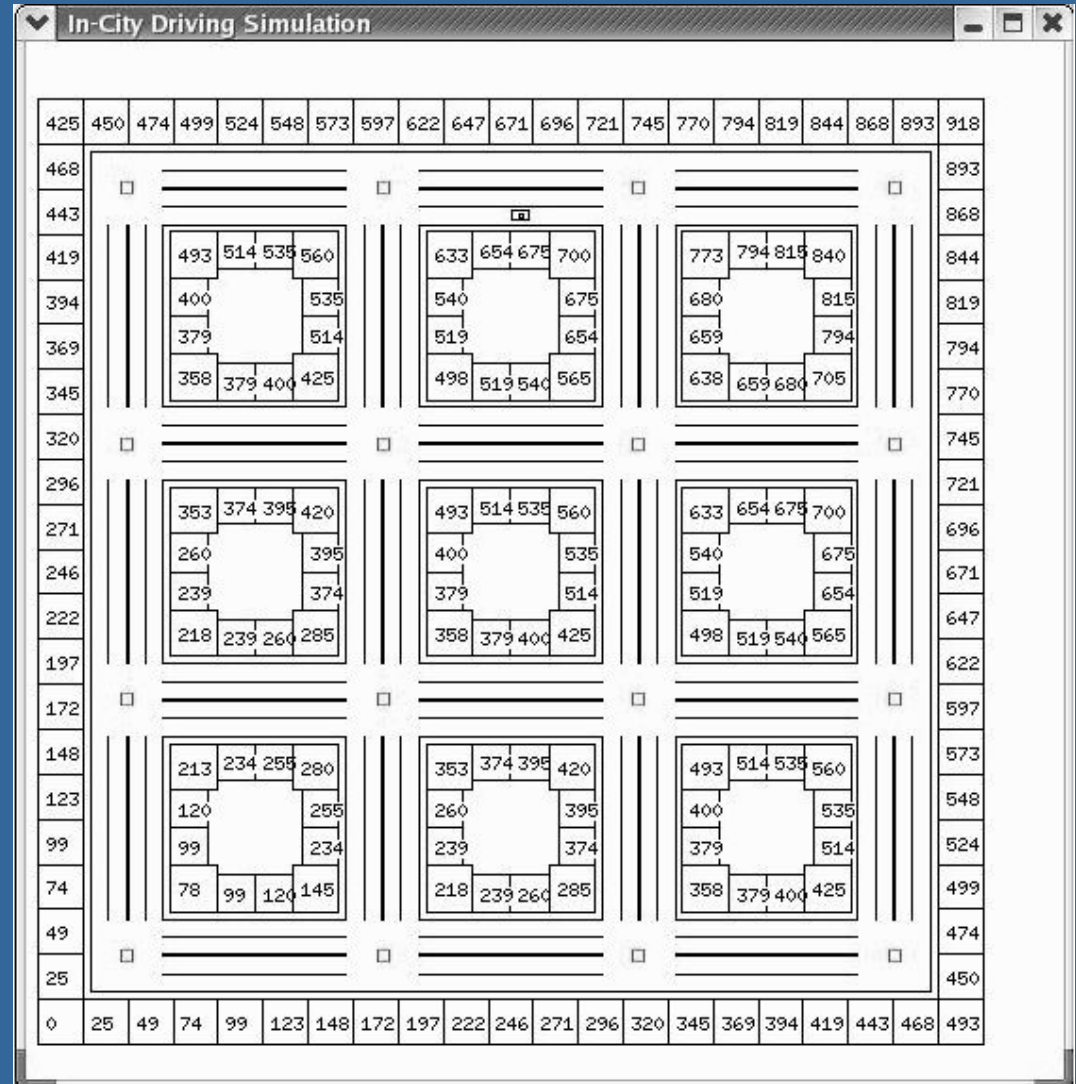
In these ways, it is similar to explanation-based approaches to learning from problem solving.

However, the method for finding conditions involves neither analytic or inductive learning in their traditional senses.

An In-City Driving Environment

Our focus on learning for reactive control comes from an interest in complex physical domains, such as driving a vehicle in a city.

To study this problem, we have developed a realistic simulated environment that can support many different driving tasks.



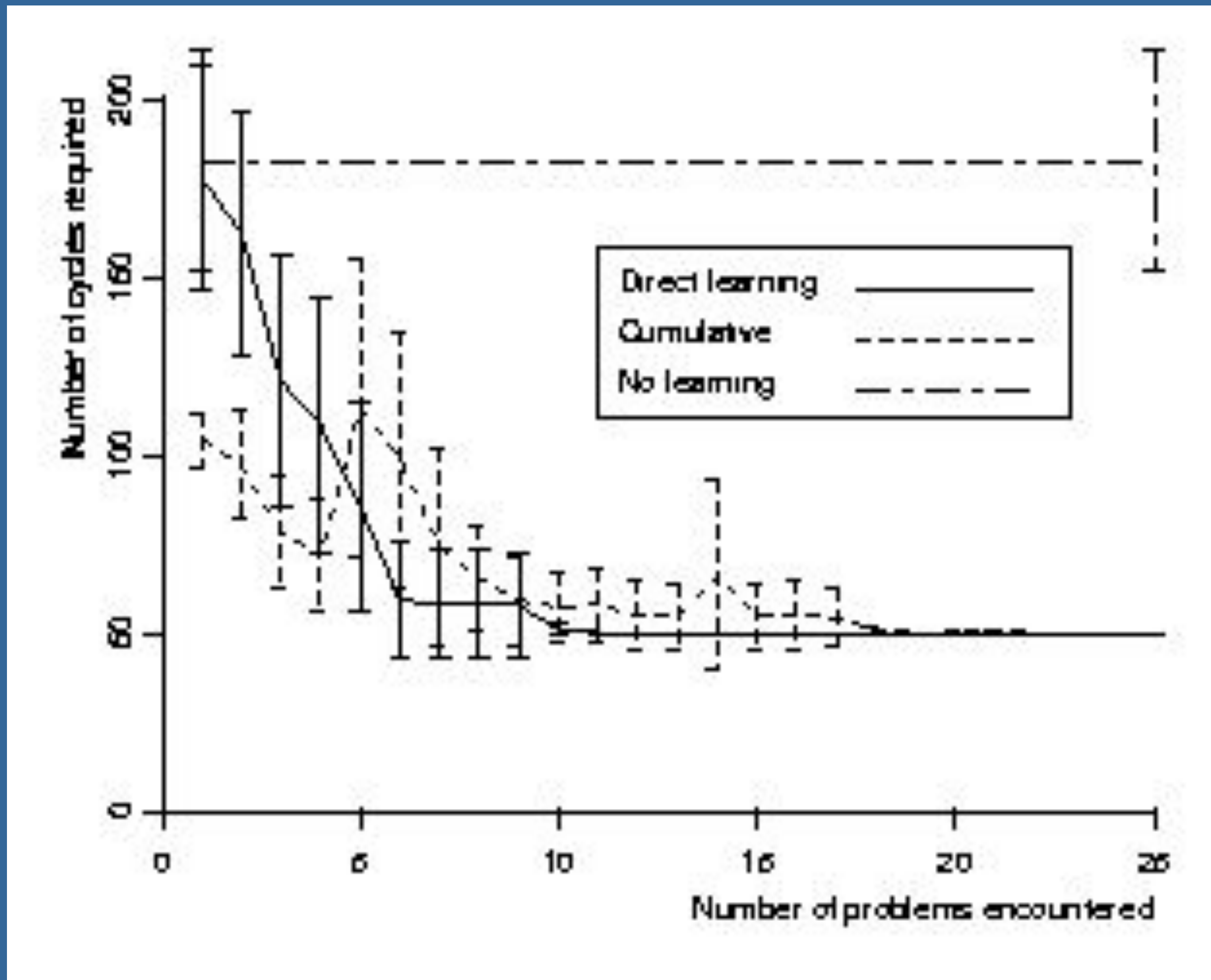
Skill Clauses Learning for In-City Driving

```
parked (?ME ?G1152)
   :percepts   (     (lane-line ?G1152) (self ?ME))
   :start   ( )
   :skills  (     (in-rightmost-lane ?ME ?G1152)
                  (stopped ?ME))

 in-rightmost-lane (?ME ?G1152)
   :percepts   (     (self ?ME) (lane-line ?G1152))
   :start      (     (last-lane ?G1152))
   :skills     (     (driving-in-segment ?ME ?G1101 ?G1152))

driving-in-segment (?ME ?G1101 ?G1152)
   :percepts   (     (lane-line ?G1152) (segment ?G1101) (self
?ME))
   :start      (     (steering-wheel-straight ?ME))
   :skills     (     (in-lane ?ME ?G1152)
                  (centered-in-lane ?ME ?G1101 ?G1152)
                  (aligned-with-lane-in-segment ?ME ?G1101 ?G1152)
                  (steering-wheel-straight ?ME))
```

Learning Curves for In-City Driving



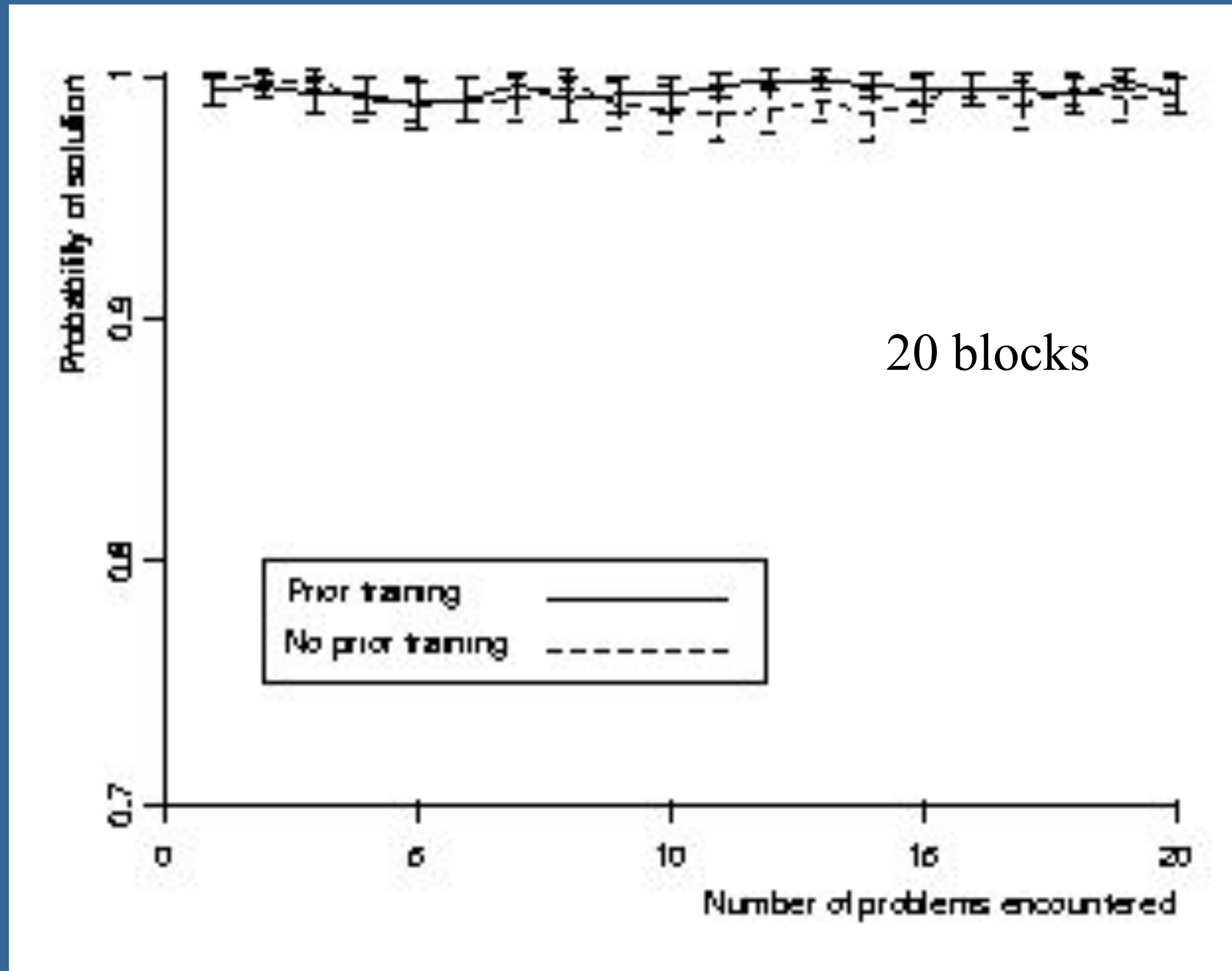
Transfer Studies of HTN Learning

Because we were interested in our method's ability to transfer its learned skills to harder problems, we:

- created concepts and operators for Blocks World and FreeCell;
- let the system solve and learn from simple training problems;
- asked the system to solve and learning from harder test tasks;
- recorded the number of steps taken and solution probability;
- as a function of the number of transfer problems encountered;
- averaged the results over many different problem orders.

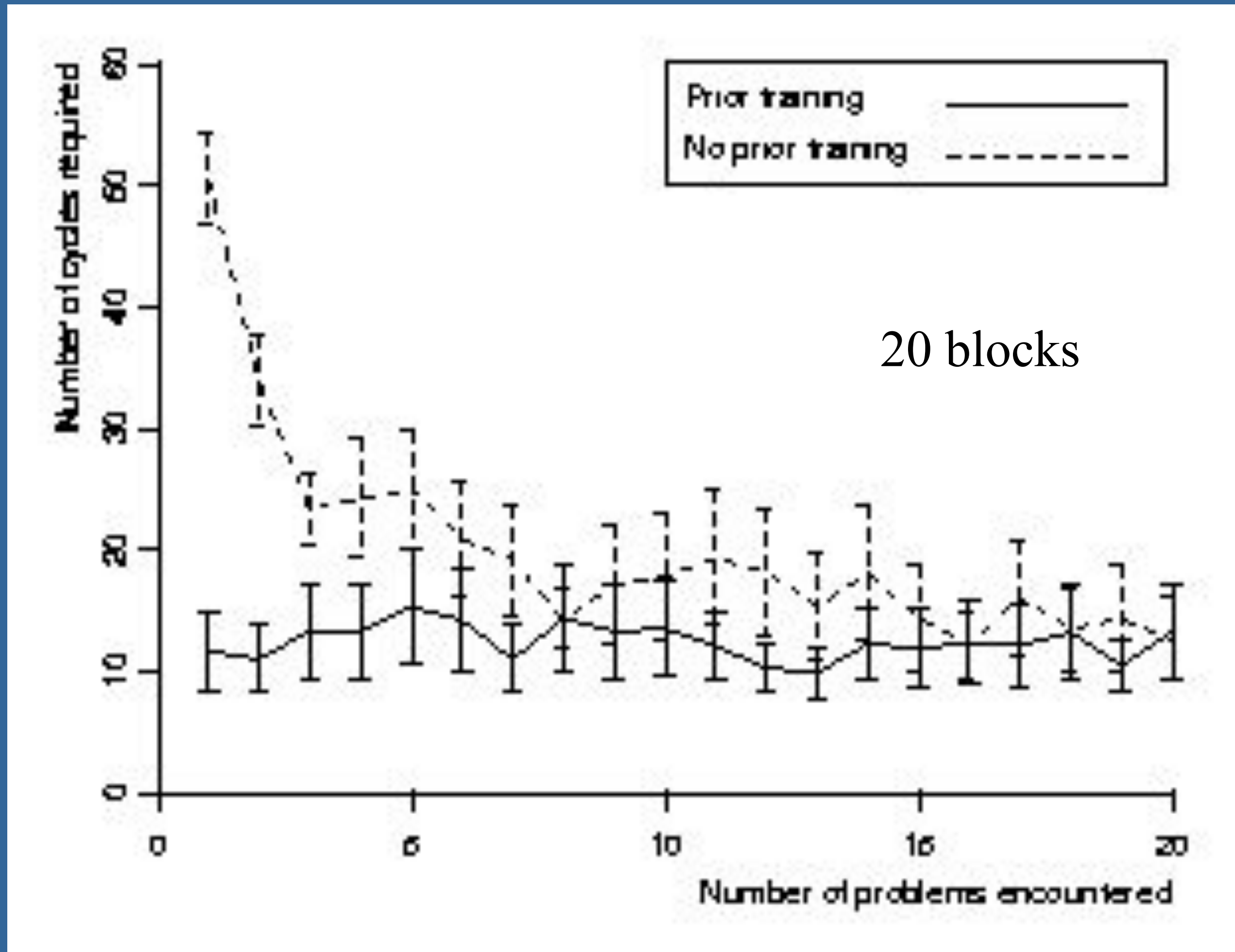
The resulting transfer curves revealed the system's ability to take advantage of prior learning and generalize to new situations.

Transfer Effects in the Blocks World



On 20-block tasks, there is no difference in solved problems.

Transfer Effects in the Blocks World

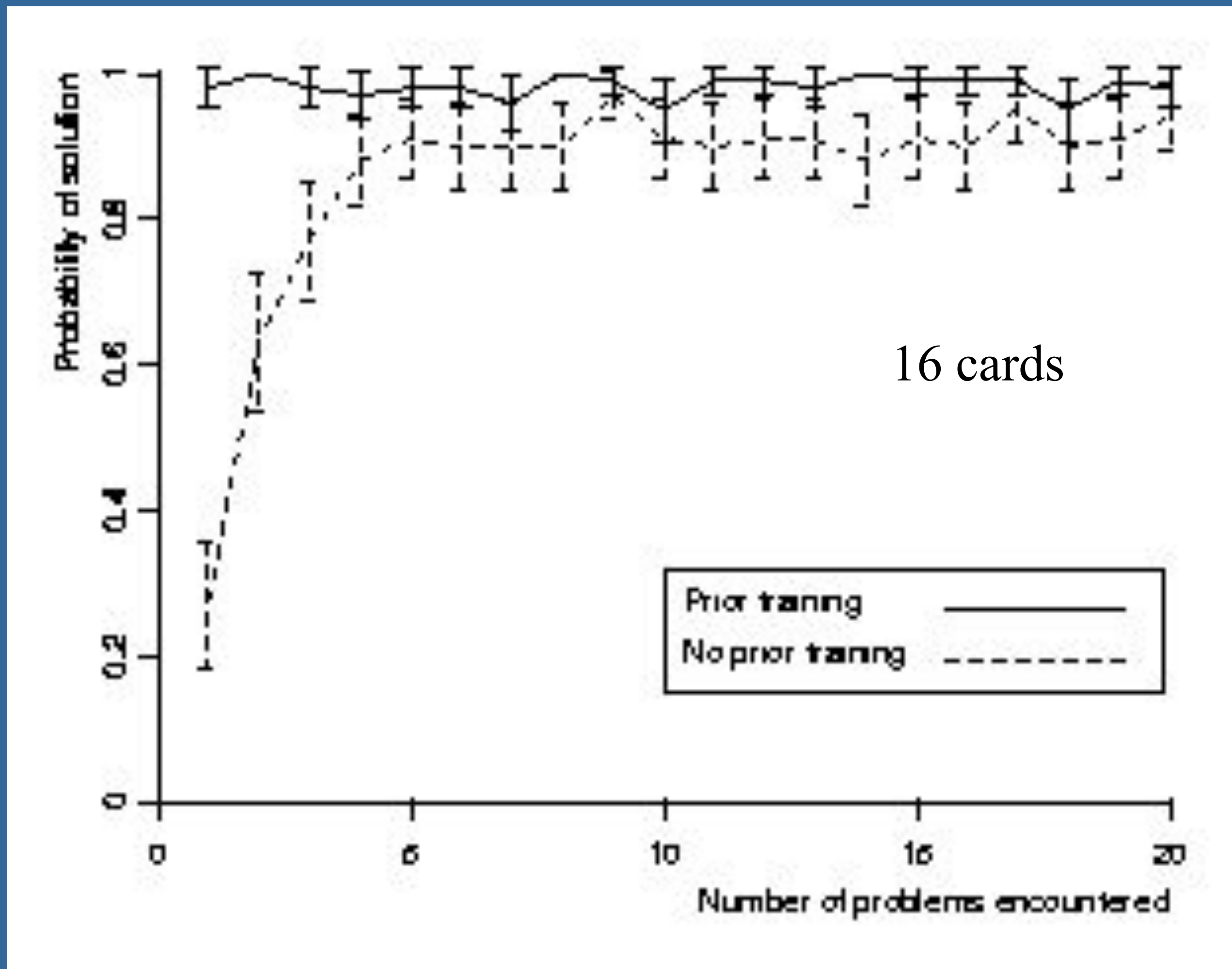


However, there is difference in the effort needed to solve them.

FreeCell Solitaire

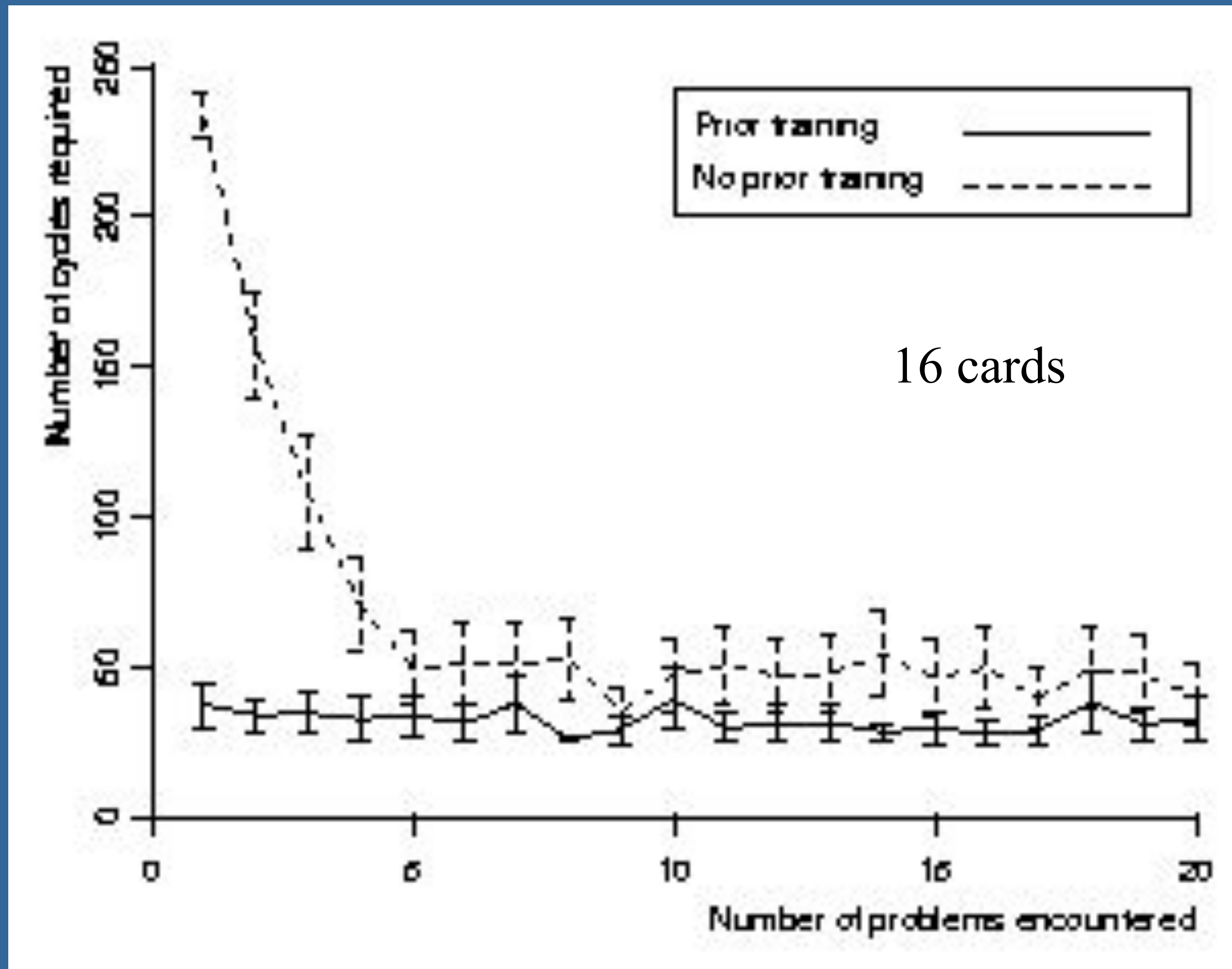
FreeCell is a full-information card game that, in most cases, can be solved by planning; it also has a highly recursive structure.

Transfer Effects in FreeCell



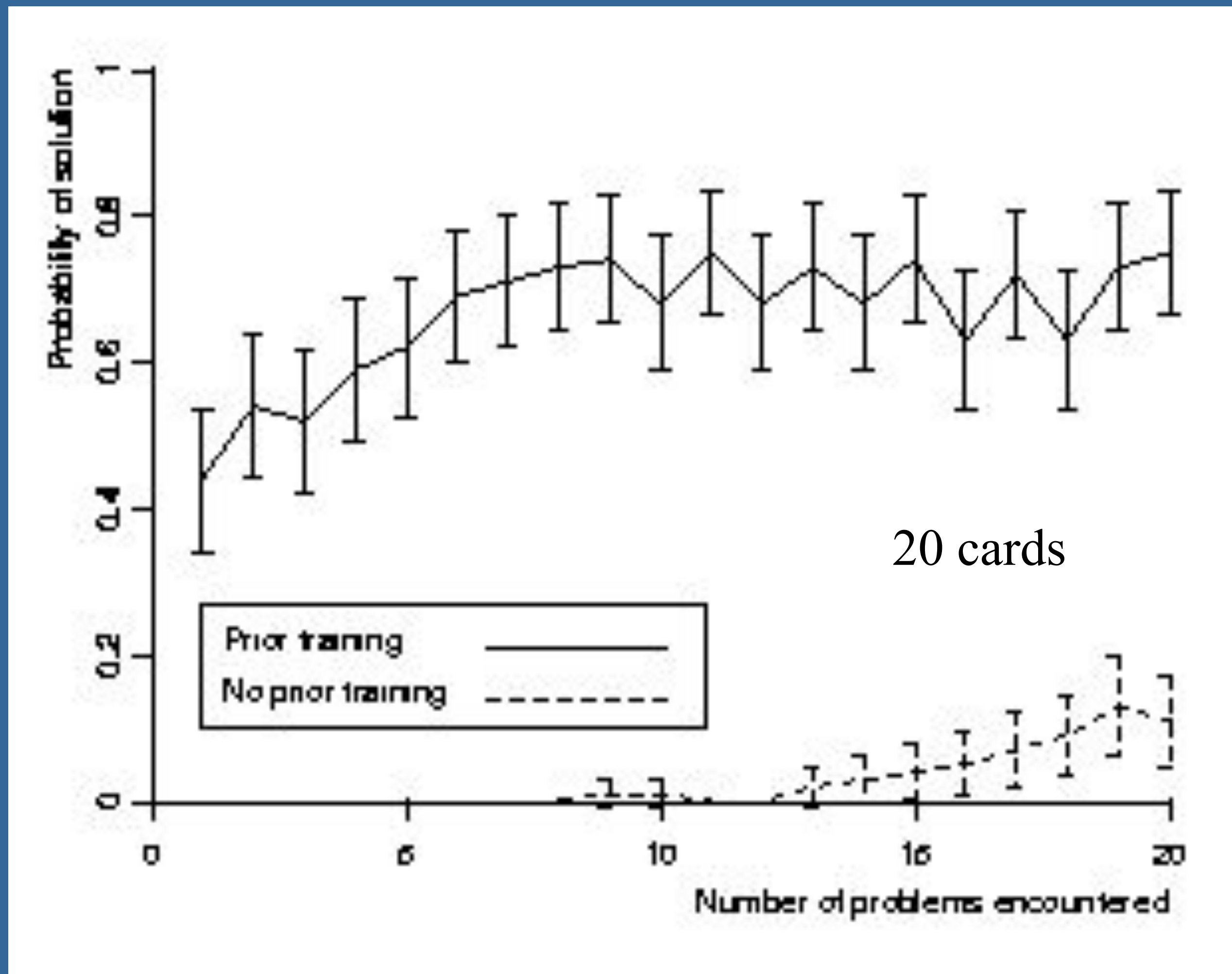
On 16-card FreeCell tasks, prior training aids solution probability.

Transfer Effects in FreeCell



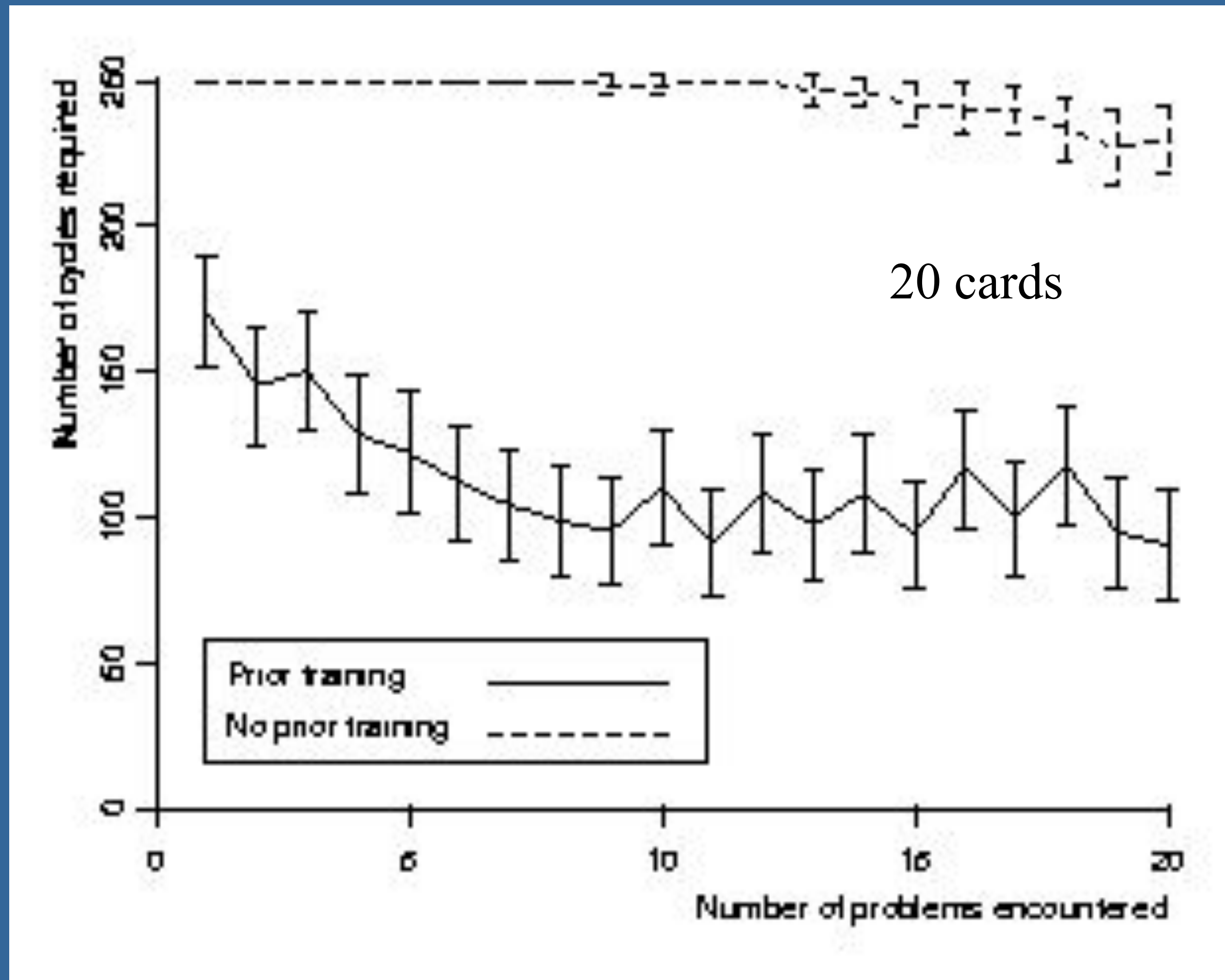
However, it also lets the system solve problems with less effort.

Transfer Effects in FreeCell



On 20-card tasks, the benefits of prior training are much stronger.

Transfer Effects in FreeCell



However, it also lets the system solve problems with less effort.

Where is the Utility Problem?

Many previous studies of learning and planning found that:

- learned knowledge reduced problem-solving steps and search
- but increased CPU time because it was specific and expensive

We have not yet observed the utility problem, possibly because:

- the problem solver does not chain off learned skill clauses;
- our performance module does not attempt to eliminate search.

If we encounter it in future domains, we will collect statistics on clauses to bias selection, like Minton (1988) and others.

Related Work on Planning and Execution

Our approach to planning and execution bears similarities to:

•problem-solving architectures like Soar and Prodigy
•Nilsson's (1994) notion of teleoreactive controllers
•execution architectures that use HTNs to structure knowledge
•Nau et al.'s encoding of HTNs for use in plan generation

Other mappings between classical and HTN planning come from:

- Erol et al.'s (1994) complexity analysis of HTN planning
- Barrett and Weld's (1994) use of HTNs for plan parsing

These mappings are valid but provide no obvious approach to learning HTN structures from successful plans.

Related Research on Learning

Our learning mechanisms are similar to those in earlier work on:

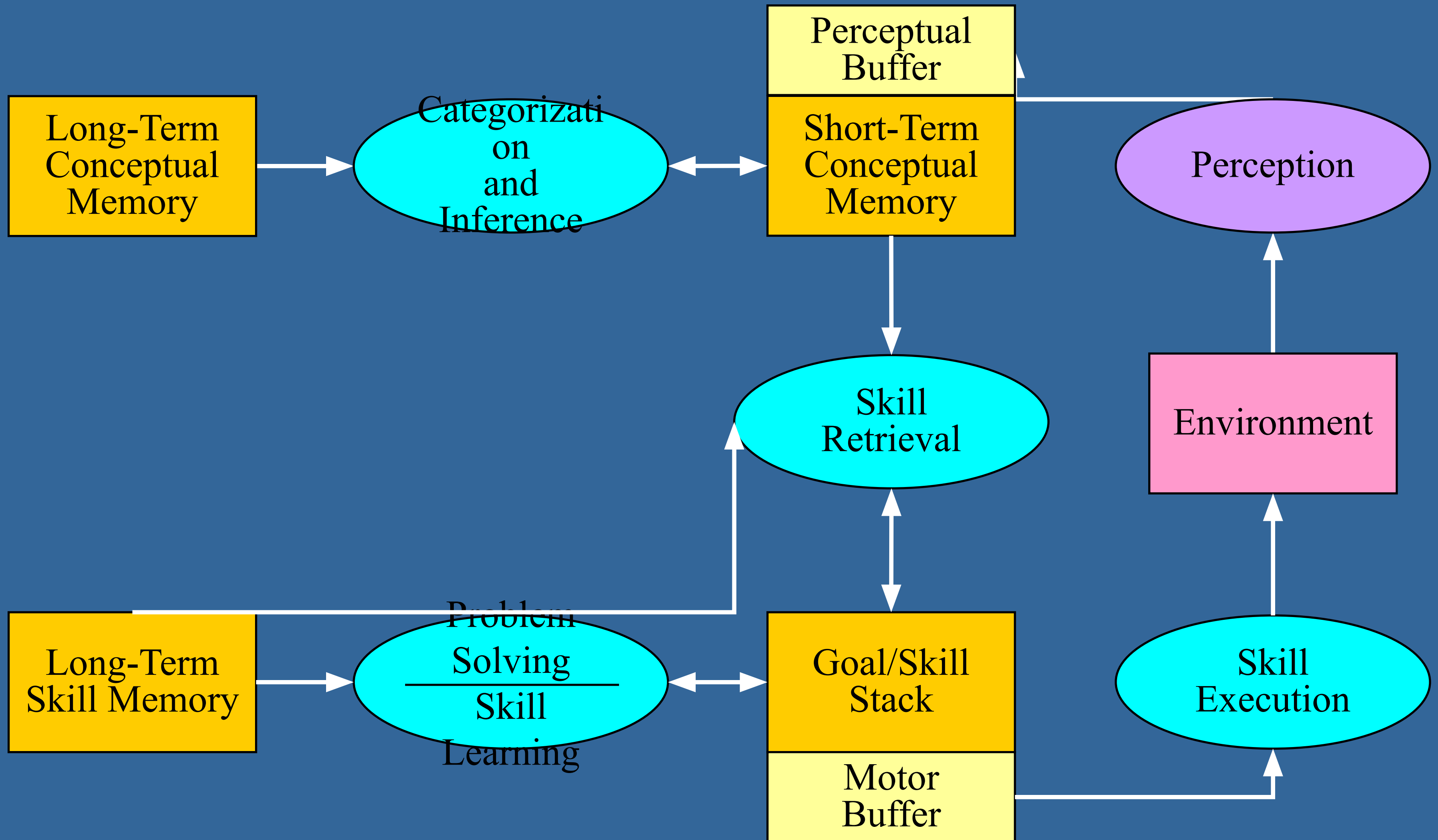
•production composition (e.g., Neves & Anderson, 1981)
•macro-operator formation (e.g., Iba, 1985)
•explanation-based learning (e.g., Mitchell et al., 1986)
•chunking in Soar (Laird, Rosenbloom, & Newell, 1986)

But they do not rely on analytical schemes like goal regression, and their creation of hierarchical structures is closer to that by:

- Marsella and Schmidt’s (1993) REAPPR
- Ruby and Kibler’s (1993) SteppingStone
- Reddy and Tadepalli’s (1997) X-Learn

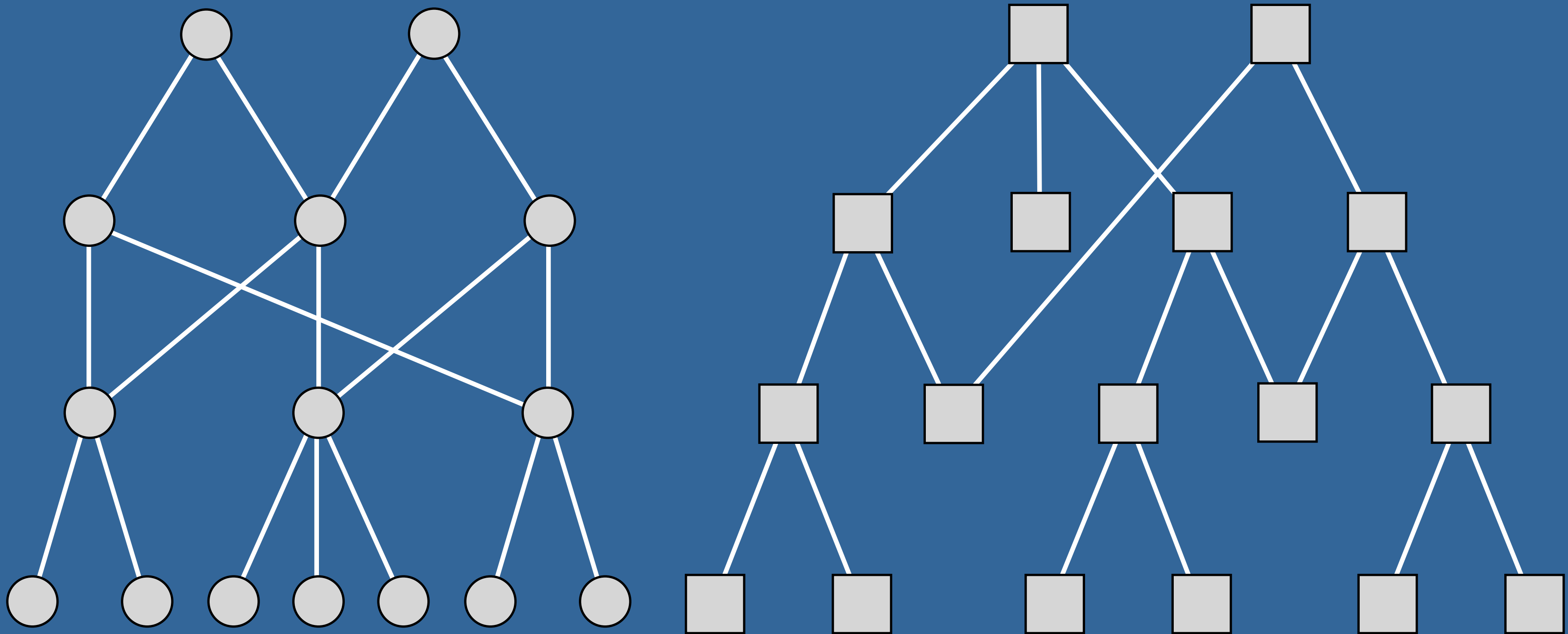
which also learned decomposition rules from problem solutions.

The ICARUS Architecture



Hierarchical Structure of Long-Term Memory

ICARUS organizes both concepts and skills in a hierarchical manner.



concepts

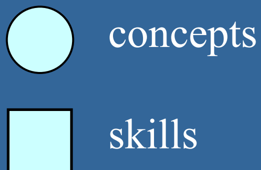
skills

Each concept is defined in terms of other concepts and/or percepts.

Each skill is defined in terms of other skills, concepts, and percepts.

Interleaved Nature of Long-Term Memory

ICARUS interleaves its long-term memories for concepts and skills.



For example, the skill highlighted here refers directly to the highlighted concepts.

Recognizing Concepts and Selecting Skills

ICARUS matches patterns to recognize concepts and select skills.

Concepts are matched bottom up, starting from percepts.

Skill paths are matched top down, starting from intentions.

Directions for Future Work

Despite our initial progress on structure learning, we should still:

- evaluate approach on more complex planning domains;
- extend method to support HTN planning rather than execution;
- generalize the technique to acquire partially ordered skills;
- adapt scheme to work with more powerful planners;
- extend method to chain backward off learned skill clauses;
- add technique to learn recursive concepts for preconditions;
- examine and address the *utility problem* for skill learning.

These should make our approach a more effective tool for learning hierarchical task networks from classical planning.

Concluding Remarks

We have described an approach to planning and execution that:

- relies on a new formalism – *teleoreactive logic programs* – that identifies heads with goals and has single preconditions;
- executes stored HTN methods when they are applicable but resorts to classical planning when needed;
- caches the results of successful plans in new HTN methods using a simple learning technique;
- creates recursive, hierarchical structures from individual problems in an incremental and cumulative manner.

This approach holds promise for bridging the longstanding divide between two major paradigms for planning.

End of Presentation