```python

# %%%%%%%%%% A. NOTEBOOK SETUP %%%%%%%%%%
# This cell sets up the environment by importing the necessary libraries.

import torch
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from IPython.display import display, Markdown

# Set a consistent style for plots
plt.style.use('seaborn-v0_8-whitegrid')

print("Libraries imported successfully.")


# %%%%%%%%%% B. SIMULATION CORE: GRAPH GENERATION %%%%%%%%%%
# This cell contains functions to generate the different task dependency graphs
# that will be used in our assessment scenarios.

def generate_chain_graph(num_nodes):
    """Generates a simple chain graph (0 -> 1 -> 2 -> ...)."""
    G = nx.DiGraph()
    G.add_nodes_from(range(num_nodes))
    for i in range(num_nodes - 1):
        G.add_edge(i, i + 1)
    return G

def generate_tree_graph(num_nodes, fan_out=3):
    """Generates a tree graph with one root branching out."""
    if num_nodes == 0:
        return nx.DiGraph()
    G = nx.DiGraph()
    G.add_node(0) # Root node
    nodes_to_process = [0]
    next_node_id = 1
    while nodes_to_process and next_node_id < num_nodes:
        current_node = nodes_to_process.pop(0)
        for i in range(fan_out):
            if next_node_id < num_nodes:
                G.add_node(next_node_id)
                G.add_edge(current_node, next_node_id)
                nodes_to_process.append(next_node_id)
                next_node_id += 1
    return G

def generate_inverted_tree_graph(num_nodes, fan_in=3):
    """Generates an inverted tree with many roots converging to one final task."""
    if num_nodes == 0:
        return nx.DiGraph()
    G = nx.DiGraph()
    final_task = num_nodes - 1
    G.add_node(final_task)
    # The last node is the sink. All other nodes are potential sources.
    # We will form layers that converge on the final task.

    # This is a simplified generation logic
    num_leaves = num_nodes - 1
    for i in range(num_leaves):
        G.add_node(i)
        G.add_edge(i, final_task)

    return G

def generate_dag(num_nodes, sparsity=0.2):
    """Generates a random Directed Acyclic Graph (DAG)."""
    G = nx.DiGraph()
    G.add_nodes_from(range(num_nodes))
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            if np.random.rand() < sparsity:
```

```
                G.add_edge(i, j)
    return G

print("Graph generation functions are defined.")


# %%%%%%%%%% C. SIMULATION CORE: THE SIMULATOR CLASS %%%%%%%%%%
# This is the heart of the benchmark. The `CurriculumSimulator` class runs the
# simulation for a given curriculum, task graph, and set of dynamic parameters.
# It now includes logic for all the new curricula we designed.

class CurriculumSimulator:
    """Runs a curriculum learning simulation for a given configuration."""

    def __init__(self, config, graph, curriculum_type):
        self.config = config
        self.G = graph
        self.curriculum = curriculum_type

        # Pre-compute graph properties
        self.adj = [list(self.G.successors(i)) for i in range(self.config.NUM_TASKS)]
        self.parents = [list(self.G.predecessors(i)) for i in range(self.config.NUM_TASKS)]
        try:
            self.topological_order = list(nx.topological_sort(self.G))
            self.reverse_topological_order = self.topological_order[::-1]
        except nx.NetworkXUnfeasible: # For graphs with no clear topological sort (e.g., disconnected)
            self.topological_order = list(range(self.config.NUM_TASKS))
            self.reverse_topological_order = list(range(self.config.NUM_TASKS))[::-1]


        # Initialize performance and history trackers
        self.P = torch.full((self.config.NUM_TASKS,), 0.01)
        self.P_history = [self.P.clone()]
        self.sampling_probs_history = [torch.full((self.config.NUM_TASKS,), 1.0 / self.config.NUM_TASKS)]

        # State for specific curricula
        self.fast_ema = torch.zeros(self.config.NUM_TASKS)
        self.slow_ema = torch.zeros(self.config.NUM_TASKS)
        self.perf_window = torch.zeros((self.config.VARIANCE_WINDOW, self.config.NUM_TASKS))
        self.topological_idx = 0


    def _get_sampling_probs(self, epoch):
        """Calculates sampling probabilities based on the chosen curriculum."""
        # --- BASELINES AND HEURISTICS ---
        if self.curriculum == 'random' or epoch == 0:
            return torch.full((self.config.NUM_TASKS,), 1.0 / self.config.NUM_TASKS)

        if self.curriculum == 'topological':
            probs = torch.zeros(self.config.NUM_TASKS)
            current_task = self.topological_order[self.topological_idx]
            probs[current_task] = 1.0
            # Move to the next task if current one is mastered
            if self.P[current_task] >= self.config.PERFORMANCE_THRESHOLD:
                self.topological_idx = min(self.topological_idx + 1, self.config.NUM_TASKS - 1)
            return probs

        if self.curriculum == 'reverse_topological':
            probs = torch.zeros(self.config.NUM_TASKS)
            current_task = self.reverse_topological_order[self.topological_idx]
            probs[current_task] = 1.0
            if self.P[current_task] >= self.config.PERFORMANCE_THRESHOLD:
                self.topological_idx = min(self.topological_idx + 1, self.config.NUM_TASKS - 1)
            return probs

        if self.curriculum == 'oracle':
            # This heuristic Oracle identifies tasks whose parents are all mastered
            # and samples uniformly from that set of "unlocked" tasks.
            unlocked_tasks = []
            for i in range(self.config.NUM_TASKS):
                if self.P[i] < self.config.PERFORMANCE_THRESHOLD: # Task is not yet mastered
                    parents_mastered = all(self.P[p] >= self.config.PERFORMANCE_THRESHOLD for p in self.parents[i])
                    if parents_mastered:
                        unlocked_tasks.append(i)

```

```python
            if not unlocked_tasks: # If nothing is unlocked, fall back to random
                return torch.full((self.config.NUM_TASKS,), 1.0 / self.config.NUM_TASKS)

            probs = torch.zeros(self.config.NUM_TASKS)
            for task_idx in unlocked_tasks:
                probs[task_idx] = 1.0
            return probs / probs.sum()

        # --- ADAPTIVE CURRICULA ---
        last_P = self.P_history[-1]

        if self.curriculum == 'lp_finite_diff':
            second_last_P = self.P_history[-2] if len(self.P_history) > 1 else last_P
            progress = torch.abs(last_P - second_last_P)

        elif self.curriculum == 'lp_ema_diff':
            self.fast_ema = self.config.FAST_EMA_ALPHA * last_P + (1 - self.config.FAST_EMA_ALPHA) * self.fast_ema
            self.slow_ema = self.config.SLOW_EMA_ALPHA * last_P + (1 - self.config.SLOW_EMA_ALPHA) * self.slow_ema
            progress = torch.abs(self.fast_ema - self.slow_ema)

        elif self.curriculum == 'variance':
            # Update performance window
            self.perf_window = torch.roll(self.perf_window, shifts=-1, dims=0)
            self.perf_window[-1, :] = last_P
            # Progress is the variance over the last K epochs
            progress = torch.var(self.perf_window, dim=0)

        else:
            raise ValueError(f"Unknown curriculum: {self.curriculum}")

        # Convert progress scores to probabilities using softmax
        if torch.all(progress == 0): # Avoid NaN if progress is zero everywhere
            return torch.full((self.config.NUM_TASKS,), 1.0 / self.config.NUM_TASKS)

        return torch.nn.functional.softmax(progress / self.config.SOFTMAX_TEMP, dim=0)


    def run(self):
        """Executes the full simulation loop."""
        for epoch in range(self.config.NUM_EPOCHS):
            sampling_probs = self._get_sampling_probs(epoch)
            self.sampling_probs_history.append(sampling_probs)

            # Get sample counts for each task
            S = torch.multinomial(sampling_probs, self.config.TOTAL_SAMPLES_PER_EPOCH, replacement=True)
            S_counts = torch.bincount(S, minlength=self.config.NUM_TASKS).float()

            # Calculate performance change (P_dot)
            current_P = self.P
            P_dot = torch.zeros(self.config.NUM_TASKS)

            for i in range(self.config.NUM_TASKS):
                parent_contribution = sum(S_counts[p_idx] for p_idx in self.parents[i])
                total_stimulus = S_counts[i] + self.config.GAMMA * parent_contribution

                children_gate = 1.0
                if self.adj[i]: # If task i has children
                    children_gate = torch.prod(torch.tensor([current_P[c_idx] for c_idx in self.adj[i]]))

                growth = total_stimulus * children_gate * (1 - current_P[i])
                forgetting = self.config.LAMBDA * current_P[i]

                # Normalize by total samples to make update step size independent of sample count
                P_dot[i] = (growth - forgetting) / self.config.TOTAL_SAMPLES_PER_EPOCH

            # Update performance (Euler integration with dt=1)
            new_P = current_P + P_dot
            self.P = torch.clamp(new_P, 0, 1)
            self.P_history.append(self.P.clone())

            # Early stopping if threshold is met
            if torch.all(self.P >= self.config.PERFORMANCE_THRESHOLD):
                # Pad history to full length for consistent analysis
                pad_len = self.config.NUM_EPOCHS - epoch -1
                if pad_len > 0:
```

```
                    self.P_history.extend([self.P.clone()] * pad_len)
                break

    def calculate_metrics(self):
        """Calculates summary metrics after the simulation."""
        P_history_tensor = torch.stack(self.P_history)
        mean_perf_per_epoch = [p.mean().item() for p in self.P_history]

        # Learning Efficiency
        efficiency = sum(mean_perf_per_epoch)

        # Time to Thresholds
        time_to_threshold = -1
        time_to_first_mastery = -1

        for i, p_epoch in enumerate(self.P_history):
            if time_to_first_mastery == -1 and torch.any(p_epoch >= self.config.PERFORMANCE_THRESHOLD):
                 time_to_first_mastery = i
            if time_to_threshold == -1 and torch.all(p_epoch >= self.config.PERFORMANCE_THRESHOLD):
                time_to_threshold = i

        # Final Performance Variance
        final_perf_variance = torch.var(self.P_history[-1]).item()

        return {
            'efficiency': efficiency,
            'time_to_threshold': time_to_threshold,
            'time_to_first_mastery': time_to_first_mastery,
            'final_perf_variance': final_perf_variance,
        }

print("CurriculumSimulator class defined.")


# %%%%%%%%%% D. BENCHMARKING FRAMEWORK %%%%%%%%%%
# This cell defines the full benchmark. It sets up the scenarios and curricula
# to test, then runs the sweep, storing results in a pandas DataFrame.

class Config:
    # Graph params
    NUM_TASKS = 12
    SPARSITY = 0.3
    # Dynamics params
    GAMMA = 0.5
    LAMBDA = 0.01
    # Simulation params
    NUM_EPOCHS = 200
    TOTAL_SAMPLES_PER_EPOCH = 100
    PERFORMANCE_THRESHOLD = 0.9
    # LP/Variance Curriculum params
    SOFTMAX_TEMP = 0.1
    FAST_EMA_ALPHA = 0.3
    SLOW_EMA_ALPHA = 0.05
    VARIANCE_WINDOW = 5 # For variance curriculum

# --- Define Scenarios ---
scenarios = {
    "1_Chain_Simple": {
        "graph_type": "chain",
        "LAMBDA": 0.0,
        "GAMMA": 0.5
    },
    "2_Chain_HighForget": {
        "graph_type": "chain",
        "LAMBDA": 0.05, # High forgetting
        "GAMMA": 0.5
    },
    "3_Tree_Divergent": {
        "graph_type": "tree",
        "LAMBDA": 0.01,
        "GAMMA": 0.5
    },
    "4_InvertedTree_Convergent": {
        "graph_type": "inverted_tree",
        "LAMBDA": 0.01,
```

```python
        "GAMMA": 0.5
    },
    "5_ComplexDAG_LowTransfer": {
        "graph_type": "dag",
        "LAMBDA": 0.01,
        "GAMMA": 0.1 # Low knowledge transfer
    },
}

# --- Define Curricula ---
curricula_to_test = [
    "random",
    "topological",
    "reverse_topological",
    "lp_finite_diff",
    "lp_ema_diff",
    "variance",
    "oracle"
]

def run_benchmark_sweep():
    """Runs the full experimental sweep over all scenarios and curricula."""
    results = []

    # Generate a fixed set of graphs for consistency across scenarios of the same type
    base_graphs = {
        "chain": generate_chain_graph(Config.NUM_TASKS),
        "tree": generate_tree_graph(Config.NUM_TASKS),
        "inverted_tree": generate_inverted_tree_graph(Config.NUM_TASKS),
        "dag": generate_dag(Config.NUM_TASKS, Config.SPARSITY)
    }

    total_runs = len(scenarios) * len(curricula_to_test)
    run_count = 0

    for s_name, s_params in scenarios.items():
        for curriculum in curricula_to_test:
            run_count += 1
            print(f"Running ({run_count}/{total_runs}): Scenario='{s_name}', Curriculum='{curriculum}'...")

            # Setup config for this run
            config = Config()
            config.LAMBDA = s_params["LAMBDA"]
            config.GAMMA = s_params["GAMMA"]
            graph = base_graphs[s_params["graph_type"]]

            # Run simulation
            simulator = CurriculumSimulator(config, graph, curriculum)
            simulator.run()
            metrics = simulator.calculate_metrics()

            # Store results
            result_row = {
                "scenario": s_name,
                "curriculum": curriculum,
                "graph_type": s_params["graph_type"],
                **metrics
            }
            results.append(result_row)

    print("\nBenchmark sweep complete.")
    return pd.DataFrame(results)

# --- Execute the sweep ---
results_df = run_benchmark_sweep()
display(results_df)


# %%%%%%%%%% E. REGRET CALCULATION %%%%%%%%%%
# This cell processes the raw results DataFrame to calculate regret metrics.
# Regret is calculated by comparing each curriculum's performance to the
# 'oracle' in the same scenario.

def calculate_regret(df):
    """Calculates regret metrics based on the oracle's performance."""
```

```
    oracle_metrics = df[df['curriculum'] == 'oracle'].set_index('scenario')

    regret_data = []

    for index, row in df.iterrows():
        if row['curriculum'] == 'oracle':
            # Oracle has zero regret by definition
            efficiency_regret = 0
            time_regret = 0
        else:
            scenario = row['scenario']
            oracle_row = oracle_metrics.loc[scenario]

            # Efficiency Regret
            efficiency_regret = oracle_row['efficiency'] - row['efficiency']

            # Time Regret
            # If oracle or curriculum failed (-1), regret is complex.
            # We'll define it as a large number if the curriculum fails but oracle succeeds.
            if row['time_to_threshold'] == -1 and oracle_row['time_to_threshold'] != -1:
                time_regret = Config.NUM_EPOCHS # Max penalty
            elif row['time_to_threshold'] == -1 and oracle_row['time_to_threshold'] == -1:
                time_regret = 0 # Both failed
            elif row['time_to_threshold'] != -1 and oracle_row['time_to_threshold'] == -1:
                 time_regret = -Config.NUM_EPOCHS # Actually did better than failing oracle!
            else:
                time_regret = row['time_to_threshold'] - oracle_row['time_to_threshold']

        regret_data.append({
            'efficiency_regret': efficiency_regret,
            'time_regret': time_regret
        })

    regret_df = pd.DataFrame(regret_data, index=df.index)
    return df.join(regret_df)

# --- Execute regret calculation ---
results_with_regret_df = calculate_regret(results_df)
display(Markdown("### Results DataFrame with Regret Metrics"))
display(results_with_regret_df)


# %%%%%%%%%% F. VISUALIZATION SUITE %%%%%%%%%%
# This final part generates the visualizations. It includes functions for both
# the detailed bar charts and the high-level summary star plots, which provide
# a compelling final comparison.

def plot_bar_charts(df, scenario_name):
    """Generates a bar chart comparing all curricula for a given scenario."""

    scenario_df = df[df['scenario'] == scenario_name].set_index('curriculum')

    metrics_to_plot = [
        'efficiency', 'time_to_threshold',
        'efficiency_regret', 'time_regret',
        'time_to_first_mastery', 'final_perf_variance'
    ]

    fig, axes = plt.subplots(3, 2, figsize=(18, 15))
    axes = axes.ravel()
    fig.suptitle(f"Metric Comparison for Scenario: {scenario_name}", fontsize=20)

    for i, metric in enumerate(metrics_to_plot):
        data = scenario_df[metric].sort_values(ascending=False)
        colors = plt.cm.viridis(np.linspace(0, 1, len(data)))
        bars = axes[i].bar(data.index, data.values, color=colors)
        axes[i].set_title(metric.replace('_', ' ').title(), fontsize=14)
        axes[i].tick_params(axis='x', rotation=45)
        axes[i].bar_label(bars, fmt='%.1f')

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()


def plot_star_plots(df):
    """
```

```
    Generates summary star plots (radar charts) comparing curricula across all scenarios.
    This is the final, compelling visualization of strengths and weaknesses.
    """

    # --- 1. Normalize Metrics ---
    # Star plots require metrics to be on a similar scale, where 'bigger is better'.
    df_norm = df.copy()

    # For 'lower is better' metrics, we invert them.
    for col in ['time_to_threshold', 'time_to_first_mastery', 'final_perf_variance', 'efficiency_regret', 'time_regret
        # Handle -1 (failure) case for time metrics
        if 'time' in col:
            max_val = Config.NUM_EPOCHS
            # Replace -1 with max penalty
            df_norm[col] = df_norm[col].replace(-1, max_val * 1.1)

        # Invert the metric so higher is better
        # Add small epsilon to avoid division by zero
        min_val = df_norm[col].min()
        max_val = df_norm[col].max()
        if max_val - min_val == 0:
             df_norm[f'norm_{col}'] = 0.5
        else:
             df_norm[f'norm_{col}'] = (max_val - df_norm[col]) / (max_val - min_val)

    # For 'higher is better' metrics, we just scale them from 0 to 1.
    for col in ['efficiency']:
        min_val = df_norm[col].min()
        max_val = df_norm[col].max()
        if max_val - min_val == 0:
             df_norm[f'norm_{col}'] = 0.5
        else:
            df_norm[f'norm_{col}'] = (df_norm[col] - min_val) / (max_val - min_val)

    # --- 2. Create Plots ---
    metrics_for_star = [
        'norm_efficiency', 'norm_time_to_threshold',
        'norm_time_to_first_mastery', 'norm_final_perf_variance',
        'norm_efficiency_regret', 'norm_time_regret'
    ]

    # Clean labels for the plot
    theta_labels = [m.replace('norm_', '').replace('_', ' ').replace('Regret', ' (Low Regret)').title() for m in metri

    # Get unique scenarios and curricula
    scenarios = df_norm['scenario'].unique()
    curricula = df_norm[df_norm['curriculum'] != 'oracle']['curriculum'].unique() # Exclude oracle from comparison aga

    fig = make_subplots(
        rows=1, cols=len(scenarios),
        specs=[[{'type': 'polar'}] * len(scenarios)],
        subplot_titles=scenarios
    )

    for i, scenario in enumerate(scenarios):
        scenario_df = df_norm[df_norm['scenario'] == scenario]

        for curriculum in curricula:
            row = scenario_df[scenario_df['curriculum'] == curriculum]
            if not row.empty:
                r_values = row[metrics_for_star].values.flatten().tolist()
                fig.add_trace(
                    go.Scatterpolar(
                        r=r_values,
                        theta=theta_labels,
                        fill='toself',
                        name=curriculum,
                        legendgroup=curriculum,
                        showlegend=(i==0) # Show legend only for the first subplot
                    ),
                    row=1, col=i+1
                )

    fig.update_layout(
        height=600,
```

```
        width=400*len(scenarios),
        title_text="Curriculum Performance Profiles (Normalized)",
        legend_title_text='Curricula',
        polar=dict(radialaxis=dict(visible=True, range=[0, 1]))
    )
    fig.show()


# --- Execute Visualizations ---
display(Markdown("## Detailed Bar Chart Comparisons"))
display(Markdown("These charts provide a detailed look at curriculum performance in each specific scenario."))
for s_name in results_with_regret_df['scenario'].unique():
    plot_bar_charts(results_with_regret_df, s_name)

display(Markdown("## Global Star Plot Comparison"))
display(Markdown("""
This star plot is the final, high-level summary. Each point on the star represents a normalized performance metric,
where **a larger area is universally better**. This visualization quickly reveals the unique strengths and weaknesses
of each curriculum across the different environmental challenges. For example, a curriculum that excels in the 'High F
scenario will have a large area in that respective plot.
"""))
plot_star_plots(results_with_regret_df)
```

```
Libraries imported successfully.
Graph generation functions are defined.
CurriculumSimulator class defined.
Running (1/35): Scenario='1_Chain_Simple', Curriculum='random'...
Running (2/35): Scenario='1_Chain_Simple', Curriculum='topological'...
Running (3/35): Scenario='1_Chain_Simple', Curriculum='reverse_topological'...
Running (4/35): Scenario='1_Chain_Simple', Curriculum='lp_finite_diff'...
Running (5/35): Scenario='1_Chain_Simple', Curriculum='lp_ema_diff'...
Running (6/35): Scenario='1_Chain_Simple', Curriculum='variance'...
Running (7/35): Scenario='1_Chain_Simple', Curriculum='oracle'...
Running (8/35): Scenario='2_Chain_HighForget', Curriculum='random'...
Running (9/35): Scenario='2_Chain_HighForget', Curriculum='topological'...
Running (10/35): Scenario='2_Chain_HighForget', Curriculum='reverse_topological'...
Running (11/35): Scenario='2_Chain_HighForget', Curriculum='lp_finite_diff'...
Running (12/35): Scenario='2_Chain_HighForget', Curriculum='lp_ema_diff'...
Running (13/35): Scenario='2_Chain_HighForget', Curriculum='variance'...
Running (14/35): Scenario='2_Chain_HighForget', Curriculum='oracle'...
Running (15/35): Scenario='3_Tree_Divergent', Curriculum='random'...
Running (16/35): Scenario='3_Tree_Divergent', Curriculum='topological'...
Running (17/35): Scenario='3_Tree_Divergent', Curriculum='reverse_topological'...
Running (18/35): Scenario='3_Tree_Divergent', Curriculum='lp_finite_diff'...
Running (19/35): Scenario='3_Tree_Divergent', Curriculum='lp_ema_diff'...
Running (20/35): Scenario='3_Tree_Divergent', Curriculum='variance'...
Running (21/35): Scenario='3_Tree_Divergent', Curriculum='oracle'...
Running (22/35): Scenario='4_InvertedTree_Convergent', Curriculum='random'...
Running (23/35): Scenario='4_InvertedTree_Convergent', Curriculum='topological'...
Running (24/35): Scenario='4_InvertedTree_Convergent', Curriculum='reverse_topological'...
Running (25/35): Scenario='4_InvertedTree_Convergent', Curriculum='lp_finite_diff'...
Running (26/35): Scenario='4_InvertedTree_Convergent', Curriculum='lp_ema_diff'...
Running (27/35): Scenario='4_InvertedTree_Convergent', Curriculum='variance'...
Running (28/35): Scenario='4_InvertedTree_Convergent', Curriculum='oracle'...
Running (29/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='random'...
Running (30/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='topological'...
Running (31/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='reverse_topological'...
Running (32/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='lp_finite_diff'...
Running (33/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='lp_ema_diff'...
Running (34/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='variance'...
Running (35/35): Scenario='5_ComplexDAG_LowTransfer', Curriculum='oracle'...

Benchmark sweep complete.
```

	scenario	curriculum	graph_type	efficiency	time_to_threshold	time_to_first_mastery	final_perf_
0	1_Chain_Simple	random	chain	170.333115	66	17	6.8
1	1_Chain_Simple	topological	chain	59.916556	-1	29	1.9
2	1_Chain_Simple	reverse_topological	chain	188.156021	24	2	0.0
3	1_Chain_Simple	lp_finite_diff	chain	172.032845	64	5	6.9
4	1_Chain_Simple	lp_ema_diff	chain	111.270636	184	6	1.3
5	1_Chain_Simple	variance	chain	169.361163	67	16	7.4
6	1_Chain_Simple	oracle	chain	61.887785	-1	29	1.6
7	2_Chain_HighForget	random	chain	167.824185	63	20	6.6
8	2_Chain_HighForget	topological	chain	58.821264	-1	28	1.7
9	2_Chain_HighForget	reverse_topological	chain	187.235618	24	2	1.1
10	2_Chain_HighForget	lp_finite_diff	chain	172.371547	65	6	7.1
11	2_Chain_HighForget	lp_ema_diff	chain	109.621859	185	6	2.1
12	2_Chain_HighForget	variance	chain	169.088831	67	16	6.7
13	2_Chain_HighForget	oracle	chain	60.436184	-1	29	1.9
14	3_Tree_Divergent	random	tree	186.776147	55	17	7.6
15	3_Tree_Divergent	topological	tree	38.272161	-1	200	8.6
16	3_Tree_Divergent	reverse_topological	tree	188.741723	24	2	8.5
17	3_Tree_Divergent	lp_finite_diff	tree	186.622626	47	13	7.1
18	3_Tree_Divergent	lp_ema_diff	tree	177.935767	81	10	7.8
19	3_Tree_Divergent	variance	tree	186.499868	53	16	6.7
20	3_Tree_Divergent	oracle	tree	31.211558	-1	-1	5.5
21	4_InvertedTree_Convergent	random	inverted_tree	177.925998	32	3	5.4

22	4_InvertedTree_Convergent	topological	inverted_tree	188.712836	23	3	1.1
23	4_InvertedTree_Convergent	reverse_topological	inverted_tree	188.009549	24	2	4.2
24	4_InvertedTree_Convergent	lp_finite_diff	inverted_tree	177.705068	33	2	6.4
25	4_InvertedTree_Convergent	lp_ema_diff	inverted_tree	169.647550	62	3	8.9
26	4_InvertedTree_Convergent	variance	inverted_tree	176.611744	32	3	6.6
27	4_InvertedTree_Convergent	oracle	inverted_tree	175.801671	27	4	7.2
28	5_ComplexDAG_LowTransfer	random	dag	171.140873	79	16	7.6
29	5_ComplexDAG_LowTransfer	topological	dag	20.379538	-1	-1	6.4
30	5_ComplexDAG_LowTransfer	reverse_topological	dag	187.979976	24	2	6.6
31	5_ComplexDAG_LowTransfer	lp_finite_diff	dag	175.494895	65	6	9.3
32	5_ComplexDAG_LowTransfer	lp_ema_diff	dag	144.764146	138	6	1.0
33	5_ComplexDAG_LowTransfer	variance	dag	171.462537	78	17	7.5
34	5_ComplexDAG_LowTransfer	oracle	dag	42.932153	-1	43	1.5

Results DataFrame with Regret Metrics

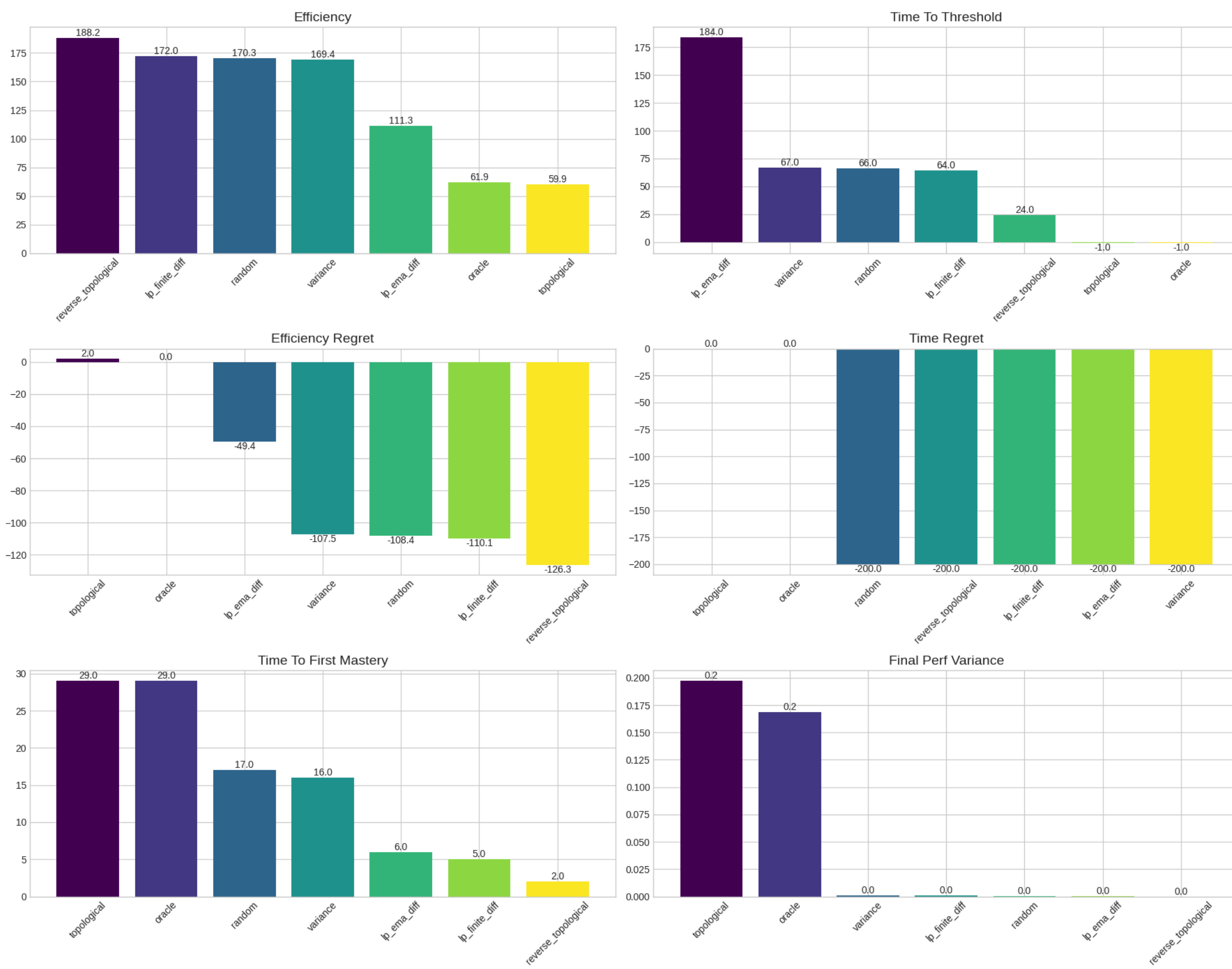
	scenario	curriculum	graph_type	efficiency	time_to_threshold	time_to_first_mastery	final_perf_
0	1_Chain_Simple	random	chain	170.333115	66	17	6.8
1	1_Chain_Simple	topological	chain	59.916556	-1	29	1.9
2	1_Chain_Simple	reverse_topological	chain	188.156021	24	2	0.00
3	1_Chain_Simple	lp_finite_diff	chain	172.032845	64	5	6.9
4	1_Chain_Simple	lp_ema_diff	chain	111.270636	184	6	1.3
5	1_Chain_Simple	variance	chain	169.361163	67	16	7.4
6	1_Chain_Simple	oracle	chain	61.887785	-1	29	1.6
7	2_Chain_HighForget	random	chain	167.824185	63	20	6.6
8	2_Chain_HighForget	topological	chain	58.821264	-1	28	1.7
9	2_Chain_HighForget	reverse_topological	chain	187.235618	24	2	1.1
10	2_Chain_HighForget	lp_finite_diff	chain	172.371547	65	6	7.1
11	2_Chain_HighForget	lp_ema_diff	chain	109.621859	185	6	2.1
12	2_Chain_HighForget	variance	chain	169.088831	67	16	6.7
13	2_Chain_HighForget	oracle	chain	60.436184	-1	29	1.9
14	3_Tree_Divergent	random	tree	186.776147	55	17	7.6
15	3_Tree_Divergent	topological	tree	38.272161	-1	200	8.6
16	3_Tree_Divergent	reverse_topological	tree	188.741723	24	2	8.5
17	3_Tree_Divergent	lp_finite_diff	tree	186.622626	47	13	7.1
18	3_Tree_Divergent	lp_ema_diff	tree	177.935767	81	10	7.8
19	3_Tree_Divergent	variance	tree	186.499868	53	16	6.7
20	3_Tree_Divergent	oracle	tree	31.211558	-1	-1	5.5
21	4_InvertedTree_Convergent	random	inverted_tree	177.925998	32	3	5.4
22	4_InvertedTree_Convergent	topological	inverted_tree	188.712836	23	3	1.1
23	4_InvertedTree_Convergent	reverse_topological	inverted_tree	188.009549	24	2	4.2
24	4_InvertedTree_Convergent	lp_finite_diff	inverted_tree	177.705068	33	2	6.4
25	4_InvertedTree_Convergent	lp_ema_diff	inverted_tree	169.647550	62	3	8.9
26	4_InvertedTree_Convergent	variance	inverted_tree	176.611744	32	3	6.6
27	4_InvertedTree_Convergent	oracle	inverted_tree	175.801671	27	4	7.2
28	5_ComplexDAG_LowTransfer	random	dag	171.140873	79	16	7.6
29	5_ComplexDAG_LowTransfer	topological	dag	20.379538	-1	-1	6.4

30	5_ComplexDAG_LowTransfer	reverse_topological	dag	187.979976	24	2	6.6
31	5_ComplexDAG_LowTransfer	lp_finite_diff	dag	175.494895	65	6	9.3
32	5_ComplexDAG_LowTransfer	lp_ema_diff	dag	144.764146	138	6	1.0
33	5_ComplexDAG_LowTransfer	variance	dag	171.462537	78	17	7.5
34	5_ComplexDAG_LowTransfer	oracle	dag	42.932153	-1	43	1.5

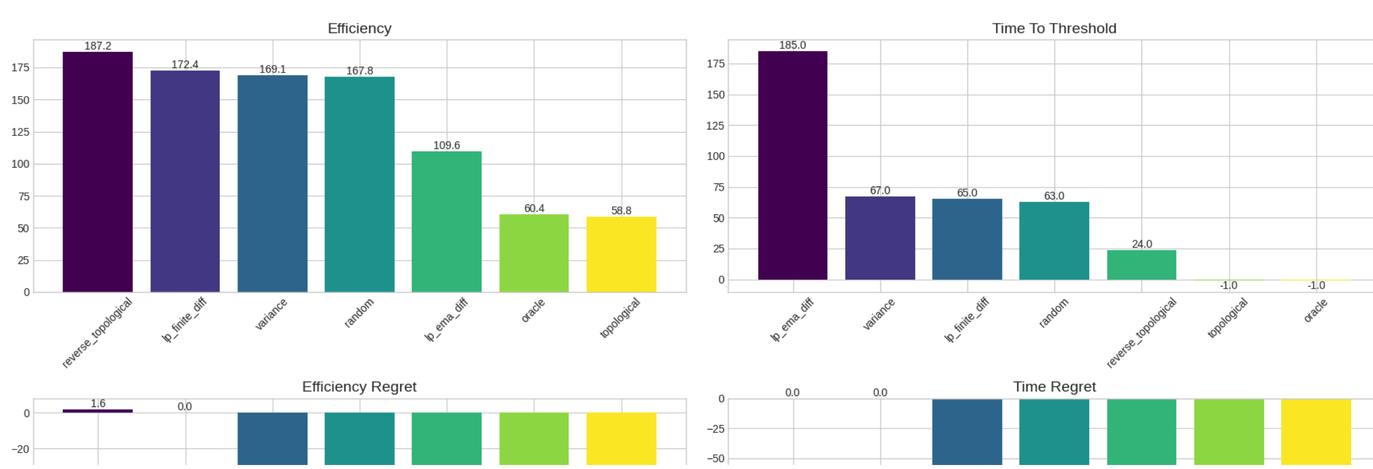
Detailed Bar Chart Comparisons

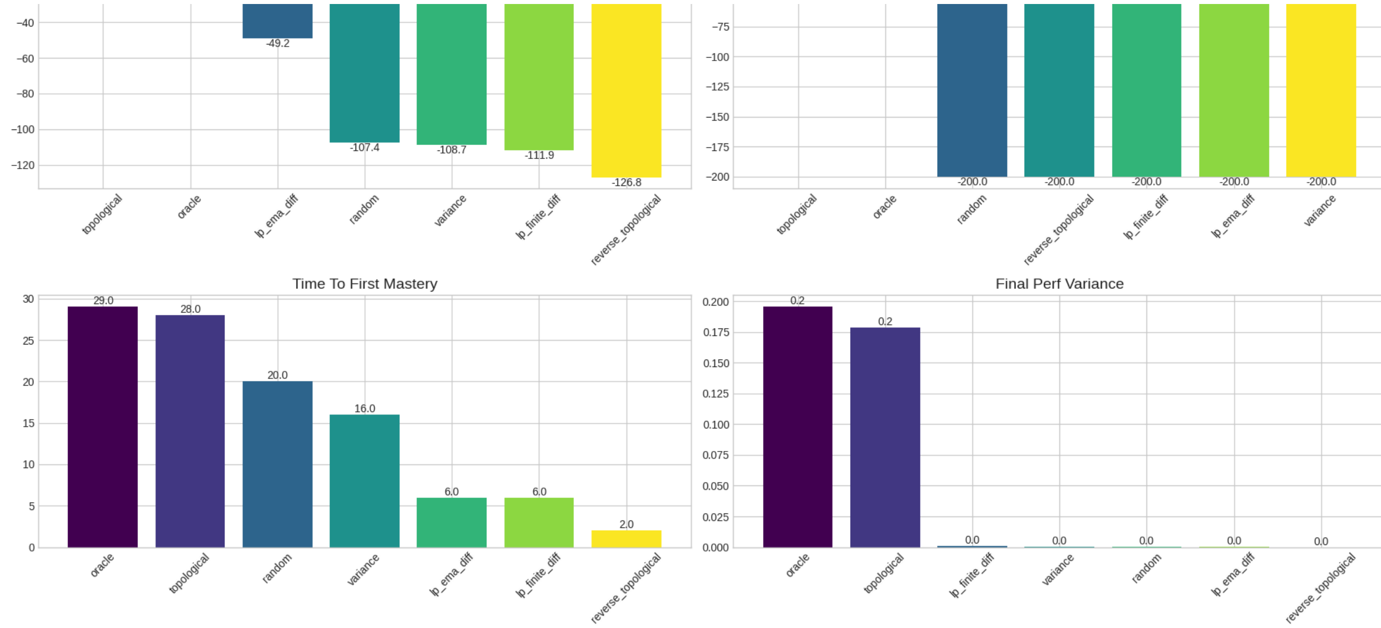
These charts provide a detailed look at curriculum performance in each specific scenario.

Metric Comparison for Scenario: 1_Chain_Simple

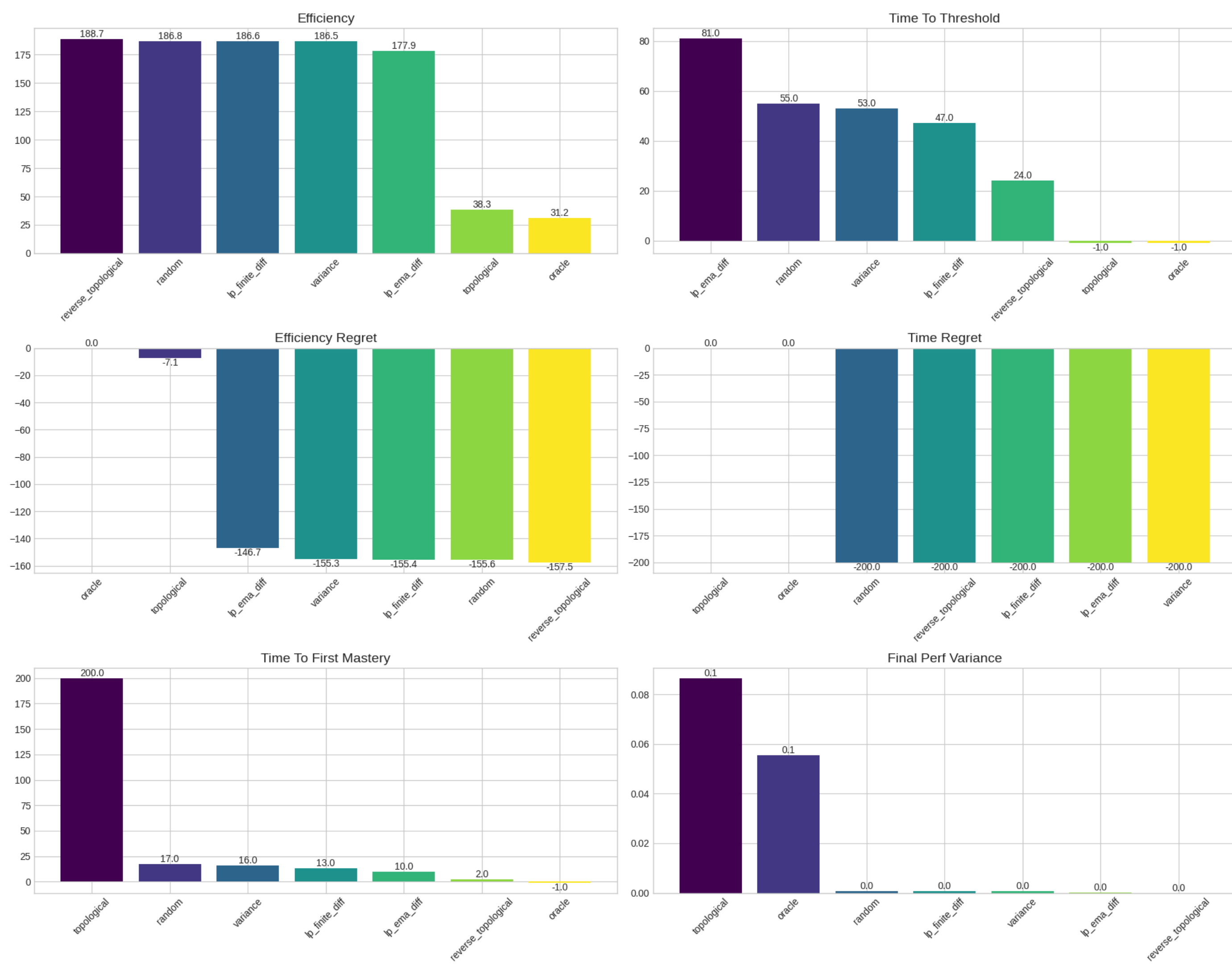


Metric Comparison for Scenario: 2_Chain_HighForget

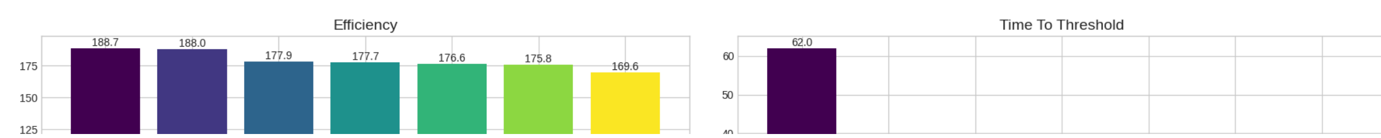


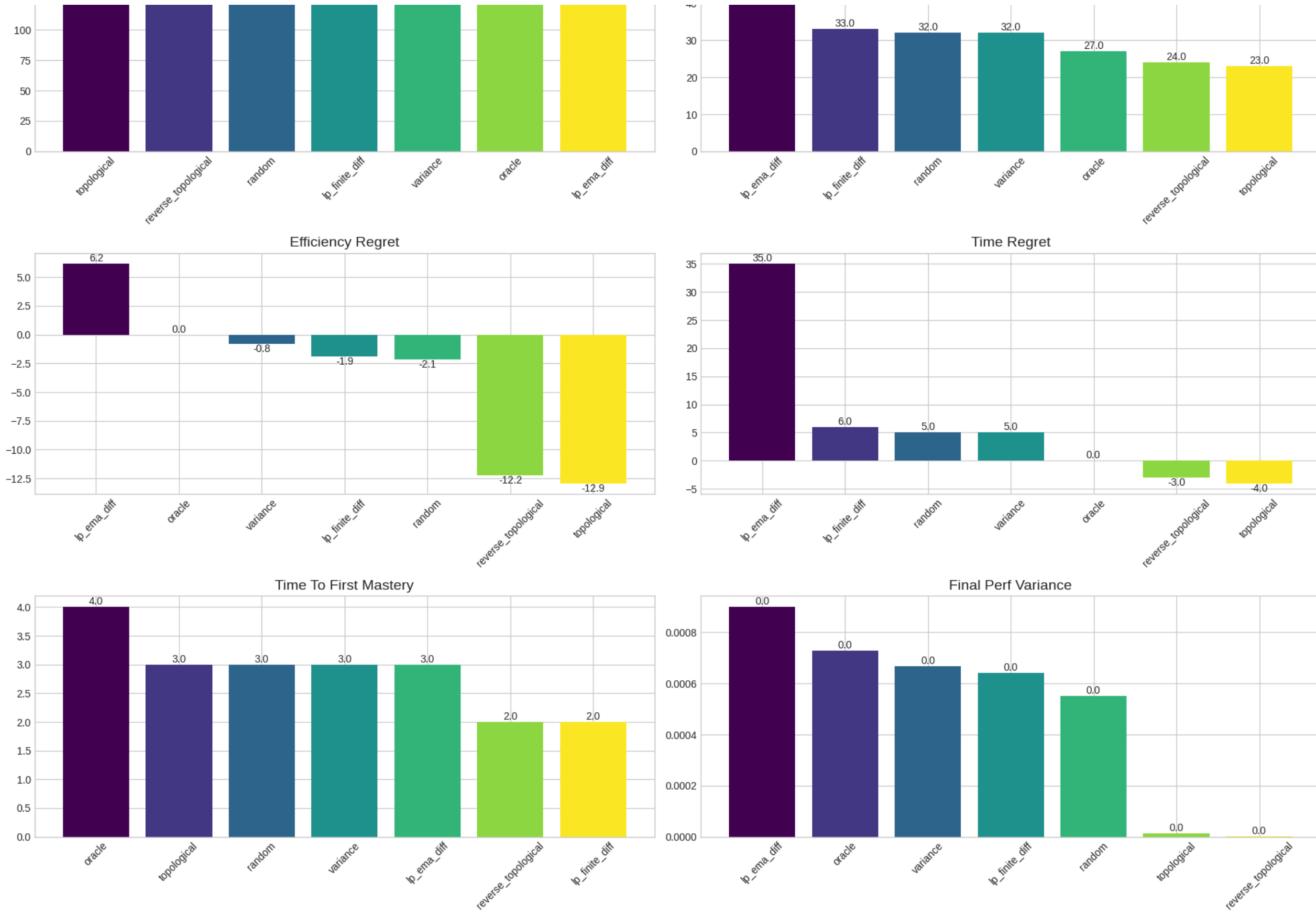


Metric Comparison for Scenario: 3_Tree_Divergent

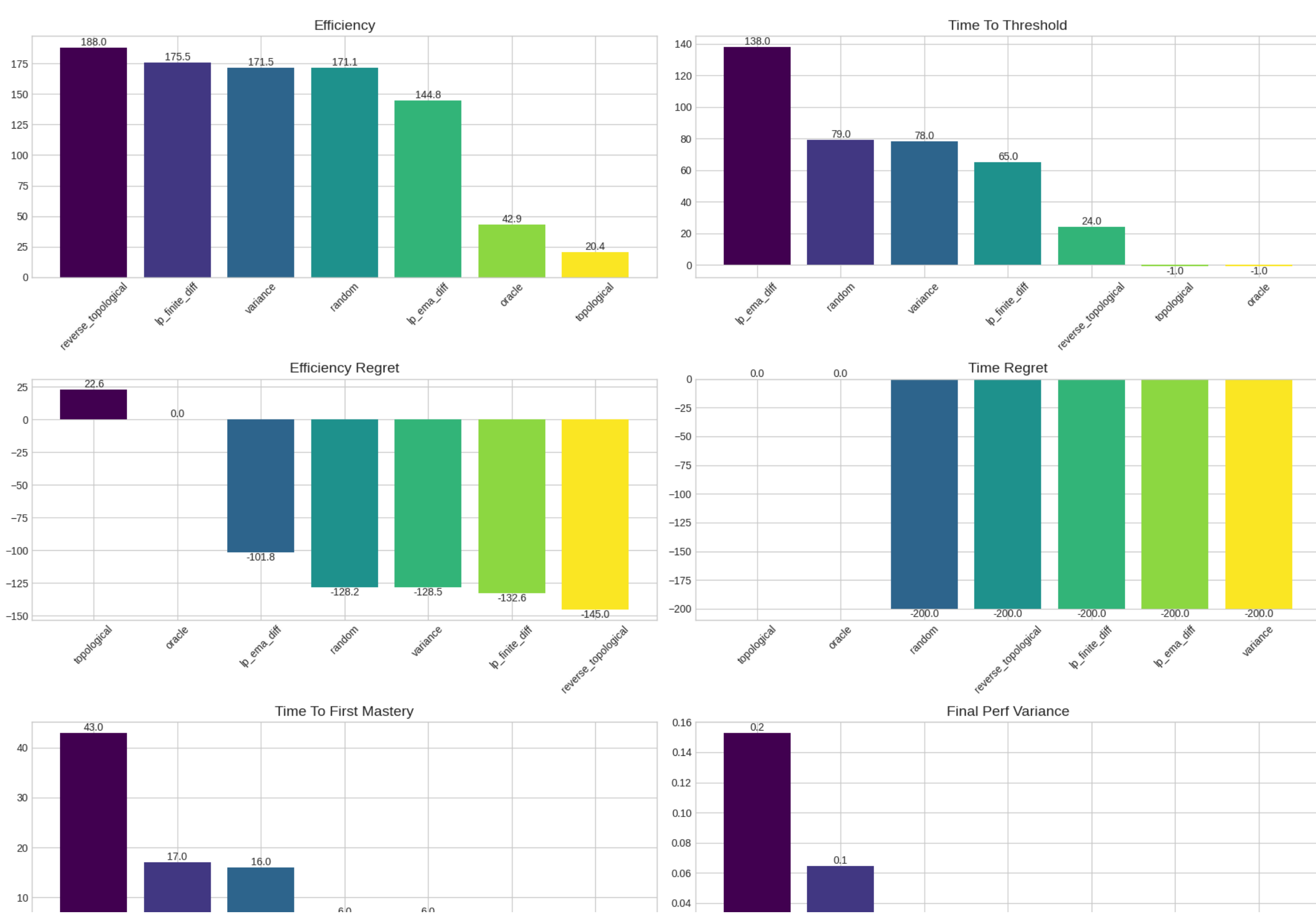


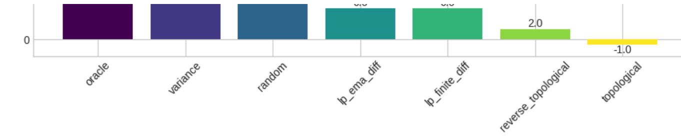
Metric Comparison for Scenario: 4_InvertedTree_Convergent

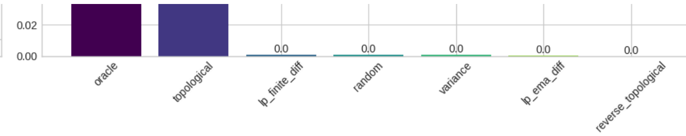




Metric Comparison for Scenario: 5_ComplexDAG_LowTransfer







Global Star Plot Comparison

This star plot is the final, high-level summary. Each point on the star represents a normalized performance metric, where **a larger area is universally better**. This visualization quickly reveals the unique strengths and weaknesses of each curriculum across the different environmental challenges. For example, a curriculum that excels in the 'High Forget' scenario will have a large area in that respective plot.