Reward Augmentation in Reinforcement Learning for Testing Distributed Systems

(paper presentation)

Bijayan Ray
MCS202403

May 7, 2025

Contents

Example of distributed system testing I

- Concurrent Programming is a technique where two or more processes start to run in an interleaved manner through context switch and complete in an overlapping time period by sharing the resource.
- Multi-threaded programs with shared memory: Atomic actions are read and write (possibly user defined) to shared memory, synchronization is done by locking of shared resource
- Distributed systems: No shared memory, actor/machines communicate with each other via message passing, action is triggered via message, synchronization also done by message passing.
- Control non determinism: Control non determinism occurs when more than one machine becomes enabled.
- Data non determinism: Data non determinism is used to generate uncontrolled scalar values e.g. model for handling user given input.

Example of distributed system testing II

- Stateful techniques fully model the system's state space (agent states and network) and explore reachable states through explicit methods like BMC or SPIN.
- Stateless techniques, such as Controlled Concurrency Testing (CCT), dynamically control and explore interleavings without modeling the full state space, aiming to enhance coverage by systematically testing different action schedules.

Example of distributed system testing III

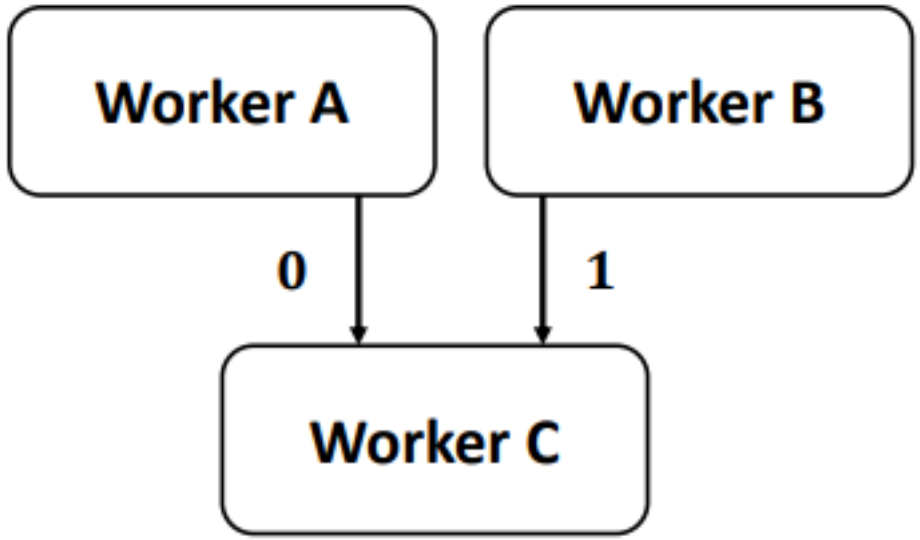


Figure: Simple messaging example with 3 workers

- Take the program that involves two workers, A and B, sending messages to a third worker, C.
- When A sends the message its denoted by 0 and when B sends the message its denoted by 1.

Example of distributed system testing IV

- Worker C has a constant string $\eta \in \{0,1\}^*$ and a counter m initialized to 0.
- If the i-th message (0 when A sends it and 1 when B sends it) received by C matches the i-th character of η, m is incremented. Otherwise, m is set to -1 and is never updated again.
- The program reaches a bad configuration state if $m = n$.
- We *test* if the program reach a bad configuration.
- Note that for any string η of length n, there is exactly one way of scheduling between A and B that causes C to raise an error after n messages.
- The effectiveness of each scheduler is measured by the B-% value, which is defined as the percentage of buggy program runs in a sufficiently large number of runs.

Example of distributed system testing V

- In case of Random scheduling, we schedule A and B randomly to message C, thereby C keeps on getting a random string in $\{0,1\}^n$ after n messages.
- For any string η of length n the Random scheduler has $\frac{1}{2^n}$ chance of producing that buggy string, as it must choose between workers A and B exactly according to η.
- Although the Random scheduler has similar B-% values for different strings, its effectiveness is poor due to the exponential dependence on the string length.

Example of distributed system testing VI

- The QL scheduler exposes the bug for each string with a much higher B-% compared to the other two schedulers.
- QL takes advantage from observing the state of the program (counter m) during exploration, and optimizes for coverage.
- For this problem, we set it up to observe just the value of counter m of worker C and as m is set to -1 on a wrong scheduling choice, it is forced to learn scheduling decisions that keep incrementing m, which leads to the bug.

Problem statement I

Finding optimal search policies in distributed systems as a reinforcement learning (RL) problem and propose new algorithms for biased exploration of the program state space towards semantically significant states for better coverage and bug detection.

Agent and Environment RL Setup I

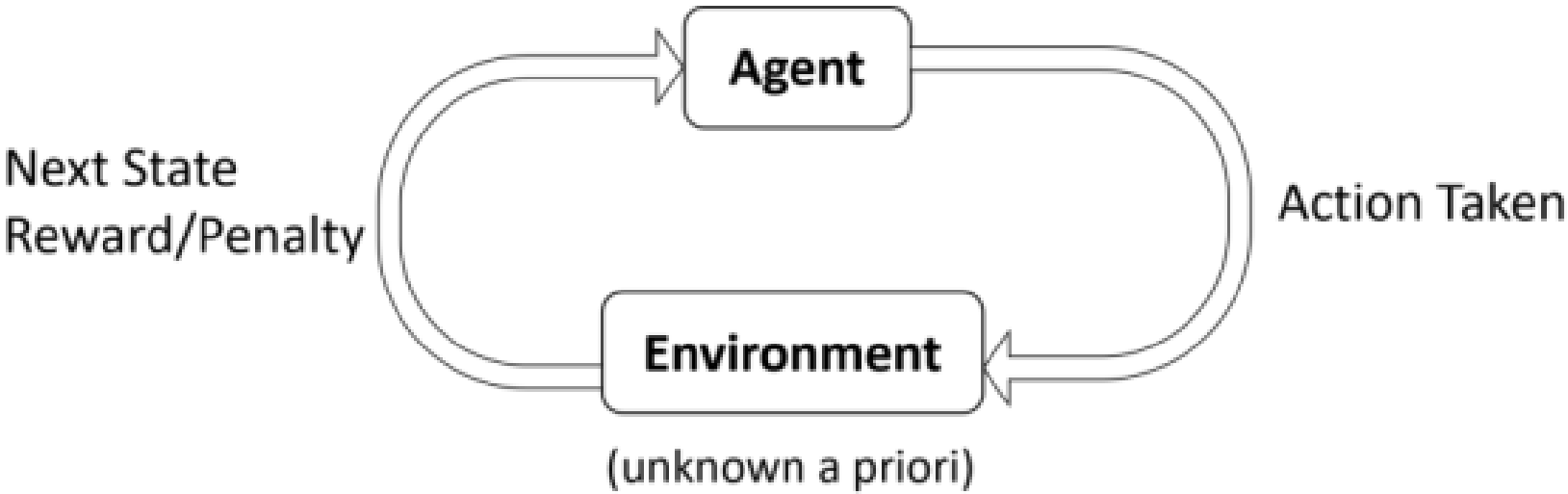


Figure: Reinforcement Learning problem

- The *environment* is a program whose configurations form the set Σ.
- A user-defined abstraction function $H : \Sigma \to S$ maps program configurations to abstract states, representing the *agent's* observation of the environment.

Agent and Environment RL Setup II

- The *reward* function $R : S \times A \to \mathbb{R}$ is fixed as a constant value of -1 for all state-action pairs since that ensures the Q-values to decrease over time for seen state-action pairs thus increasing exploration (for maximizing Q-values).
- The *agent* uses a Q-learning based scheduler that selects actions using a softmax policy over Q-values.
- The *agent* maintains a partial Q-value map $Q : S \times A \to \mathbb{R}$, initialized to 0 for unseen state-action pairs.
- The *agent's* action selection is based on Q-values and it updates the Q-values based on abstracted past schedules.

Main ideas of RL based approaches I

- NegRL: The reward function is chosen as -1 to increase exploration.
- BonuxMaxRL: Keeps track of number of visits to a state while exploration.
- WayPointRL: Biases exploration towards potentially buggy states.

NegRL I

Algorithm 1: GetNext-QL

Input: Set of actions $\{a_1, \ldots, a_n\}$, Configuration σ

$s \leftarrow H(\sigma)$;
foreach $a \in \{a_1, \ldots, a_n\}$ **do**
  if $Q(s, a)$ *is undefined* **then**
    $Q(s, a) \leftarrow 0$ x; // Initialize Q-value of new (s, a) pair to 0
$D \leftarrow \langle\rangle$; // Probability distribution over actions
foreach $i \in \{1, \ldots, n\}$ **do**
  $D(i) \leftarrow \frac{e^{Q(s,a_i)}}{\sum_{j=1}^{n} e^{Q(s,a_j)}}$;
$i \leftarrow \text{Sample}(D)$;
return a_i;

Algorithm 2: PrepareNext-QL

Input: Schedule $\ell = \langle \sigma_0 \xrightarrow{a_1} \sigma_1, \ldots, \sigma_{n-1} \xrightarrow{a_n} \sigma_n \rangle$

$\hat{\ell} \leftarrow H(\ell)$; // Set $\hat{\ell}$ to $\langle s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \rangle$
foreach $i \in \{n, \ldots, 1\}$ **do**
  $\text{maxQ} \leftarrow \max_a Q(s_i, a)$;
  $Q(s_{i-1}, a_i) \leftarrow (1 - \alpha) \cdot Q(s_{i-1}, a_i) + \alpha \cdot [R(s_{i-1}, a_i) + \gamma \cdot \text{maxQ}]$;

NegRL II

- Take H be a user-defined function $H : \Sigma \to S$, where Σ is the set of program configurations and S is a set of abstract states.
- $H(\sigma)$ represents the observation the agent makes when the environment is in configuration σ.
- H defines an abstraction over the configuration space, implemented as a hashing function over part of the program's configuration.
- The reward function $R : S \times A \to \mathbb{R}$ is fixed to be a constant -1, i.e., $R(s, a) = -1$ for all (s, a).
- The constant negative reward ensures Q-values decrease over time for seen state-action pairs.
- For a schedule $\ell = \langle \sigma_{\text{init}} \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \sigma_n \rangle$, $H(\ell)$ denotes the abstracted schedule where each σ_i is replaced by $s_i = H(\sigma_i)$.
- The QL scheduler is parameterized by H and uses it in its *GetNext* and *PrepareNext* procedures.

NegRL III

- It maintains a partial map $Q : S \times A \to \mathbb{R}$ representing Q-values for abstract state-action pairs.
- *GetNext* takes input configuration σ and n enabled actions.
- It computes $s = H(\sigma)$, and initializes $Q(s, a)$ to 0 for any action a not yet in Q.
- A probability distribution D is computed over actions using the Softmax policy based on $Q(s, a)$ values.
- An action is selected by sampling from D and returned.
- *PrepareNext* updates Q-values based on the abstracted previous schedule $\hat{\ell}$.
- For each (s_{i-1}, a_i) in $\hat{\ell}$, Q-values are updated.
- The Softmax policy then reduces the probability of selecting actions that lead to known states, encouraging exploration of new states.

BonusMaxRL I

Algorithm 3: BonusMaxRL: Positive reward based exploration algorithm

Input: α, γ, ϵ

def `init` *()*:
  $Q(s,a) \leftarrow 1$, $V(s,a) \leftarrow 0$ for all $s \in S$, $a \in A$

def `newEpisode` *(_)*:
  trace $\leftarrow$ [ ] ; `// reset the trace`

def `pick` *(s, actions)*:
  $x \sim \mathcal{U}(0,1)$; `// sample uniformly at random from (0,1)`
  if $x < \epsilon$ **then**
    return $a \sim \mathcal{U}\{\text{actions}\}$; `// uniform random action`
  else
    return $\arg\max_a Q(s,a)$

def `recordStep` *(state, action, newState, _)*:
  trace $\leftarrow$ append(trace, $(state, action, newState)$)

def `processEpisode` *()*:
  for $i = length(trace), \ldots, 1$ **do**
    $(s, a, s') \leftarrow \text{trace}[i]$
    $t \leftarrow V(s,a) + 1$
    $V(s,a) \leftarrow t$
    $r \leftarrow \frac{1}{t}$
    if $i < length(trace)$ **then**
      $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \max\left(r, \gamma \cdot \max_{a'} Q(s',a')\right)$
    else
      $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \max(r, 0)$

BonusMaxRL II

- The algorithm implements the *BonusMaxRL* exploration policy, with hyperparameters α, γ, and ϵ as input.
- The action-selection mechanism *pick* uses the standard ϵ-greedy strategy:
  - With probability ϵ, a random action is chosen.
  - With probability $1 - \epsilon$, the action with the highest Q-value is selected.
- The *processEpisode* function updates the policy based purely on exploration bonuses, as no external rewards are provided.
- The policy maintains a visit count table $V(s, a)$ for each state-action pair.
- The internal exploration reward is computed as $\frac{1}{t}$, where $t = V(s, a)$ is the *number of visits* to (s, a).
- This reward is high for new states and decreases as the pair is revisited more often.

BonusMaxRL III

- For every transition (s, a, s'), the visit count $V(s, a)$ is incremented.
- The Q-value update rule is given by:

$$Q(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot \max\left(r, \gamma \cdot \max_{a'} Q(s', a')\right)$$

- This update differs from traditional Q-learning by using max instead of a sum, focusing only on the best reachable future state.
- The update rule favors transitions that lead to new states and devalues over-visited states, pushing Q-values toward 0 over time in stale regions.
- This method does not guarantee optimal policy learning in environments with explicit reward functions.

BonusMaxRL IV

- In particular, BonusMaxRL may favor shorter paths that reach novelty faster, even if longer paths have higher cumulative rewards.
- Despite potential sub-optimality, BonusMaxRL is effective in distributed system testing, as it aligns with the goal of aggressive state exploration.

WayPointRL I

Algorithm 4: WaypointRL: `init`, `newEpisode`, and `pick` methods

Input: predicates $= \{\text{pred}_1, \ldots, \text{pred}_n\}$, oneTime $\in \{\top, \bot\}$, α, γ, ϵ

```
1  def init ():
2      for i = 1...n do
3          Q_i(s,a) ← 1, V_i(s,a) ← 0 ;                    // for all s ∈ S, a ∈ A
4  def newEpisode (initialState):
5      trace ← [ ], reached ← ⊥
6      for i = n...1 do
7          if predicate_i(initialState) = ⊤ then
8              activePredicate ← i
9              break
10 def pick (s, actions):
11     x ~ U(0,1)
12     if x < ε then
13         return a ~ U(actions)
14     else
15         i ← activePredicate
16         return arg max_a Q_i(s,a)
```

WayPointRL II

Algorithm 5: WaypointRL: `recordStep` method

Input: predicates $= \{\text{pred}_1, \ldots, \text{pred}_n\}$, oneTime $\in \{\top, \bot\}$, α, γ, ϵ

```
1  def recordStep (s, a, s', _):
2      if reached = ⊥ then
3          for i = length(predicates) ... 1 do
4              if predicate_i(s') = ⊤ then
5                  nextActivePredicate ← i
6                  break
7          if nextActivePredicate = n ∧ oneTime = ⊤ then
8              reached ← ⊤
9          else
10             nextActivePredicate ← n
11     trace ← append(trace, (s, a, s', activePredicate, nextActivePredicate))
12     activePredicate ← nextActivePredicate
```

WayPointRL III

Algorithm 6: WaypointRL: `processEpisode` method

```
1  def processEpisode ():
2      for i = 1 ... length(trace) do
3          (s, a, s', p, p') ← trace[i]
4          if p = n then
5              reachedFinal ← ⊤, reachedStep ← i
6              break
7      for i = length(trace) ... 1 do
8          (s, a, s', p, p') ← trace[i]
9          t ← V_p(s, a) + 1
10         V_p(s, a) ← t
11         explR ← 1/t ;                                  // visits-based bonus
12         if p = p' ∨ p = n then
13             if i < length(trace) then
14                 Q_p(s, a) ← (1−α)·Q_p(s, a)+α·max(explR, γ·max_a' Q_p(s', a'))
15             else
16                 Q_p(s, a) ← (1 − α) · Q_p(s, a) + α · max(explR, 0)
17         else
18             if p' > p then
19                 progR ← 2
20             else
21                 progR ← 0
22             if reachedFinal then
23                 d ← reachedStep − i − 1
24                 finalR ← γ^d · 2
25             Q_p(s, a) ← (1 − α) · Q_p(s, a) + α · max(explR, γ · (progR + finalR))
```

WayPointRL IV

- WaypointRL uses a *sequence of predicates* $\{\text{pred}_1, \ldots, \text{pred}_n\}$ to guide exploration. pred_n defines the target space, while pred_1 is always true.
- A separate exploration Q-table is maintained for each predicate. At each timestep, the highest-indexed true predicate becomes the active predicate p.
- The agent selects actions using the Q-table of p and stores transitions (s, a, s', p, p') in an episode trace, where p' is the active predicate in s'.
- The *processEpisode* method updates Q-values using the trace. It:
  - checks if the final predicate pred_n was reached and stores the index,
  - performs backward updates for each step (s, a, s', p, p') in the trace.

WayPointRL V

- If $p = p'$, it uses the BonusMaxRL update rule:

$$Q_p(s,a) \leftarrow (1-\alpha)Q_p(s,a) + \alpha \cdot \max(\text{explR}, \gamma \cdot \max_{a'} Q_p(s',a'))$$

where $\text{explR} = \frac{1}{t}$ with t being the visit count of (s,a).

- If $p \neq p'$, and $p' > p$, a progress bonus $\text{progR} = 2$ is added. If the final predicate was reached later in the trace, an additional reward is given:

$$\text{finalR} = \gamma^d \cdot 2 \quad \text{where } d = \text{reachedStep} - i - 1$$

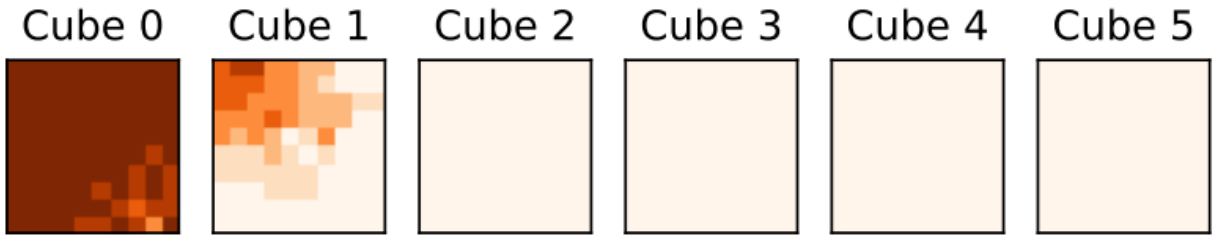
- The final update in this case becomes:

$$Q_p(s,a) \leftarrow (1-\alpha)Q_p(s,a) + \alpha \cdot \max(\text{explR}, \gamma \cdot (\text{progR} + \text{finalR}))$$

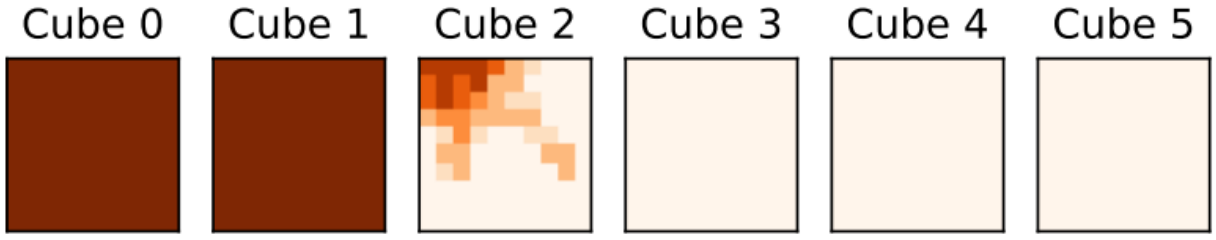
WayPointRL VI

- Hyperparameters include α (learning rate), γ (discount factor), and ϵ (for ϵ-greedy action selection). A boolean flag oneTime indicates if once pred_n is reached, all further states are considered in the target space.
- If oneTime $= \top$, reaching pred_n sets reached $\leftarrow \top$, and all subsequent transitions use the Q-table of pred_n.
- The *init* method initializes one Q-table per predicate. The *newEpisode* method resets the trace and reached flag, and determines the initial active predicate.
- The *pick* method performs ϵ-greedy action selection using the Q-table of the current active predicate.
- The *recordStep* method updates the active predicate for the next state and records the transition, setting the reached flag if the target predicate is encountered.

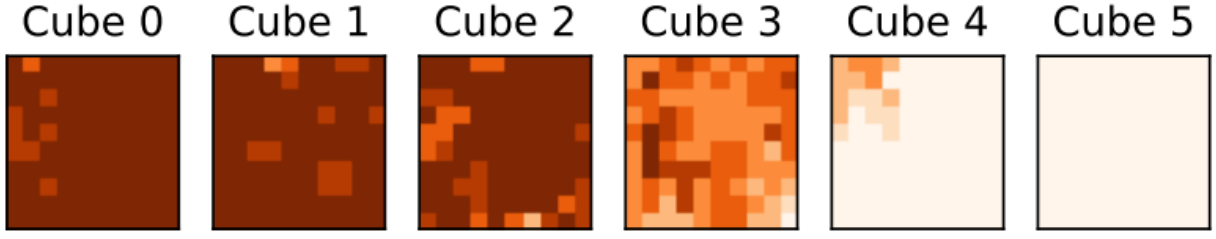
Experiments: Intuition with exploring cube world example I



(a) Random exploration

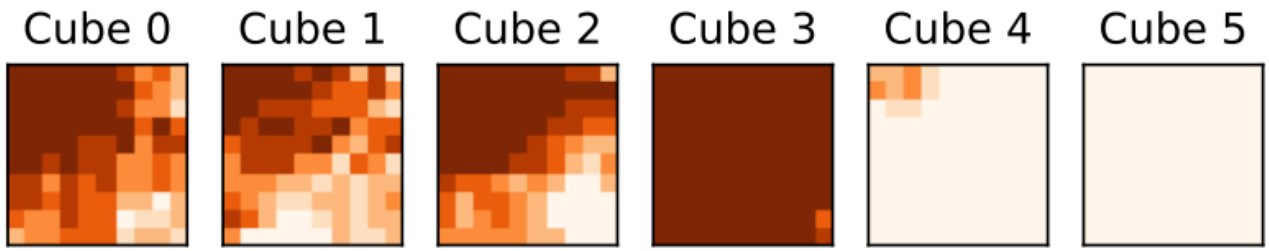


(b) BonusMaxRL exploration



(c) BonusMaxRL exploration with depth abstraction

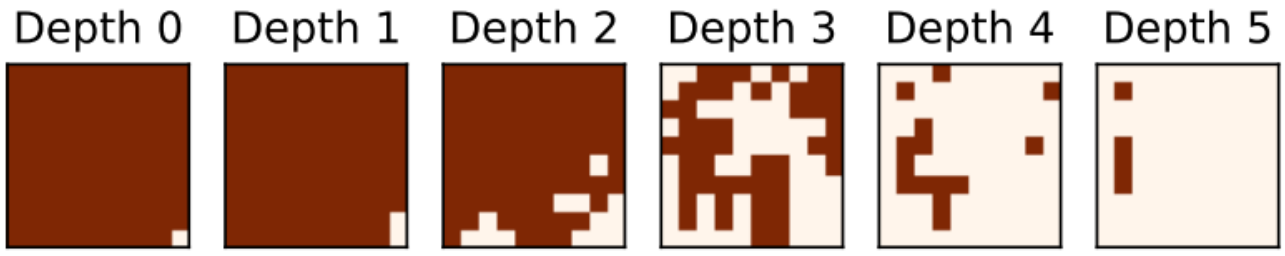
Experiments: Intuition with exploring cube world example II



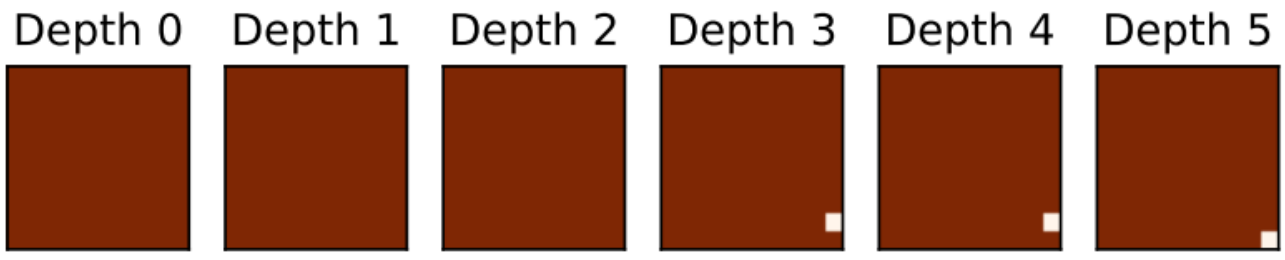
(d) WaypointRL exploration (target cube 3)

Fig. 1. Exploration of a $6 \times 10 \times 10 \times 6$ cube world, with a given episode budget, using different agents. We plot the heatmap of the top of each cube. The intensity is the sum of the visited cells along the depth of the cube, with the darkest color meaning all the cells have been visited. Here we showcase several points. First, BonusMaxRL (b) achieves better exploration than Random (a), covering more cells. Second, unbiased exploration struggles to reach cubes away from the starting point (b). Third, chosing an appropriate state space abstraction can lead to better coverage, but it can result in reduced capabilities of systematically exploring a target subspace (c), while WaypointRL is able to effectively bias the exploration towards the target cube and almost fully cover it (d).

Experiments: Intuition with exploring cube world example III



(a) BonusMaxRL exploration with depth abstraction



(b) WaypointRL cube 3 exploration

Fig. 2. Detailed Exploration of cube 3 in the $6 \times 10 \times 10 \times 6$ cube world. Each grid represents a depth level of the cube. The colored cells have been explored by the agent. (a) BonusMaxRL using the depth abstraction (b) WaypointRL with reaching cube 1, 2, and 3 as waypoints. WaypointRL is able to explore almost all the cells of the cube.

Experiments: Intuition with exploring cube world example IV

- The environment is a 3D cube world with a state space defined as $S = (g, w, b, d)$, where g is the cube index, and w, b, d are spatial coordinates within each cube.
- Agents start at $(0, 0, 0, 0)$ and can take actions: up, down, left, right, above, below, into, and reset_depth.
- *into* action moves the agent to the next cube via doors at specific positions (e.g., $(0, 5, 5, d) \xrightarrow{\text{into}} (1, 0, 0, 0)$ for any d), modeling irreversible transitions.
- *reset_depth* returns the agent to depth zero, modeling state resets.
- With 6 cubes of size $10 \times 10 \times 6$, agents run for a fixed number of steps per episode across multiple episodes.

Experiments: Intuition with exploring cube world example V

- BonusMaxRL achieves greater coverage than random exploration but struggles to reach distant cubes within the episode budget.
- Adding a bonus reward for reaching cube 3 is ineffective if the agent never gets there, illustrating the limits of naive reward strategies.
- Introducing an abstraction by removing depth reduces state space and improves cube coverage but sacrifices depth granularity.
- WaypointRL splits the task into: reaching the target cube and exploring it thoroughly.
- WaypointRL uses separate Q-tables and reward functions for each task, allowing targeted and fine-grained exploration.

Experiments: Intuition with exploring cube world example VI

- Intermediate waypoints (e.g., reaching cubes 1 and 2) help guide the agent to cube 3 efficiently.
- WaypointRL generalizes the reward approach via target predicates associated with Q-tables, achieving both efficient navigation and full-depth exploration.
- This setup is like distributed systems where the state transition depend on message interleavings and events like group decisions or timeouts.

Benchmark	Random	BonusMaxRL	NegRL
RedisRaft	27081.9 ± 2627.31	32818.4 ± 3017.32	33433.2 ± 2972.34
Etcd	19179.3 ± 106.26	22202.7 ± 137.75	**24898.6 ± 97.40**
RSL	678.9 ± 36.43	**2020.9 ± 190.67**	751.2 ± 37.24

Table 2. Final average coverage values for the different benchmarks.

Experiments: Intuition with exploring cube world example VII

Benchmarks	No.Pred	WaypointRL	BonusMaxRL	NegRLVisits	Random
RedisRaft					
OneInTerm(3)	1	15934 ± 3040	16640 ± 2033	17214 ± 2103	14370 ± 1810
AllInTerm(2)	1	**23695 ± 3810**	7660 ± 945	8437 ± 876	6401 ± 794
TermDiff(2)	1	23478 ± 2917	22214 ± 1973	23323 ± 2149	20112 ± 2007
CommitEntries(2)	1	**24656 ± 3995**	4894 ± 552	5325 ± 741	2835 ± 530
EntryInTerm(2)	3	**22758 ± 5457**	8267 ± 272	10114 ± 418	9812 ± 662
LeaderInTerm(2)	1	**22533 ± 5418**	8971 ± 299	10834 ± 410	10361 ± 684
LogDiff(1)	1	**30779 ± 3365**	5755 ± 713	5958 ± 825	3332 ± 606
LogCommitDiff(3)	2	**14960 ± 4100**	154 ± 63	158 ± 52	102 ± 41
OneLeaderOneCandidate	3	1301 ± 1360	482 ± 120	471 ± 158	356 ± 93
Etcd					
LogCommitGap(3)	4	**13336 ± 664**	5692 ± 132	6411 ± 112	4717 ± 92
OneInTerm(4)	3	**33011 ± 826**	29545 ± 258	29739 ± 167	25309 ± 200
MinCommit(2)	3	**29862 ± 885**	25015 ± 193	24031 ± 110	21765 ± 195
TermDiff(2)	1	**15289 ± 885**	4673 ± 162	7792 ± 66	4879 ± 142
LeaderInTerm(4)	3	**15727 ± 1157**	10684 ± 142	11171 ± 215	9571 ± 101
AtLeastOneCommitInTerm(2)	3	**32709 ± 1025**	28169 ± 262	25863 ± 176	23903 ± 314
OneLeaderOneCandidate	3	35403 ± 958	36178 ± 142	32891 ± 208	29021 ± 307
LogGap(2)	2	**37832 ± 3485**	31372 ± 314	33040 ± 115	27445 ± 238
AllInTerm(5)	3	**10346 ± 1121**	8202 ± 65	7888 ± 122	6682 ± 94
RSL					
AnyBallot(3)	1	**1573 ± 174**	837 ± 75	301 ± 34	264 ± 31
AllBallot(3)	1	**1021 ± 57**	493 ± 92	102 ± 14	101 ± 16
EntryBallot(2)	1	1016 ± 460	**1954 ± 131**	698 ± 27	658 ± 48
AnyDecree(2)	1	**1068 ± 119**	663 ± 52	188 ± 19	155 ± 23
BallotDiff(2)	1	19 ± 5	12 ± 6	2 ± 2	3 ± 3
AnyDecided(3)	1	**856 ± 93**	492 ± 46	134 ± 18	110 ± 16
PrimaryInBallot(2)	2	**607 ± 66**	467 ± 54	232 ± 22	196 ± 30
DecidedDiff(3)	3	**113.7 ± 46.8**	22.7 ± 10.1	5.7 ± 3.1	2.9 ± 1.5

Table 4. Coverage results - the table shows the target coverage results in our benchmarks. Each row contains the target predicate, the number of predicates in the sequence used by WaypointRL (excluding the first one), and, for each algorithm, the average number of unique explored states (± Standard Deviation).

Remarks and similar works I

- *Systematic exploration techniques* have scalability issues due to large state spaces in distributed systems.
- *Model-based testing* also struggles to enumerate all possible failure scenarios.
- *Randomized techniques* are effective by inducing arbitrary failures like node crashes and network partitions.
- *Other random methods* include:
  - Probabilistic guarantees for bug detection.
  - Partial order reduction to reduce state space.
  - Standard fuzzing via input mutation.
- *Verification approaches* aim to eliminate testing by proving correctness or generating code from verified specs, but:
  - Rarely used in production.
  - Bugs can still occur at the interfaces between verified and unverified components.
- *Reinforcement learning for testing*:

Remarks and similar works II

- Q-learning has been applied to concurrent and distributed systems.
- Custom reward mechanisms encourage exploration but may not bias exploration effectively.
- RL has also been used in input generation, parameter tuning, and control system failure induction.

- *Reward-free exploration*:
  - Theoretical guarantees over state coverage using decaying reward augmentation.
  - Requires polynomial (5th degree) number of episodes in state/action space size.
  - Ineffective in practice for large-scale systems and hard to bias exploration.
- *The paper's approach*: Extends pure exploration with semantic reward augmentation using programmer knowledge, enabling effective biasing.
- *Hierarchical RL*:

Remarks and similar works III

- Splits tasks into subtasks with learned policies and global coordination.
- The paper's method instead uses fixed-priority waypoints without policy reuse.

- *Reward machines* offer structured reward specification and decomposition:
  - Enable hierarchical learning and efficient policy optimization.
  - Not applicable since the rewards depend on actual system transitions, which are not decoupled from environment.
- *Temporal logic goals in RL*:
  - Use automata to define goals and subgoals as intermediate automaton states.
  - Simulation of transitions based on temporal subgoals is infeasible with real-world systems due to limited instrumentation.

References

- Andrea Borgarelli, Constantin Enea, Rupak Majumdar, and Srinidhi Nagendra. *Reward Augmentation in Reinforcement Learning for Testing Distributed Systems*. Proceedings of the ACM on Programming Languages, 8(OOPSLA2):1928–1954, 2024.
- Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. *Learning-based controlled concurrency testing*. Proceedings of the ACM on Programming Languages, 4(OOPSLA):1–31, 2020.

Thank You!