###### **قال ChatGPT:**

**Summary of “Test Suite Prioritization Based on Optimization Approach Using Reinforcement Learning” (Waqar et al., 2022)**

### **1. Objectives**

* **Reduce regression‑testing cost** by selecting and ordering a minimized subset of test cases that maximizes early fault detection.
* **Leverage reinforcement learning (RL)** to automatically learn which sequences of user/tester interactions yield the highest fault‑detection “reward.”
* **Validate** the RL‑based prioritization against baseline techniques (random ordering and t‑SANT) across five Android applications.

### **2. Methodology**

1. **Data Collection & Preparation**
   * **Interaction Recording System** logs both tester and end‑user GUI events (activity IDs, button clicks) for Android apps.
   * From these logs, extract sequences of states (screens) visited during testing or real‑use sessions.
   * Compute per‑state statistics (frequency, complexity, mean, median) used in reward calculation.
2. **Reinforcement Learning Model**
   * **Environment**: States = screens (S₀…Sₙ), Actions = {up, down, left, right, no op}.
   * **Reward Function** assigns positive/negative values based on state frequency vs. mean/median and complexity:  
      r(s,a)=f(freq(s))×complexity(s)r(s,a) = f(\text{freq}(s)) \times \text{complexity}(s)r(s,a)=f(freq(s))×complexity(s)  
      where f(x)f(x)f(x) gives +λ, +1, or −λ depending on x relative to mean/median, and large positive/negative values at goal/block states to encourage/penalize reaching them.
   * **Q‑Learning** (Bellman updates) over 1,000 episodes yields a Q‑table estimating max future rewards.
3. **Sequence Extraction & Prioritization**
   * **Trace Extraction**: From final Q‑table, follow the highest‑Q actions from start to goal to produce the “highest‑future‑reward” state sequence (e.g. S₀→S₁→S₅→S₉→S₁₀).
   * **Cosine Similarity**: Compute similarity between this RL‑derived sequence and each test‑case state sequence.
   * **Ordering**: Sort test cases by descending similarity; ties broken by longer test‑case length (favoring more comprehensive tests).
4. **Validation via Fault Seeding**
   * Seed faults (computational, initialization, interface, data, logic) into each application per known distributions.
   * Execute **top 30 %** of test cases under three strategies (random, t‑SANT, RL) and measure percentage of seeded faults detected.

### **3. Key Findings**

* **Superior Early Fault Detection**:  
  + Across all five case studies, the RL approach detected an **average of 87.6 %** of seeded faults within the top 30 % of test cases, compared to **82.8 %** for t‑SANT and **48.2 %** for random ordering.
* **Consistent Performance Gains** in diverse domains (healthcare app, vehicle‑assistance system, real‑estate app, POS system, jobs portal).
* **Proof‑of‑Concept Example**: Illustrated on a simplified 11‑state grid, the RL‑derived sequence (S₀→S₁→S₅→S₉→S₁₀) led to correct prioritization of test cases by similarity.

### **4. Contributions & Significance**

* **Four‑Fold RL‑Based Framework** for test‑suite prioritization integrating user/tester logs, RL modeling, sequence extraction, and similarity‑based ordering.
* **Automated Android Regression Tool** that adapts to real interaction patterns rather than static code‑coverage heuristics.
* **Empirical Evidence** demonstrating RL’s efficacy over established baselines in minimizing testing effort while maximizing fault detection.

### **5. Limitations & Future Work**

* **Applicability** tested only on Android apps with moderate test‑suite sizes; scalability to large‑scale systems remains to be explored.
* **Dependency** on quality of interaction logs and the assumption that frequent/complex states correlate with fault‑proneness.
* **Future Directions**:  
  + Integration of **deep RL** (e.g. DQN) for high‑dimensional state spaces.
  + Application to other platforms (web, desktop) and larger codebases.
  + Exploration of alternative reward‑shaping strategies and multi‑objective optimization.

The authors’ workflow can be broken down into the following five main steps:

1. **Interaction Logging and Data Preparation**
   * Instrument each Android application to record GUI events (activity IDs, button clicks) from both testers and real end‑users.
   * Aggregate these logs into sequences of visited “states” (screens) and compute per‑state metrics (frequency, complexity, mean/median) for use in rewards.
2. **Reinforcement‑Learning (RL) Model Setup**
   * Define the RL environment:  
     + **States** are the screens S0,S1,…,SnS\_0, S\_1, …, S\_nS0​,S1​,…,Sn​.
     + **Actions** are the possible GUI transitions (e.g. up, down, left, right, no‑op).
   * Craft a reward function that gives positive or negative rewards based on how a state’s frequency compares to the dataset’s mean/median and on its complexity, with large terminal rewards/penalties for goal/block states.
3. **Q‑Learning Training**
   * Run standard Q‑learning (Bellman updates) for a fixed number of episodes (1,000) to learn a Q‑table Q(s,a)Q(s,a)Q(s,a) estimating the maximum expected future reward for each state–action pair.
4. **High‑Reward Sequence Extraction & Test‑Case Prioritization**
   * From the learned Q‑table, start at the initial state S0S\_0S0​ and greedily follow the highest‑Q action at each step until reaching a goal state—yielding the “optimal” state‑transition sequence.
   * For each existing test case, compute the cosine similarity between its recorded state sequence and the RL‑derived sequence.
   * Sort (prioritize) test cases in **descending** order of similarity, breaking ties by longer test length (favoring more comprehensive tests).
5. **Empirical Validation via Fault Seeding**
   * Seed a variety of faults (initialization, logic, interface, data, etc.) into each app.
   * Execute the top 30 % of test cases under three strategies—random, t‑SANT, and the RL‑based ordering—and measure the percentage of seeded faults detected early.

These steps together form an end‑to‑end pipeline: from raw interaction

The paper you provided is titled **"An Intelligent Edge Computing Platform Using Advanced Bio-inspired Algorithms for Smart City Applications"** by W. Shi, et al., published in 2020. Here’s a summary of its **key objectives, methodologies, and findings**:

### **Key Objectives**

1. **Design and develop** an intelligent edge computing platform for smart city applications.
2. **Improve resource management and task offloading** in edge computing environments using **bio-inspired optimization algorithms**.
3. **Reduce latency and energy consumption**, while enhancing performance and efficiency in smart city services.

### **Methodologies**

1. **Edge Computing Architecture**:  
    A three-tier architecture is proposed:  
   * **Cloud layer**: Centralized processing and storage.
   * **Edge layer**: Local computation near data sources (e.g., base stations, access points).
   * **Device layer**: End-user devices (e.g., sensors, mobile phones).
2. **Bio-inspired Algorithms**:  
    The paper integrates **Artificial Bee Colony (ABC)** and **Particle Swarm Optimization (PSO)** algorithms for:  
   * Task scheduling
   * Load balancing
   * Resource allocation
3. These algorithms simulate natural behavior (bees and bird flocking) to find near-optimal solutions for complex, dynamic systems.
4. **Simulation and Evaluation**:  
    The system is tested using a simulated smart city scenario. Key metrics evaluated include:  
   * Latency
   * Energy consumption
   * Success rate of task offloading

### **Findings**

1. **Performance Gains**:  
    The proposed platform outperforms traditional methods in reducing task latency and energy consumption.
2. **Efficiency in Task Scheduling**:  
    The bio-inspired algorithms effectively optimize edge resource utilization and improve decision-making in dynamic environments.
3. **Suitability for Smart Cities**:  
    The platform shows strong potential for real-time, data-intensive smart city services like traffic control, environmental monitoring, and public safety.

Would you like a diagram summarizing the architecture or algorithms used in this paper?

4o

##### **أنت قلت:**

no, please read last attached file

###### **قال ChatGPT:**

Here’s a concise summary of the key elements from **“Reinforcement Learning Based Test Case Prioritization for Enhancing the Security of Software”** by Shi et al. (2020):

## **Objectives**

1. **Accelerate feedback** in CI regression testing by prioritizing test cases that detect faults early.
2. **Leverage reinforcement learning (RL)** to adaptively learn which test cases are most effective at uncovering defects over time.
3. **Incorporate execution history and time constraints** into the RL reward and selection process to boost fault‐detection rate without excessive test‐run overhead.

## **Methodologies**

### **1. Reinforcement‐Learning Framework (RETECS baseline)**

* **State**: For each test case, its last execution time, duration, and binary pass/fail history.
* **Action**: Assign a priority score used to order tests for the next CI cycle.
* **Reward**: In the original RETECS, failed tests get a reward of 1; all others 0.

### **2. New Reward Functions**

Shi et al. propose **“RHE”** and **“THE”** rewards that weight recent failures more heavily:

| **Name** | **Weight Function w(i)** | **History Length m** | **Strategy** |
| --- | --- | --- | --- |
| **RHE (ReLU)** | w(i) = max(0, i) | either all cycles or last 4 | whole/partial |
| **THE (tanh)** | w(i) = tanh(i), capped at i=4 | last 4 cycles only | whole/partial |

* **Whole vs. Partial**:  
  + *Whole*: Every test gets its weighted‐history sum.
  + *Partial*: Only tests that failed in the current cycle earn rewards; others get zero.

### **3. Multi‐Objective Prioritization**

* After RL assigns base priorities, they form **s priority buckets**.
* Within each bucket, tests are re‐ranked by execution time:  
   pi←pi+(1−teiTmax⁡) p\_i \leftarrow p\_i + \bigl(1 - \tfrac{t\_{e\_i}}{T\_{\max}}\bigr)pi​←pi​+(1−Tmax​tei​​​)  
   so shorter tests move up, maximizing the number of tests run in a fixed time window.

## **Experimental Setup**

* **Datasets** from three industrial suites:  
  1. PaintControl (114 tests, 312 CI cycles, ~19% failure)
  2. IOF/ROL (2,086 tests, 320 cycles, ~28% failure)
  3. GSDTSR (5,555 tests, 336 cycles, ~0.25% failure)
* **Evaluation Metric**: Normalized APFD (NAPFD), capturing early‐fault detection under time‐limited test runs.
* **Comparison** against the original RETECS (tcfail reward) over 30 runs.

## **Findings**

1. **Four‐Cycle History Wins** Reward functions using only the last 4 runs consistently outperformed those using all history, since very old failures often reflect tests that have since been fixed.
2. **ReLU vs. tanh**
   * On high‐failure‐rate suites (PaintControl, IOF/ROL), **ReLU‐weighted (RHE)** yielded slightly better NAPFD.
   * On the large, low‐failure GSDTSR, both performed similarly.
3. **Whole vs. Partial** Little difference in NAPFD—**partial** rewards (only rewarding current‐cycle failures) suffice and save computation time.
4. **Time Overhead** Weighted‐history reward computations (with 4‐cycle windows) add negligible overhead compared to tcfail, and far less than using all history.
5. **Multi‐Objective Prioritization** Incorporating test duration lets you execute more tests in the same CI time budget, further boosting fault‐detection early in the run.

**In short**, by shifting to a ReLU‐weighted, four‐cycle partial‐reward function and adding an execution‐time re‐ranking step, the authors demonstrate clear gains in early fault detection under realistic CI constraints, without extra runtime cost.

o4-mini

##### **أنت قلت:**

what is the steps followed by the author

###### **قال ChatGPT:**

The authors followed a structured approach to design, implement, and evaluate their reinforcement learning-based test case prioritization method. Here's a step-by-step breakdown of the process used in the paper:

### **Step 1: Problem Identification**

* **Challenge**: In Continuous Integration (CI), regression testing needs to be fast and efficient due to time constraints.
* **Goal**: Improve test case prioritization so that fault-revealing test cases are executed earlier and within a limited time budget.

### **Step 2: Use Reinforcement Learning (RL) Framework**

* **Baseline**: Adopt RETECS (Reinforced Test Case Selection), a prior RL-based approach.
* **Agent Design**:  
  + **State**: Each test case's historical execution results, duration, and recent failure information.
  + **Action**: Assign a priority score.
  + **Reward**: Design functions to measure how valuable each test case is for fault detection.

### **Step 3: Propose New Reward Functions**

* Develop **weighted reward functions** that use execution history:  
  + **RHE (ReLU-based)** and **THE (tanh-based)**.
* Consider **two historical scopes**:  
  + Last 4 integration cycles.
  + All available historical records.
* Define **two reward strategies**:  
  + **Whole**: All test cases get a reward based on historical data.
  + **Partial**: Only test cases that fail in the current cycle receive rewards.

### **Step 4: Incorporate Multi-Objective Prioritization**

* Re-rank test cases based on:  
  + Their RL-derived priority.
  + Their execution time (favor shorter ones to fit more tests within time constraints).

### **Step 5: Experimental Setup**

* Use **three real-world datasets** with diverse characteristics.
* Configure a **neural network agent** for RL.
* Set a **time threshold** (50% of total test time) to simulate realistic CI limits.

### **Step 6: Evaluate with Metrics**

* Use **NAPFD (Normalized Average Percentage of Faults Detected)** to evaluate how quickly faults are found.
* Measure **execution time** of the sorting algorithm.
* Conduct **30 trials** for statistical reliability.

### **Step 7: Analyze Results**

* Compare performance across:  
  + Different reward functions.
  + Different historical scopes.
  + Whole vs. partial reward strategies.
  + With and without multi-objective sorting.

### **Step 8: Conclusion and Recommendations**

* Reward functions using **only the last 4 historical cycles** and **partial reward strategy** are most efficient.
* Multi-objective ranking (adding execution time) increases the number of executed fault-revealing tests.
* Recommend **partial RHE or THE with 4-cycle history** for best tradeoff between performance and efficiency.

**Reinforcement Learning for Test Case Prioritization**

Here’s a concise synthesis of the paper’s main points:

**Objectives** - **Address CI-specific challenges**: Design test‑case prioritization techniques that cope with the highly dynamic, time‑constrained nature of Continuous Integration (CI) environments—where test suites and code change constantly, and regression testing must remain fast.  
 - **Leverage Reinforcement Learning (RL)**: Go beyond static, heuristic‑ or batch‑ML‑based methods by continuously adapting a prioritization strategy via RL, aiming to match as closely as possible the “optimal” ordering of test cases (failures first, then shortest tests).

**Methodologies**

1. **Problem formalization as RL**
   * Model the interaction between a CI environment and a prioritizer as an episodic RL task, where at each cycle the agent:  
     + Observes features of test cases (execution history ± lightweight code metrics)
     + Selects actions that induce a ranking (via pointwise, pairwise, or listwise schemes)
     + Receives rewards reflecting closeness to the “optimal” rank (fail‑first, then fast‑test)
2. **Three ranking formulations**
   * **Pointwise**: Assign each test a score in [0,1], then sort by score.
   * **Pairwise**: Compare test‑pairs (which should go first) and sort via a comparison‑based algorithm (e.g. merge‑sort).
   * **Listwise**: At each step pick the next highest‑priority test from the full remaining list.
3. **State‑of‑the‑art RL algorithms**
   * Evaluate 21 configurations combining the three ranking models with ten modern model‑free RL methods (e.g., A2C, DQN, PPO, ACER, SAC) from Stable Baselines.
4. **Extensive offline training & evaluation**
   * Use a “CI‑replayer” simulator to train on historical CI cycles (simple datasets: only history; enriched: history + code metrics)
   * Measure ranking accuracy per cycle, then update the agent with new logs (no build‑time delays).

**Findings** - **Pairwise + ACER wins**: The actor‑critic–based ACER algorithm, combined with a pairwise ranking model, achieved the highest Normalized Rank Percentile Average (NRPA), significantly outperforming prior RL work and even MART, the best batch‑ML baseline.  
 - **Near‑optimal accuracy on enriched data**: With lightweight code features added, NRPA exceeded **0.96**, i.e. the learned strategies were almost indistinguishable from the oracle.  
 - **Training cost practical**: Although different RL methods vary in training time, the overhead remains acceptable and does not impede CI workflows.

By recasting test‑case prioritization under RL—and systematically exploring ranking models and modern algorithms—the paper demonstrates that adaptive, RL‑based strategies can match or exceed the accuracy of static ML approaches while seamlessly handling CI’s evolving context.

المصادر

o4-mini

##### **أنت قلت:**

what is the steps followed by the author

###### **قال ChatGPT:**

The authors of the paper followed a structured, multi-phase approach to develop and evaluate their reinforcement learning (RL)-based test case prioritization strategy for Continuous Integration (CI). Here are the key steps they followed:

### **1. Problem Definition and Motivation**

* Identified the **need for adaptive test case prioritization** in CI environments, where:  
  + Test suites and code evolve frequently.
  + Test execution time is limited.
  + Traditional static or batch ML approaches fall short.

### **2. Formalization as a Reinforcement Learning Problem**

* Defined the test case prioritization task as an **episodic RL problem**:  
  + **State**: Features of test cases (historical pass/fail status, execution time, and optionally code metrics).
  + **Action**: Assigning priority scores or orderings to test cases.
  + **Reward**: A function based on how closely the agent’s prioritization matches the "optimal" ordering (failures first, then short duration).

### **3. Design of Three Ranking Models**

* Implemented three ranking models to convert RL actions into test prioritization:  
  + **Pointwise**: Assign a priority score to each test independently.
  + **Pairwise**: Learn to compare test pairs and determine which should be prioritized.
  + **Listwise**: Choose the next test to add to the ranking iteratively from the remaining list.

### **4. Selection of RL Algorithms**

* Evaluated **10 state-of-the-art model-free RL algorithms** from the *Stable Baselines* library:  
  + Algorithms included DQN, PPO, A2C, ACER, SAC, etc.
* Combined each algorithm with each ranking model → total of **21 agent configurations**.

### **5. Offline Training Using Historical CI Data**

* Developed a **CI-replayer simulator** to replay past CI cycles without impacting real-time builds.
* Used two datasets:  
  + **Simple**: Based only on test execution history.
  + **Enriched**: Includes code metrics in addition to historical data.
* Trained agents on 75% of CI cycles and validated on the remaining 25%.

### **6. Evaluation and Metrics**

* Compared RL agent performance to baselines (random, historical ordering, and MART – a top-performing ML model).
* Used **Normalized Rank Percentile Average (NRPA)** as the primary evaluation metric:  
  + Measures how close the predicted ranking is to the ideal.

### **7. Analysis and Results**

* Identified the best-performing combination: **Pairwise + ACER**.
* Achieved high NRPA scores (>0.96 on enriched dataset).
* Showed training time was acceptable and did not introduce build delays.

### **8. Conclusions and Contributions**

* Demonstrated that RL can dynamically and effectively prioritize test cases in CI.
* Highlighted benefits of adaptive learning over static ML models.
* Proposed an extensible framework for applying RL to CI testing workflows.