Reinforcement Learning in Soar

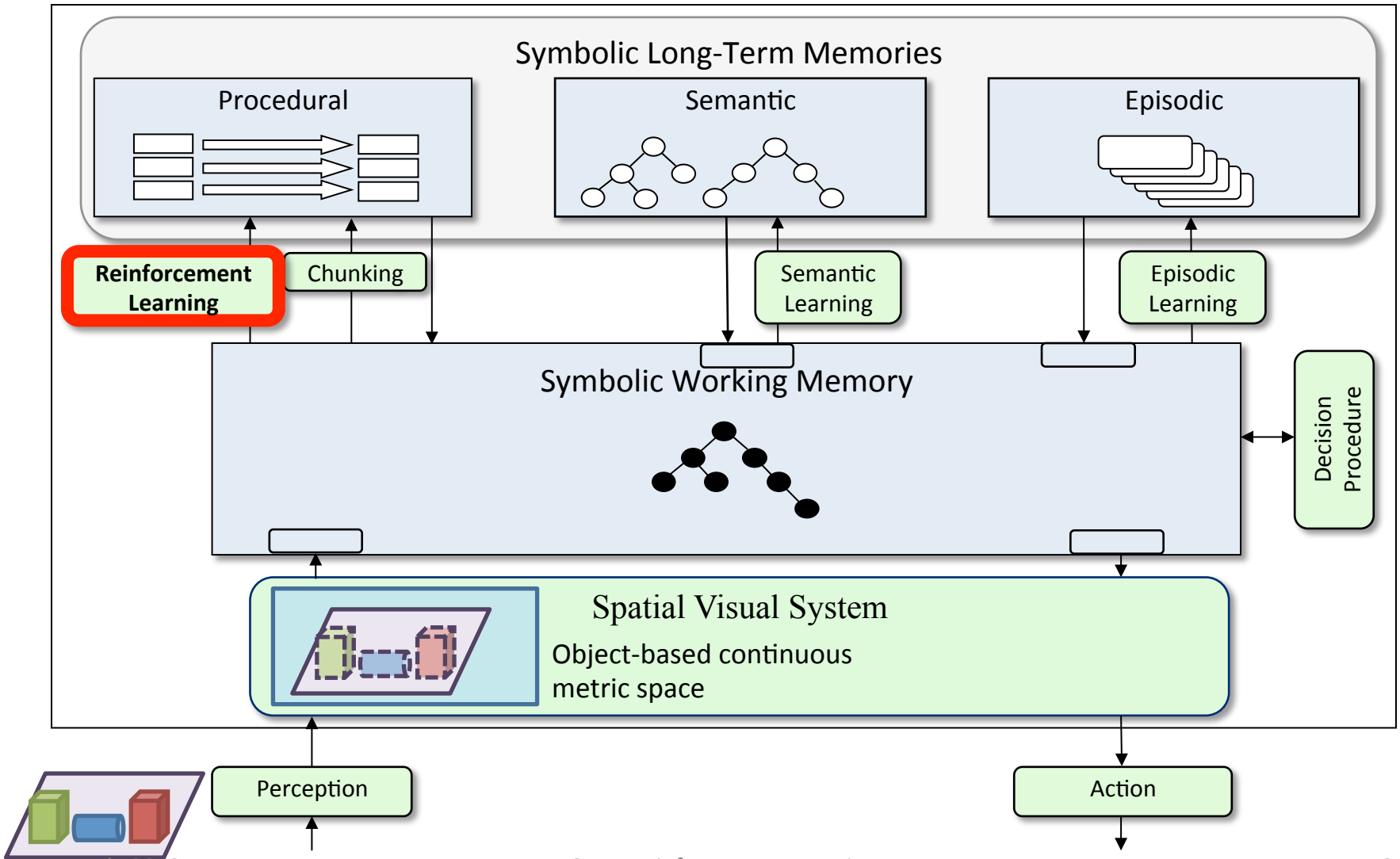
Nate Derbinsky

Disney Research, Boston

Topics

- RL as a learning mechanism
- Architecture & agent design
- Value function structure & initialization
  - Dice game revisited!
- Comparison to ACT-R

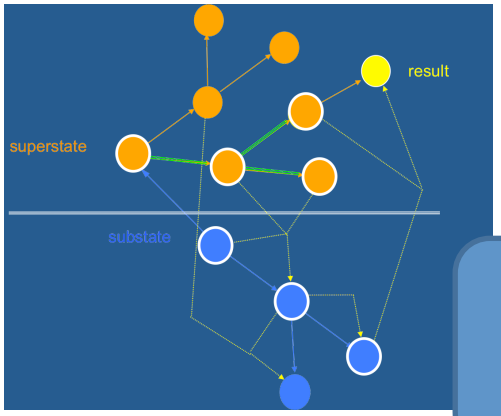
Soar 9


Symbolic Long-Term Memories
Procedural
Semantic
Episodic
Reinforcement Learning
Chunking
Semantic Learning
Episodic Learning
Symbolic Working Memory
Decision Procedure
Spatial Visual System
Object-based continuous metric space
Perception
Action

Methods for Learning Procedural Knowledge

Chunking

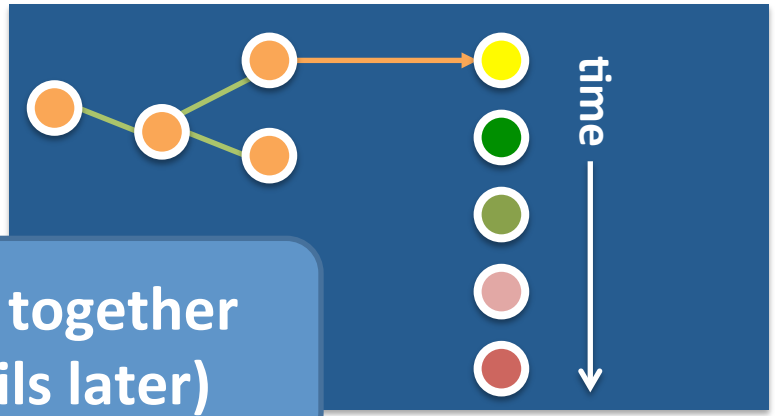
- Converts *deliberation* in substates into *reaction* via rule compilation



- Creates new rules

Reinforcement Learning

- *Tunes* operator numeric preferences to reflect expectation of reward

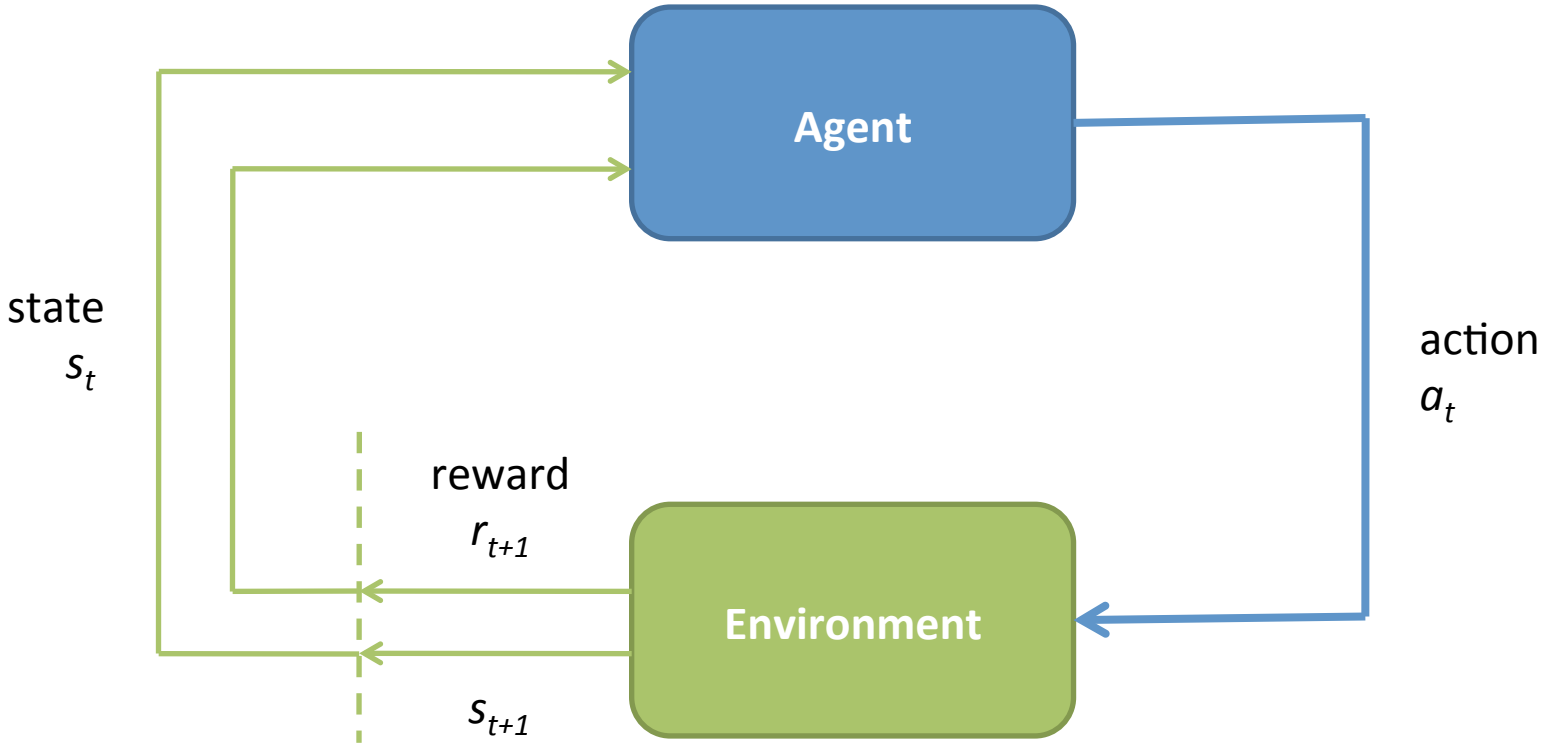


- Updates existing rules

Can be used together (more details later)

RL Cycle

Goal: learn an action-selection policy such as to maximize expected receipt of future reward



Left-Right Demo

1. Soar Java Debugger
2. Source left-right agent

   `Agents/left-right/left-right.soar`

Left-Right Demo

Script

1. `srand 5041231`
2. `step`
3. `run 1 -p`
4. click: `op_pref` tab
   - note numeric indifferents
5. `print left-right*rl*left`
6. `print left-right*rl*right`
7. `run`
   - note movement direction
8. `print left-right*rl*left`
9. `print left-right*rl*right`
10. `init-soar`
11. Repeat from #2 (~5 times)

Left-Right: Takeaways

Reinforcement learning changes rules in procedural memory

- Changes are persistent
- Change affects numeric indifferent preferences, which in turn affects the selection of operators
- Change is in the direction of the underlying reward signal (will discuss this more shortly)

RL -> Architecture & Agent Design

Value function

via RL rules [agent]

Reward

via working-memory structures [architecture, agent]

Policy updates

via Temporal Difference (TD) Learning [architecture]

RL Rules

The RL mechanism maintains Q-values for state-operator pairs in specially formulated rules, identified by syntax

- RHS with a single action, asserting a single numeric indifferent preference with a constant value

```
sp {left-right*rl*left
   (state <s> ^name left-right
              ^operator <op> +)
   (<op> ^name move
         ^dir left)
-->
   (<s> ^operator <op> = 0)
}
```

```
sp {left-right*rl*right
   (state <s> ^name left-right
              ^operator <op> +)
   (<op> ^name move
         ^dir right)
-->
   (<s> ^operator <op> = 0)
}
```

Left-Right Demo

Focus: RL Rules

1. Soar Java Debugger
2. Source left-right agent

   `Agents/left-right/left-right.soar`
3. `print --full --rl`
4. `run`
5. `print --full --rl`
6. `print --rl`

Reward Representation

Each state in WM has a `reward-link` structure

Reward is recognized by syntax

```
(<reward-link> ^reward <r>)
(<r> ^value [integer or float])
```

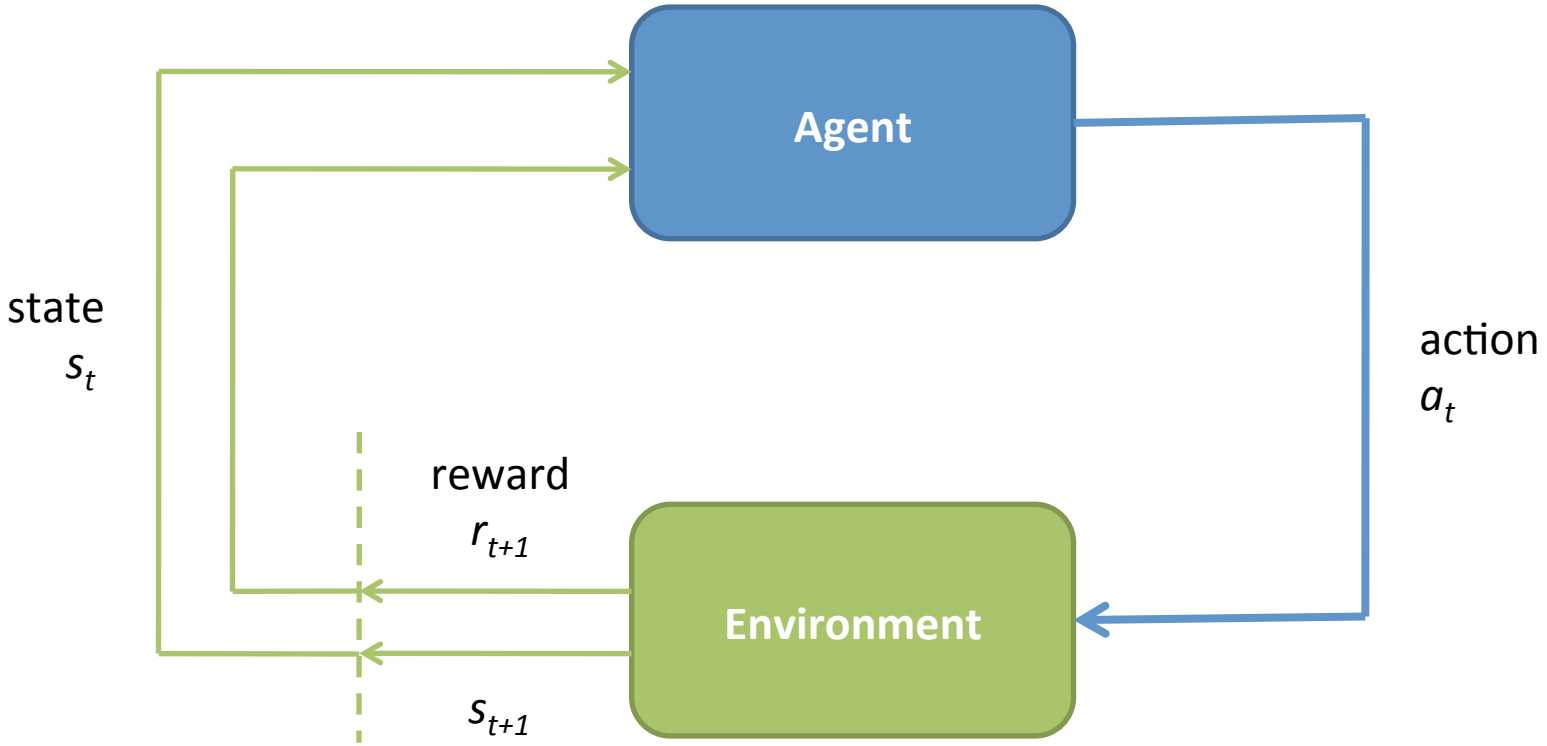
- The reward-link is **not** directly modified by the environment or architecture (i.e. requires agent interpretation/management)
- Reward is collected at the beginning of each *decide* phase
- Reward on a state's reward-link pertains only to that state (more on this later)
- Reward can come from multiple sources: reward values are summed by default

Reward Rule Examples

```
sp {left-right*reward*left
   (state <s> ^name left-right
              ^location left
              ^reward-link <rl>)
-->
   (<rl> ^reward <r>)
   (<r> ^value -1)
}
```

```
sp {left-right*reward*right
   (state <s> ^name left-right
              ^location right
              ^reward-link <rl>)
-->
   (<rl> ^reward <r>)
   (<r> ^value 1)
}
```

RL Cycle



RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d					
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$				
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$			
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		initiate external action(s)
d+1					

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		initiate external action(s)
d+1	$state_{d+1}$ $reward_{d+1}$				

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		initiate external action(s)
d+1	$state_{d+1}$ $reward_{d+1}$	evaluate $operators_{d+1}$			

RL Cycle in Soar

	Input	Propose	Decide	Apply	Output
d	$state_d$	evaluate $operators_d$	select $operator_d$		initiate external action(s)
d+1	$state_{d+1}$ $reward_{d+1}$	evaluate $operators_{d+1}$	select $operator_{d+1}$ update $policy_d$		

RL Updates

- Takes place during *decide* phase, after operator selection
- For all RL rule instantiations (**n**) that supported the *last* selected operator

$$value_{d+1} = value_d + (\delta_d / \mathbf{n})$$

Where, roughly…

$$\delta_d = \alpha[reward_{d+1} + \Upsilon(q_{d+1}) - value_d]$$

Where…

- α is a parameter (learning rate)
- Υ is a parameter (discount rate)
- q_{d+1} is dictated by learning policy
  - On-policy (SARSA): value of selected operator
  - Off-policy (Q-learning): value of operator with maximum selection probability

Value Function

Issues

Structure

- What features comprise RL-rule conditions (tradeoff: convergence speed vs. performance)
- High dimensionality -> computationally infeasible

Initialization

- Quality estimates may bootstrap agent performance and reduce time to convergence

Value Function
Approach

- Tools for rule generation given a known/ simple state space

- Selection Problem Space + Chunking!
  - Agent deliberately reasons about q-value estimates (incl. search, memories, etc.)
  - Estimate = initial numeric-indifferent preference
  - Chunks = incremental value function (supports tiling + coarse coding)

Reinforcement Learning in Dice Game

Bid: 6[4] .3

Challenge .5

Bid: 6[4] & Push: 1, 4 .4

Win Challenge

1.0

Value Function
(S,A) -> EV

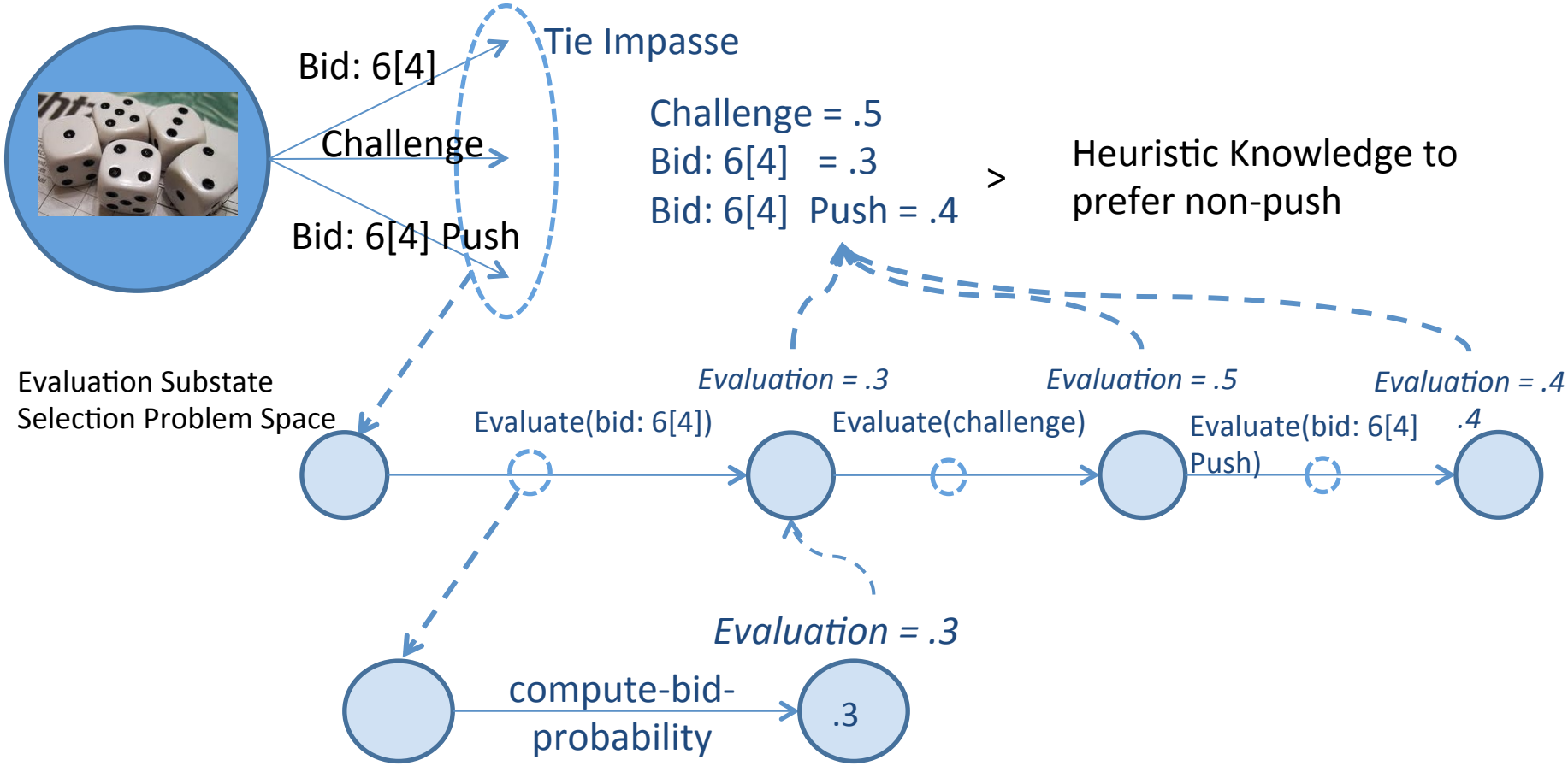
(3 4's, 6[4]) -> .3

(..., Challenge -> .6

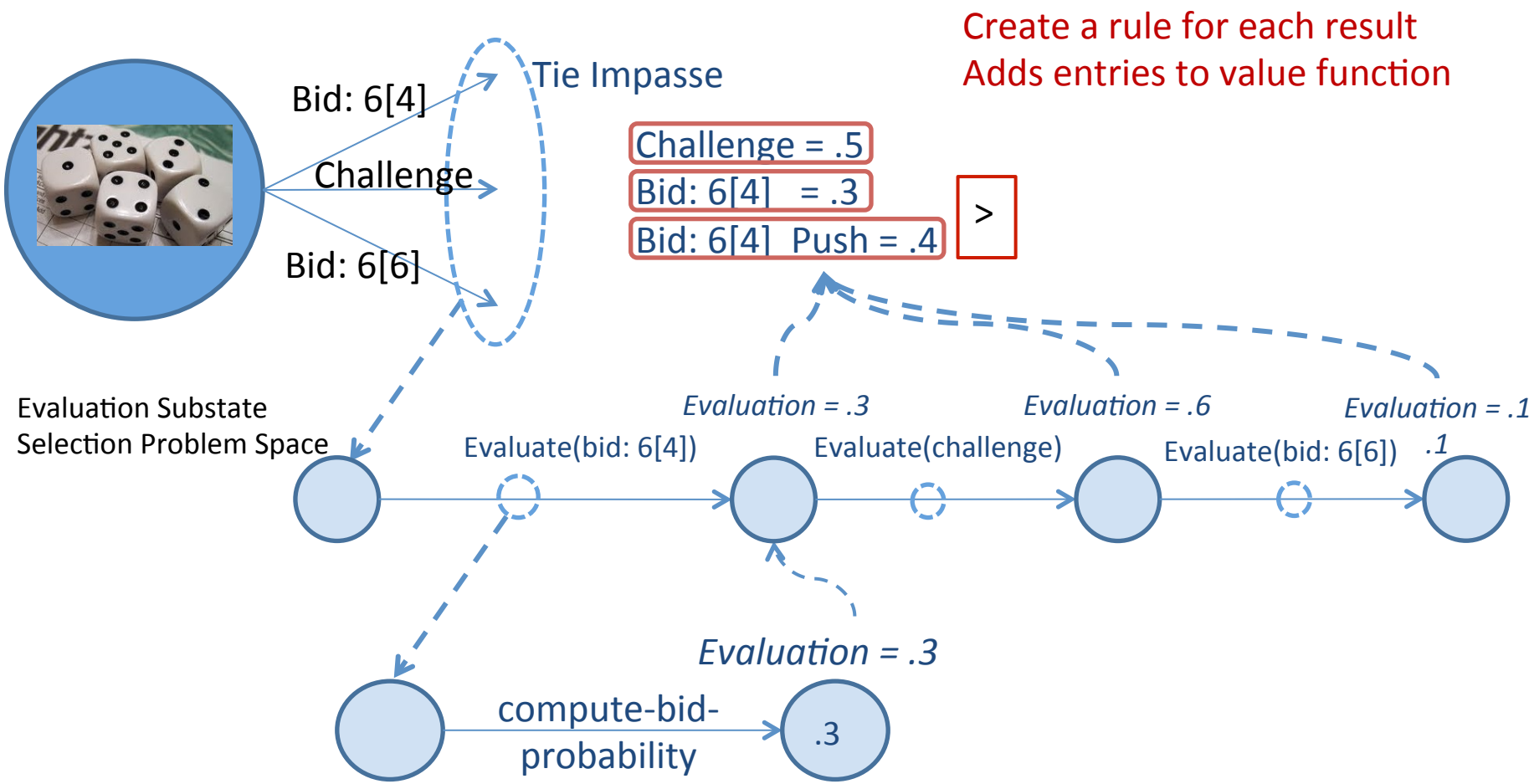
(4 6's, 6[4] & 1,4) -> .4

Value Function Size
~1,000,000 – 25,000,000

How does the work in Soar?



Create Value-Function Entry



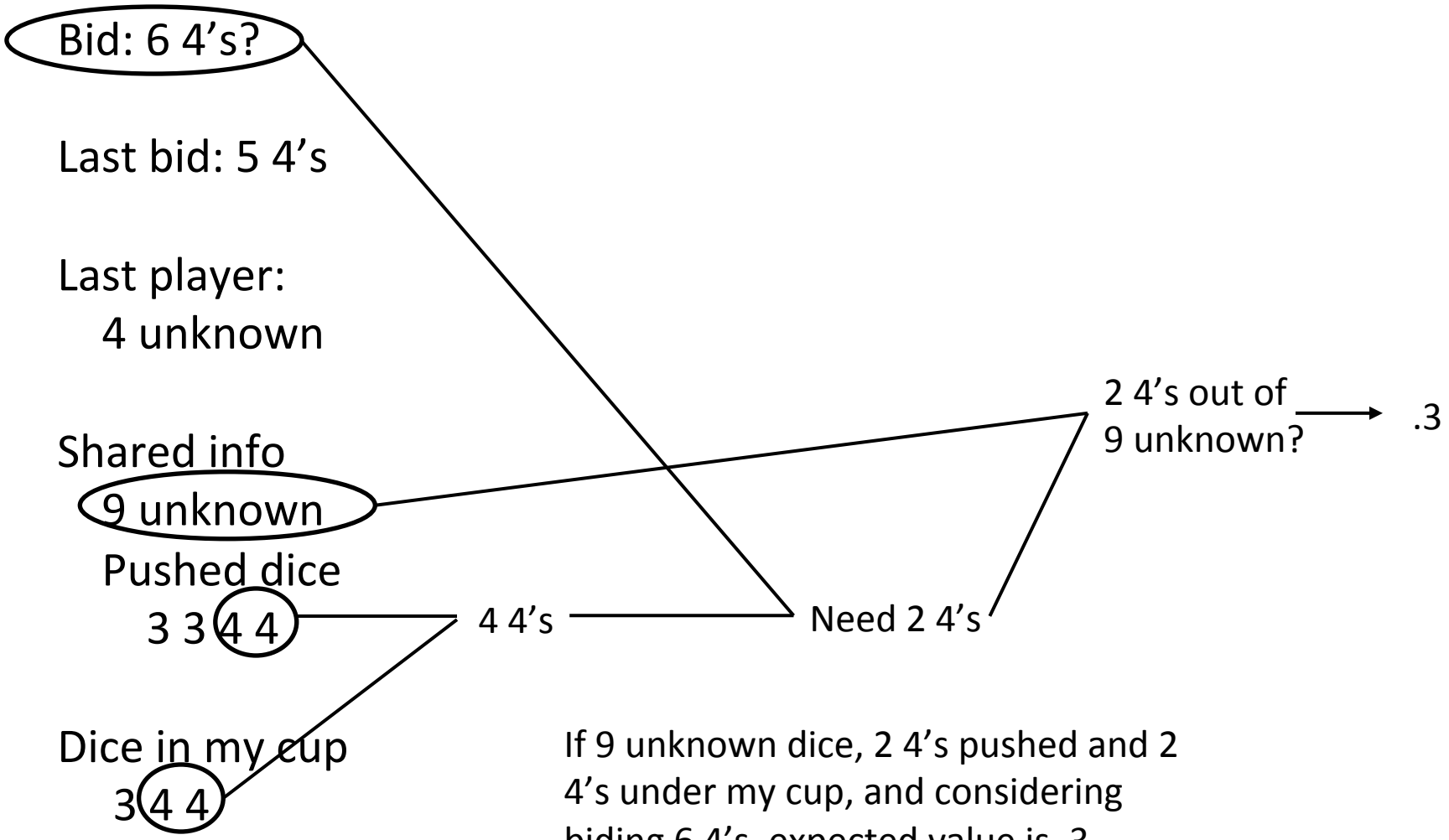
Chunking over Substate Processing

- For each preference created in the substate, chunking (EBL) creates a new rule
  - Actions are numeric or symbolic preferences
  - Conditions based on working memory elements tested in substate

- Reinforcement learning then tunes RL rules based on experience

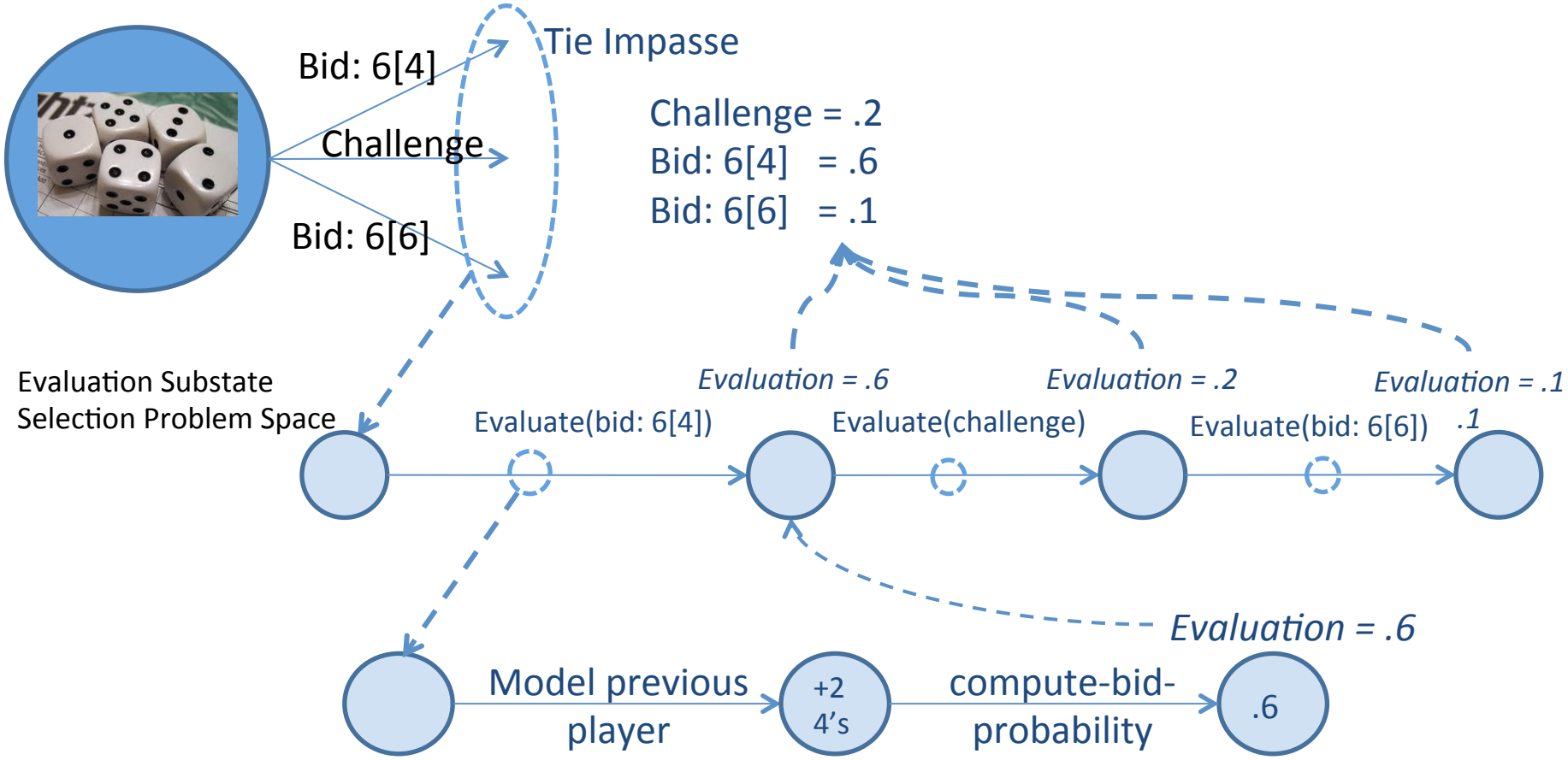
Learning RL-rules

Using only probability calculation



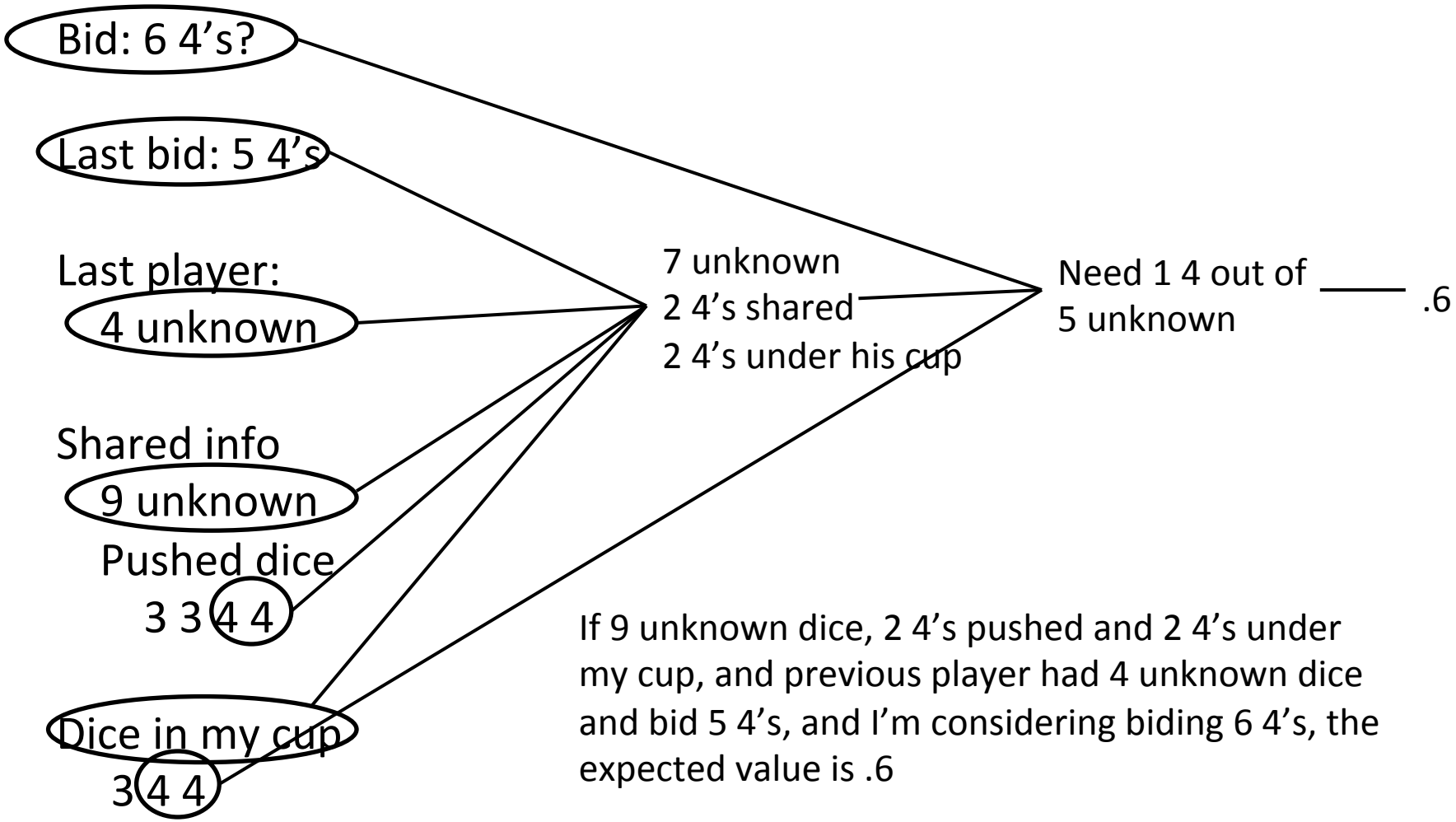
If 9 unknown dice, 2 4's pushed and 2 4's under my cup, and considering biding 6 4's, expected value is .3

Using Additional Background Knowledge (Model)



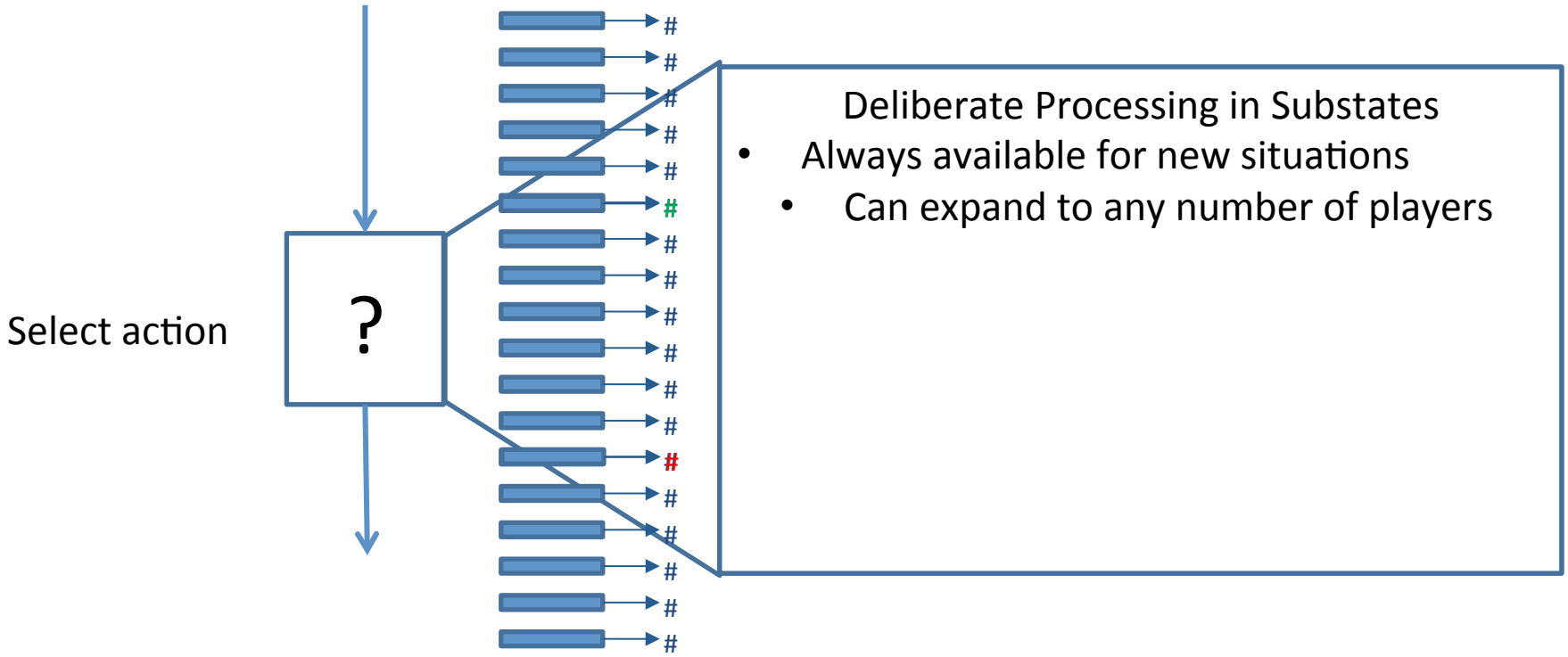
Learning RL-rules

Using probability and model



If 9 unknown dice, 2 4's pushed and 2 4's under my cup, and previous player had 4 unknown dice and bid 5 4's, and I'm considering biding 6 4's, the expected value is .6

Two-Stage Learning



RL rules updated based on agent's experience

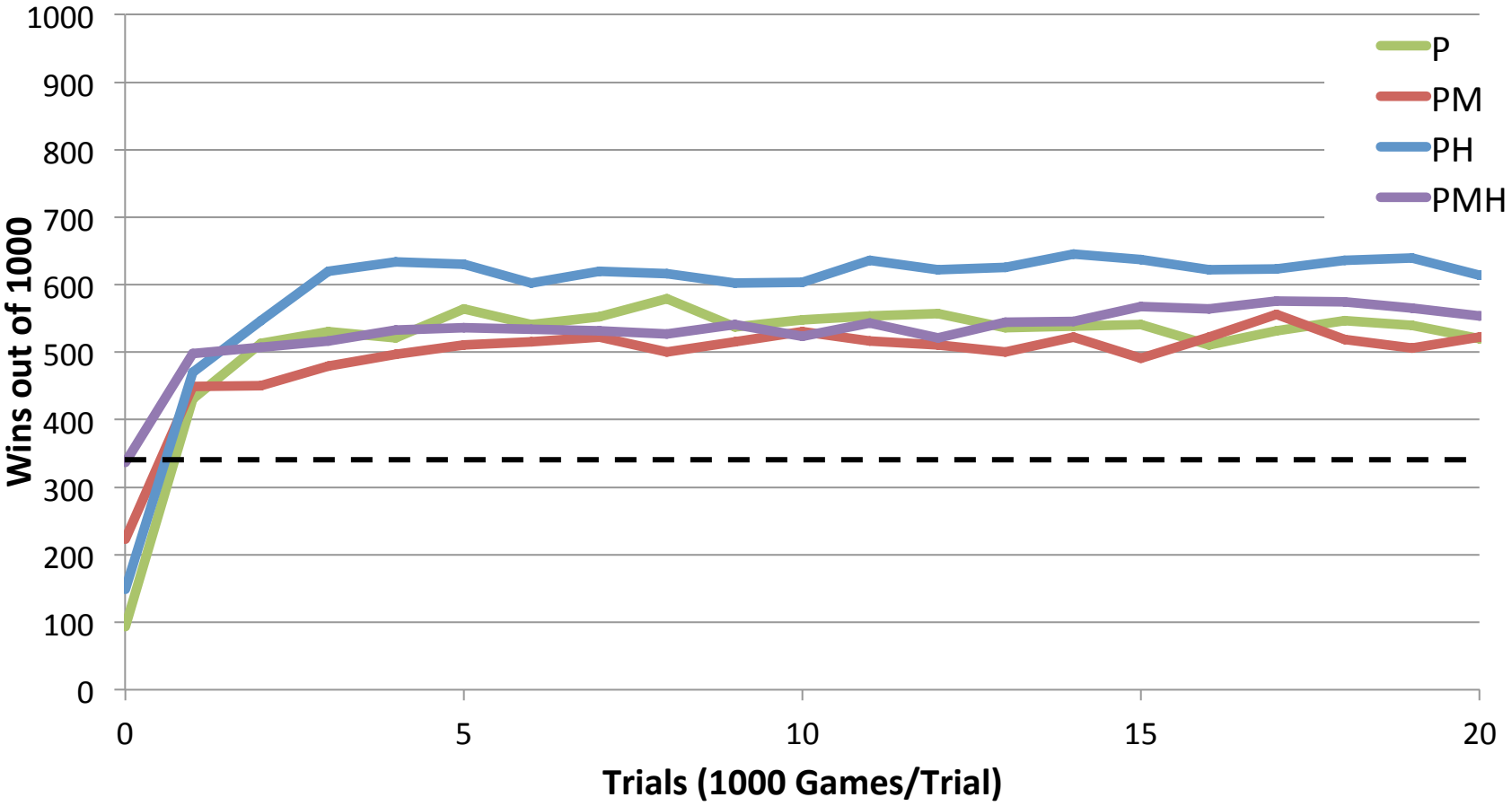
Research Questions

- Does this approach to RL help?
  - Do agents improve with experience?
  - Can learning lead to better performance than the best hand-coded agent?
- Does initialization of RL rules improve performance?
- How does background knowledge affect rules learned by chunking and how do they affect learning?

Evaluation of Learning

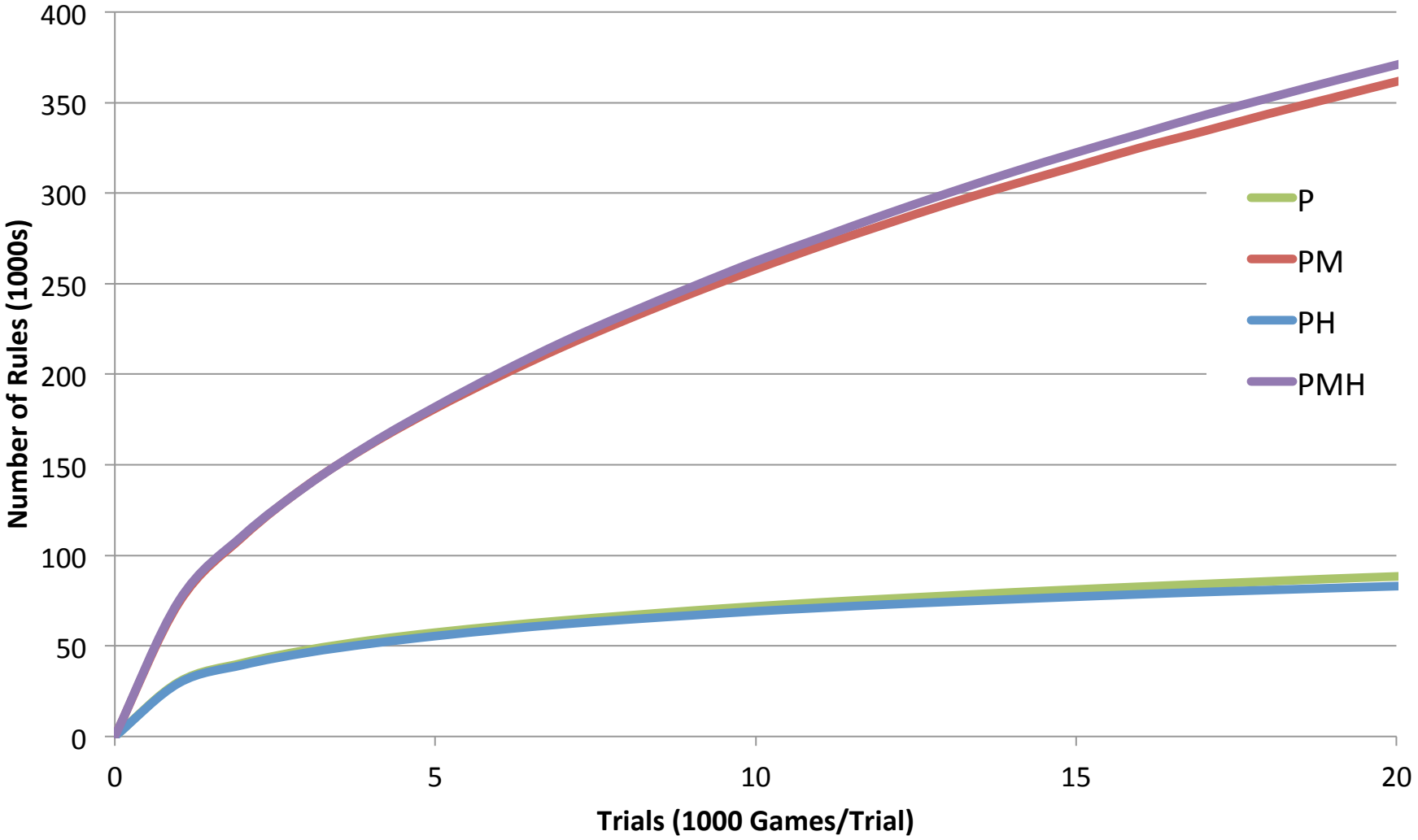
- 3-player games
  - Against best non-learning agent
    - Heuristics and opponent model
  - Alternate 1000 game blocks of testing and training
- Metrics
  - # of games won
  - Speed of learning
- Agent variants:
  - **P:** baseline using probabilities
  - **H:** with heuristics
  - **M:** with opponent model
  - **MH:** with opponent model and heuristics

Learning Agent Comparison

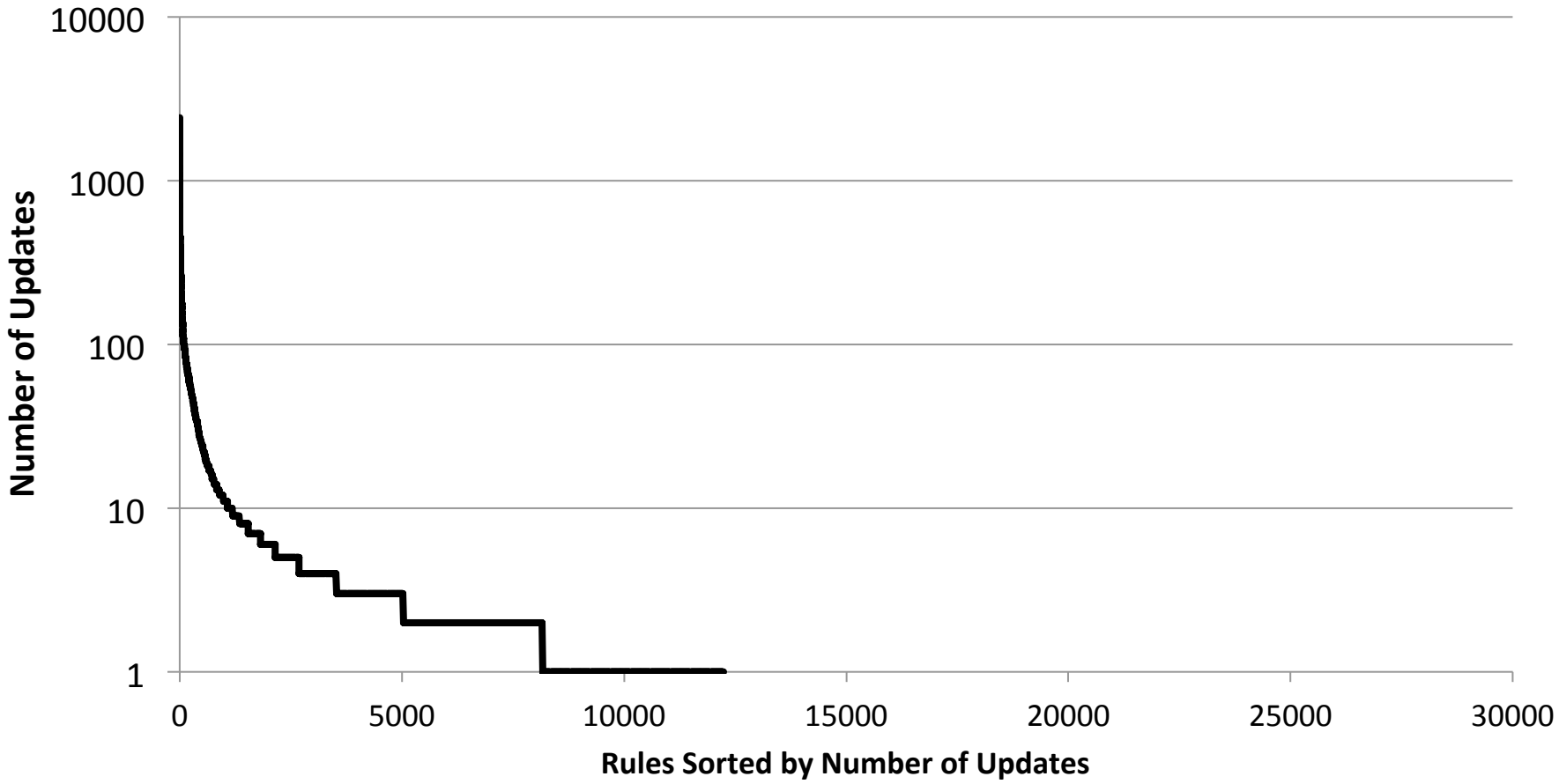


- H and M give better initial performance than P.
- With learning, all agents do significantly better than hand coded.
- H alone speeds learning (smaller state space).
- M slows learning (much larger state space).

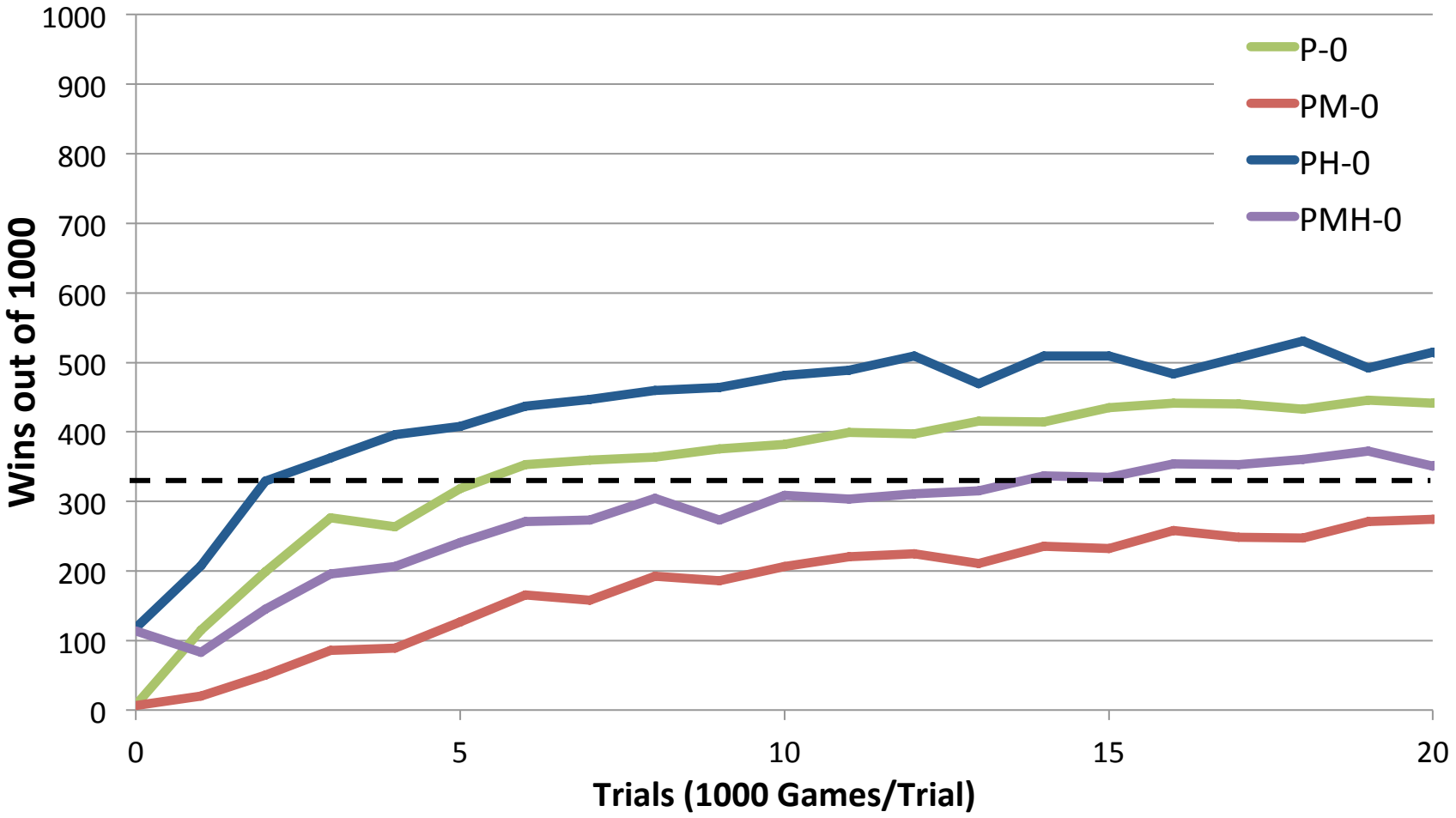
Number of Rules Learned



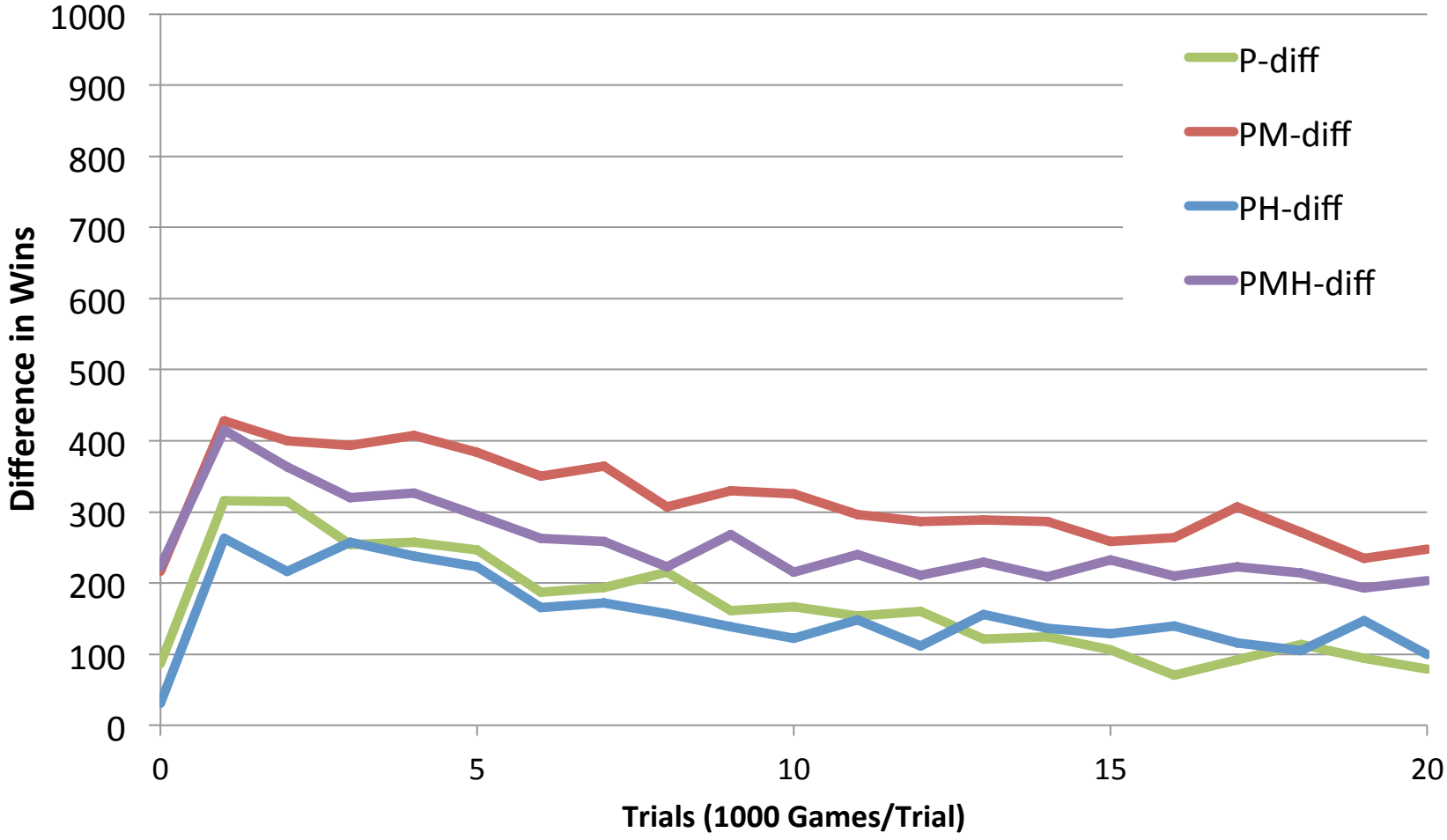
Rules Sort by Updates



Learning Agents with Initial Values = 0



Difference Between Initialized and Uninitialized



Comparison to ACT-R

	New Rules	Existing Rules
ACT-R	Compilation • Combines rules that fire in sequence • Affects model timing	Utility • Conflict resolution: Each rule has utility value (U); if multiple match, the highest U fires • Reward via programmatic hooks, rules (learn?) • "Flat" learning via T-1
Soar	Chunking • Generalizes justifications of result(s) from substate reasoning (potentially many decisions/rule firings) • Deliberation -> reaction (potentially fewer decisions, different reasoning paths)	Reinforcement Learning • Numeric indifferent preferences may impact operator selection (resulting in potentially numerous application rules firing) • Reward via rule(s) that can be learned • Hierarchical TD learning

Thank You :)

Questions?