LLM Hypothesis Evolution vs RL for Underspecified Tasks

Presenter: Allen Jue

Motivation

- Lots of tasks are underspecified
- Maybe lots of personal preference
- How can robots learn these kinds of manipulation tasks?

https://th-thumbnailer.cdn-si-edu.com/MpWUVG_c49JOLWQzJ3tHAeRj5Tw=/fit-in/1600x0/https://tf-cmsv2-smithsonianmag-media.s3.amazonaws.com/filer/24/a1/24a169e2-a5e4-4bd7-9c4a-3d9dd100450b/0messy-desk.jpg

Related Work

Reinforcement Learning for Robotics

- PPO - for manipulation and navigation
- Survey on RL in robotics (Tang et al., 2024) - sample efficiency is still an issue

LLMs for Robot Planning

- SayCan (Ahn et al., 2022) - LLMs provide task knowledge
- Inner Monologue (Huang et al., 2022) - Closed-loop feedback with LLM to adjust to new task

Core Comparison

LLMs are good with robots, let's measure *how* good for underspecified tasks.

- How does PPO compare to LLM-guided robot manipulation?
  - Data Efficiency
  - Generalization
  - Interpretability

Experiment

- Use robosuite Panda arm for experiments
- Tabletop scene with assorted colored blocks
- For simplicity, movement primitives to pickup and swap blocks provided
- Compare PPO usage of primitives with an evolutionary policy generation with LLMs (ShinkaEvolve)

LLM-guided policy generation

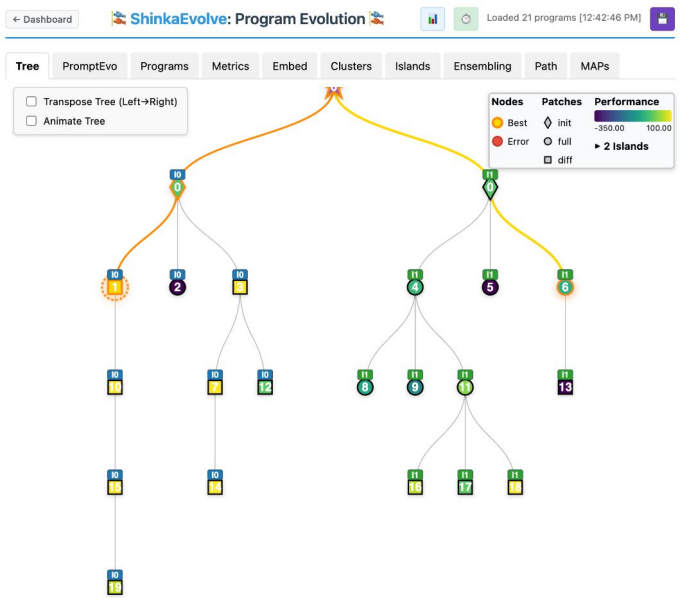
- I tried multiple prompting strategies CoT, ReAct, Self-Consistency.
- For hypothesis generation, I asked it to:
  - Generate plausible hypotheses in multiple choice fashion.
  - Select and generate the one it is most confident in.
- Ran into problems:
  - Hallucinationated policies are error-prone
  - Brittle policies break in iterations to improve
  - Prompting headaches, what to include?
  - Calibration is an issue for LLMs.

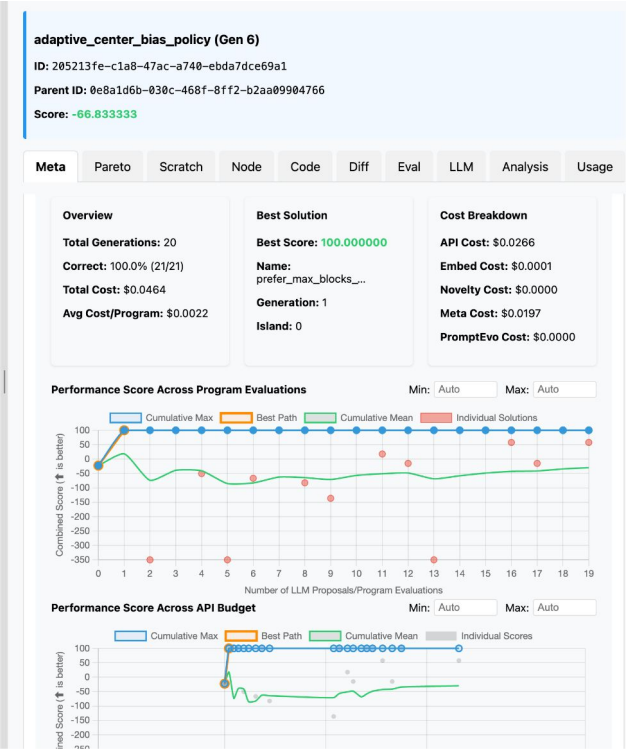
LLM first Attempt

- Learns policy for pickup certain blocks.
- Can see reasoning in policy.
- But this broke for more complex tasks like sorting and picking up the second block from the right.

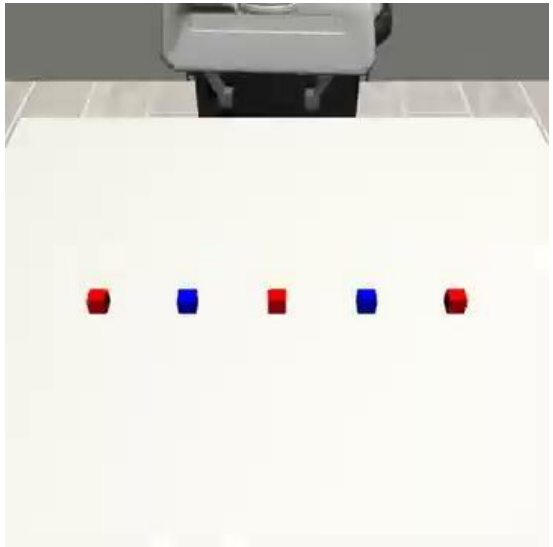
```
# Evolved policy (iteration 3)
# Accepted: True
# Reason: I'm adding a bias toward blocks that
    appear later in the sequence (higher indices)
    since we've seen successful episodes where block
     4 was picked. This might help the policy focus
    on higher-numbered blocks when making random
    choices.
def policy(state, memory):
    n = state["num_blocks"]
    picked = state.get("picked", [])
    available = [i for i in range(n)
                 if not picked[i]]
    if available:
        return max(available)
    return 0
```

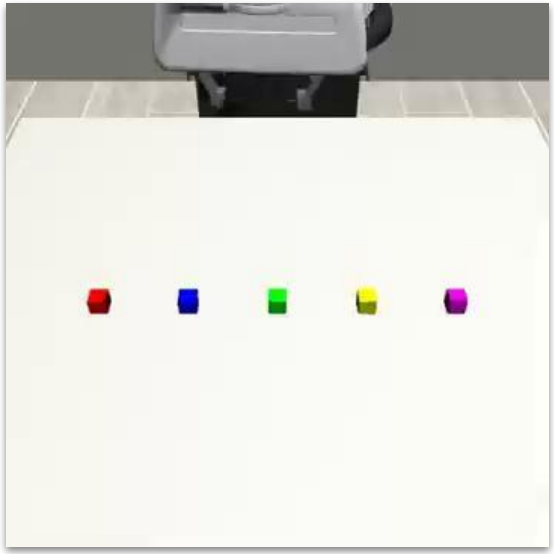
LLM with ShinkaEvolve





Example





Results: PPO

- PPO can learn to sort blocks in certain color orderings and pick up certain blocks.
- Requires thousands of episodes.
- Results are achieved with perfect success.

TABLE I
EPISODES FOR PPO TO REACH PERFECT ROLLING SR ($SR_{LAST50} = 1.0$).
ABSTRACT ENV, $N \in [3, 6]$, SEED 42.

Rule	Episodes
Sort — reds left	1,304
Sort — blues left	1,186
Pickup — leftmost	2,423
Pickup — 2nd left	2,580
Pickup — rightmost	4,917
Pickup — 2nd right	7,801

Sample Complexity Comparison

Task	Variant	PPO SR	Shinka SR	N_{PPO}	N_{Sh}	$N_{\text{PPO}}/N_{\text{Sh}}$	$\text{SR}_{\text{Sh}}/\text{SR}_{\text{PPO}}$
Pickup	rightmost	1.0000	1.0000	700	180	**3.89**	1.0000
Pickup	2nd_right	1.0000	1.0000	5800	1530	**3.79**	1.0000
Pickup	leftmost	1.0000	1.0000	700	180	**3.89**	1.0000
Pickup	2nd_left	1.0000	0.0000	1800	3030	0.59	0.0000
Sort	reds_left_dense	0.9860	1.0000	30000	630	**47.62**	**1.0142**
Sort	reds_left_sparse	0.9970	1.0000	1800	180	**10.00**	**1.0030**
Sort	blues_left_dense	0.9560	0.6267	15500	3030	**5.12**	0.6555
Sort	blues_left_sparse	0.9320	1.0000	20800	1380	**15.07**	**1.0730**
Table	dense_unlocked	1.0000	0.4200	700	3030	0.23	0.4200
Table	dense_locked	1.0000	0.0400	200	3030	0.07	0.0400
Table	sparse_unlocked	1.0000	0.0000	8400	3030	2.77	0.0000
Table	sparse_locked	1.0000	0.0000	2300	3030	0.76	0.0000

Interpretability

```python
# EVOLVE-BLOCK-START
def policy(state, memory):
    """Improved heuristic policy to cluster red blocks to the left."""
    n = int(state.get("num_blocks", 0))
    max_blocks = int(state.get("max_blocks", 10))
    noop_action = int(state.get("noop_action", max_blocks * max_blocks))
    colors = state.get("colors", [])
    if n < 2 or len(colors) < n:
        return noop_action

    # Try to find a red block on the right side and a non-red block on the left side to swap
    # Goal: cluster reds to the left
    red = "red"
    left_idx = None
    right_idx = None

    # Find leftmost non-red block that should be swapped
    for i in range(n):
        if colors[i] != red:
            left_idx = i
            break

    # Find rightmost red block that should be swapped
    for j in range(n - 1, -1, -1):
        if colors[j] == red:
            right_idx = j
            break
```

- Lots of interpretability.
- Option for feedback after running generations.

Generalization

Key Takeaways

- Generalizability with writing code policies can be limited (table scene)
- Worth just trying evolution because in some instances, it can generate excellent code policies with much fewer samples.
- LLM policies are readable, and you can often fix the policy with small changes.
- Tradeoff between just let PPO reward function vs. crafting a good initial prompt and fitness function.

Next Steps

- Try with multimodal LLMs. Visual affordances can be very helpful.
- Run experiments on local and open source LLMs
  - See if size is an indicator of performance
  - Try different parameters for Shinka evolve
  - See variability in evolution – analyze which programs are successful, evolution success over time, and variability amongst evolution branches.

Blue Left Sparse

```python
# EVOLVE-BLOCK-START
def policy(state, memory):
    """Policy that sorts blocks by ranked_colors using largest beneficial swap per step."""
    n = int(state.get("num_blocks", 0))
    max_blocks = int(state.get("max_blocks", 10))
    noop_action = int(state.get("noop_action", max_blocks * max_blocks))
    if n < 2:
        return noop_action

    colors = state.get("colors", [])
    ranked_colors = state.get("ranked_colors", [])
    if len(colors) < n or len(ranked_colors) < n:
        # Defensive fallback to random if colors info missing
        a = random.randint(0, n - 2)
        b = random.randint(a + 1, n - 1)
        return a * max_blocks + b

    # Find the swap (i,j) with i<j that most reduces disorder (rank_i > rank_j)
    # Prefer largest gap j - i to accelerate sorting
    best_i = None
    best_j = None
    max_gap = 0
    for i in range(n - 1):
        rank_i = ranked_colors[i]
        for j in range(i + 1, n):
            rank_j = ranked_colors[j]
            if rank_i > rank_j:
                gap = j - i
                if gap > max_gap:
                    max_gap = gap
                    best_i = i
                    best_j = j

    if best_i is not None and best_j is not None:
        return best_i * max_blocks + best_j

    # No beneficial swap found, noop
    return noop_action
```

Fragile Primitives…

Dashboard

http://localhost:8888/viz_tree.html?db_path=programs.sqlite&left_tab=tree-view&right_tab=agent-info&selected_node=205213fe-c1a8-47ac-a740-ebda7dce69a1

Discussion

Key Takeaways