ChatGPT

UMACO Developer Guide

Overview

Universal Multi-Agent Cognitive Optimization (UMACO) is a general-purpose optimization framework that combines swarm intelligence, multi-agent systems, topological data analysis, quantum-inspired strategies, and economic principles into a unified solver. UMACO is **solver-agnostic** – it can tackle a wide range of optimization problems (from mathematical functions to SAT puzzles to machine learning training) by configuring its modular components rather than requiring problem-specific logic. This guide provides a high-level architectural overview of UMACO's core ideas and a detailed developer reference for its components. It is intended for software engineers and practitioners who want to understand, use, or extend UMACO without needing deep expertise in machine learning or satisfiability solving.

UMACO's design introduces several novel concepts to avoid common pitfalls like local optima and stagnation in complex search spaces. At a high level, UMACO orchestrates a **colony of cognitive agents** that cooperatively explore a solution space. These agents interact indirectly through a shared **pheromone environment** (inspired by ant colony optimization) and directly via a **token-based economy** (inspired by computational market systems). The system monitors its own progress using a **Panic–Anxiety–Quantum (PAQ) Triad** mechanism to detect stagnation or "crisis" and trigger dynamic responses (like disruptive "quantum" leaps) when needed. Key hyperparameters automatically adjust based on system state (no manual tuning of, e.g., exploration/exploitation balance is needed). The result is an optimization approach that is adaptive, decentralized, and domain-independent.

In the sections below, we will first explain UMACO's core concepts (PAQ triad, Topological Stigmergy, Quantum Economy, etc.) in an intuitive way. We then document each module and class in the codebase, including their functions and how they interrelate, so you can effectively utilize and extend the framework. Finally, we provide example use cases – solving a general math function, tackling a Boolean SAT problem, and fine-tuning a machine learning model – to illustrate how UMACO can be applied in different domains without altering its fundamental architecture.

Architecture and Core Concepts

UMACO's architecture is best understood as the integration of four conceptual pillars:

- **1) Panic–Anxiety–Quantum (PAQ) Triad:** A self-monitoring mechanism that detects when the search is stuck in a poor state and triggers escape strategies.
- **2) Topological Stigmergy Field (TSF):** A global pheromone matrix that agents update to mark promising or unpromising regions, analyzed using topological methods to guide exploration.
- **3) Universal Economy:** A token economy where multiple agent "nodes" earn and spend tokens, encouraging diverse strategies and resource allocation based on performance.
- **4) Dynamic Hyperparameter Adaptation:** Key algorithm parameters (pheromone influence, evaporation rates, etc.) adjust on the fly according to system feedback (panic/anxiety levels and topological signals).

These work together to produce an **adaptive multi-agent optimizer**. The diagram below shows a high-level view of UMACO's components and their interactions:

```
flowchart LR
    subgraph UMACO Solver
        direction TB
        Pheromone["NeuroPheromone Field\n(Topological Stigmergy)"]
        Agents["Cognitive Nodes (Agents)"]
        Economy["Universal Economy\n(Token Market)"]
        PAQ["PAQ Triad\n(Panic, Anxiety, Quantum)"]
    end
    Problem["Optimization Problem\n(Objective/Loss)"]:::ext

    Agents -->|update| Pheromone
    Pheromone -->|provides landscape| Agents
    Agents -- tokens --> Economy
    Economy -- resources --> Agents
    PAQ -->|monitors & perturbs| Pheromone
    Pheromone -->|persistent homology| PAQ
    PAQ -->|adjusts strategy| Agents & Economy
    Problem -->|evaluated by| Pheromone
```

(In the above: Agents explore the problem's solution space and leave pheromone trails; the pheromone field's topology is analyzed (persistent homology) to update the PAQ state; the PAQ triad modifies pheromones or triggers "quantum" jumps when stuck; agents earn/spend tokens via the economy based on their success, influencing how they use resources; all of this runs iteratively against the problem's objective function.)

Let's break down each core idea in more detail:

Panic–Anxiety–Quantum (PAQ) Triad

UMACO introduces a novel **PAQ triad** to manage the algorithm's self-awareness and escape from local optima:

- **Panic:** The system maintains a **Panic Tensor**, which is essentially a matrix accumulating indications of "frustration" or difficulty encountered in the search. If the algorithm's recent moves aren't improving the outcome, the panic tensor values increase (akin to a gradient accumulation of loss). Think of panic as a measure of how urgently the system feels it needs to escape the current region. In implementation, the panic tensor is updated by a method `panic_backpropagate(...)` which takes a gradient-like signal of the current loss and increases panic where the loss gradient is large. Over iterations, if parts of the search space consistently yield high loss, panic will concentrate there.

- **Anxiety:** Complementing panic, UMACO keeps an **Anxiety Wavefunction** – a complex-valued vector or matrix that oscillates based on uncertainty and lack of progress. Anxiety is a bit like "momentum" or nervous energy in the system. When the solution isn't improving, the anxiety level grows (with both real and imaginary components carrying different meanings). For example, the real part of the wavefunction might increase with prolonged stagnation (general unease), and the imaginary part might encode a creative/destructive tendency to try wildly different moves. The code updates anxiety through mechanisms like `update_anxiety()`,

which raises anxiety when performance stalls and even adds an imaginary component when stagnation persists. In effect, anxiety represents the system's readiness to try something radical due to lack of recent progress.

- **Quantum Burst:** When panic and anxiety reach critical levels, UMACO triggers a **Quantum Burst** – a non-local, disruptive search jump designed to shake the solution out of a local optimum. This is inspired by quantum tunneling or leaps. In practice, a quantum burst is implemented by taking the singular value decomposition (SVD) of the pheromone matrix and injecting a sudden perturbation constructed from top singular vectors. The method `quantum_burst()` computes an SVD on the pheromone matrix and replaces or perturbs it using the top components, effectively performing a structured randomization. The result is that the solution state is "shocked" into a new configuration far from the current one, hopefully bypassing the trap it was in. The panic tensor is then reset/lowered, and the anxiety wavefunction collapses (reduced) after the burst. UMACO also enforces scheduled quantum bursts at a fixed interval (e.g., every `quantum_burst_interval` iterations) as a precautionary measure in long runs.

The PAQ triad works as a **crisis response system**. Under normal conditions, panic and anxiety remain low, and the solver behaves like a conventional optimizer. If progress stalls (loss doesn't improve over many iterations), panic and anxiety gradually accumulate. This will dynamically adjust some algorithm parameters (as described later) to try different exploration strategies. If that still fails and panic/anxiety exceed thresholds, a quantum burst is triggered – effectively a global restart guided by learned patterns. This triad allows UMACO to repeatedly escape local minima without manual intervention.

Topological Stigmergy Field (Pheromone Matrix and Homology)

UMACO uses **stigmergy** (indirect coordination through the environment) via a 2D **pheromone field**. This is analogous to how ants deposit pheromones on paths: agents in UMACO collectively "write" information to a shared matrix that represents the problem landscape. Here's how it works:

- The pheromone matrix (often denoted `pheromones`) is a **grid of complex numbers**. Each entry can be thought of as a marker of solution quality or path frequency between two state components. The **real part** of a pheromone value represents an attractive signal (desirable path) and the **imaginary part** represents a repulsive signal or alternative pathway. For example, in a path-finding or combinatorial problem, an agent might deposit pheromone on the moves it took that led to a good solution (increasing those real values to encourage other agents to try them), while imaginary components could represent diversification cues.

- Agents update this field through methods like `deposit_pheromones()`. Each agent, after evaluating its solution or partial solution, will evaporate the field slightly (global evaporation ensures older information fades) and then deposit new pheromone on the entries corresponding to its successful moves. The deposit amount might be proportional to the agent's performance (higher if the agent found a better solution). UMACO's pheromone update is flexible: it can work for continuous domains by using diagonal entries (as in the Rosenbrock function example) or for discrete sequences by treating indices as points in a graph.

- The term **"NeuroPheromone System"** in UMACO refers to an enhanced pheromone mechanism that draws inspiration from neural pathways. It not only stores pheromone levels but also maintains a `pathway_graph` that tracks connections or transitions the agents explore. This can be thought of as a weighted graph of solution transitions, updated in parallel with the

pheromone matrix. The system can perform operations like partial resets of pheromones: if certain pheromone values become very low (meaning those paths haven't been used at all), a partial reset will raise them to a baseline to reintroduce those options (preventing a premature convergence where all pheromones except one path have evaporated).

- **Topological Analysis:** What makes UMACO's stigmergy "topological" is that it uses tools from **Persistent Homology** to analyze the shape of the pheromone landscape. In simpler terms, the solver periodically takes the real part of the pheromone matrix as a kind of "height map" and computes its topological features: connected components, holes, clusters, etc. This is done via persistent homology (using libraries like Ripser and PersIm). The result is a summary (persistence diagrams) of the landscape's structure – for instance, how many distinct attractor basins or loops might be present in the pheromone distribution. UMACO uses this information to adjust strategy: one metric derived is **persistent entropy**, which measures the complexity of the pheromone landscape. If the persistent entropy is high, the pheromone values are diverse (many peaks and valleys), indicating exploration; if it's low, pheromones have converged (one big peak, meaning potential exploitation of one area).

- **Influence on Parameters:** The topological feedback influences parameters like **β (beta)** in the algorithm. In ant colony algorithms, beta often controls how much weight pheromone vs. heuristic has. In UMACO, beta is dynamically updated based on persistent entropy: e.g., `beta = 0.1 * entropy`. So, when the pheromone distribution is very uniform or simple (low entropy), beta decreases (less reliance on pheromones, to encourage more exploration or noise). If the pheromone landscape is complex (high entropy), beta increases to intensify the exploitation of the discovered structure. This is part of the **Dynamic Hyperparameter Adaptation** we will explain soon.

In summary, the **Topological Stigmergy Field** allows agents to cooperatively mark what areas of the search space are promising. It provides collective memory. The persistent homology analysis of this field gives a macro-level view of how search is progressing (e.g., have we converged to one cluster of pheromone, or are there many competing regions?), which then feeds back into strategy adjustments.

Universal Economy and Cognitive Agents

Unlike traditional optimization algorithms, UMACO explicitly incorporates a **multi-agent system**. Here, each agent (formally a **UniversalNode**) is a cognitive entity exploring the solution space. To coordinate agents and promote a healthy exploration-vs-exploitation dynamic, UMACO implements a **Universal Economy** where agents use **tokens** as currency to "buy" the ability to utilize resources or make large moves.

Key aspects of the economy and agents:

- Each agent is initialized with a certain number of tokens (configurable, e.g., `initial_tokens` per agent). Tokens represent a generic resource – it could be computational budget, permission to explore far from the current solution, or simply a notional reward system. Agents **spend tokens to perform expensive actions** and **earn tokens when they find good solutions**.

- The economy is managed by the `UniversalEconomy` class. It keeps track of each agent's token balance in a dictionary. Agents interact with the economy via two main actions:

- **Buying Resources:** Before an agent undertakes a costly exploration (for example, a very divergent random jump or an extensive local search), it must "buy" resources. In code, an agent calls `economy.buy_resources(agent_id, required_power, scarcity_factor)` – if the agent has enough tokens to cover the cost (which is determined by how scarce resources are and how big a request it is), the purchase succeeds and the agent can proceed. If not, the agent's request is denied (and in the simple economy, the agent might then increase its internal panic as a result).

- **Rewarding Performance:** After an agent completes its iteration and we see the outcome (e.g., it improved the solution or found a better loss), the economy rewards it via `economy.reward_performance(agent_id, performance)`. The performance measure is typically something like a normalized fitness (for example, UMACO uses `performance = 1/(1+loss)` for a minimization problem, so higher is better). The reward function converts that into some token amount (e.g., `int(performance * 100 * token_reward_factor)`) and adds to the agent's token balance. Thus, agents that consistently perform well become "wealthier" over time.

- **Agent Behavior:** Each `UniversalNode` agent has its own internal state. It tracks a **panic level** (initially set by `panic_level_init` configuration). If an agent fails to get resources (because it lacked tokens), it increases its panic (meaning it becomes more desperate and perhaps will attempt riskier moves next time). Agents also record their performance history. The `UniversalNode.propose_action(current_loss, scarcity)` method encapsulates how an agent decides what to do each iteration:

- It computes a performance score from the current loss.
- It updates its panic based on performance trend (if the agent's performance is dropping relative to two iterations ago, it raises panic).
- It then decides how much resource it wants: e.g., `required_power = 0.2 + 0.3 * panic * risk_appetite`. So an agent with high panic and high risk appetite will request more resources (meaning it wants to try a bigger or more expensive step).
- It asks the economy to buy that amount of resource (`buy_resources`). If the purchase fails (not enough tokens), the agent increases its panic (since it couldn't act as planned).
- Regardless, it then calls `economy.reward_performance` to get tokens based on the performance for this iteration.

In essence, agents self-regulate: a poorly performing agent gets fewer tokens and more panic, which might make it try something drastic, whereas a well-performing agent accumulates tokens and might specialize further or continue its current strategy with more resources at hand.

- **Exploration-Exploitation through Economy:** Over time, an emergent effect happens – **agent specialization and balancing**. Agents that find good solutions gain tokens and can afford to further exploit/refine those solutions (they can keep buying resources to go deeper). Agents that perform poorly lose relative ground; if they run low on tokens, they can only afford small moves unless they get a performance breakthrough (which would reward them). However, UMACO's economy typically enforces a minimum token balance so that no agent is completely bankrupt – this ensures even underperformers can continue to participate (preventing premature convergence to only one agent's approach). The economy can introduce a bit of "inflation" or token redistribution to ensure the market remains dynamic. For example, each iteration the `update_market_dynamics()` might adjust the global token values (reducing everyone's

tokens slightly to simulate inflation, or adding tokens if the total supply is too low) and it introduces randomness in a market value to mimic changing conditions.

- **Scarcity and Market Value:** The economy also tracks a `market_value` and uses a `scarcity_factor`. These concepts influence pricing of resource costs. For example, if agents collectively have left the pheromone field very sparse (meaning not much progress, implying a lot of uncertainty) or if the economy's market value is low (simulating a "recession" in resource availability), the cost of large resource requests might be higher. In UMACO's basic economy, scarcity is calculated from pheromone mean and market value (e.g., `scarcity = 0.7 * mean_pheromone + 0.3 * (1 - market_value)`). A high scarcity makes `buy_resources` costlier for agents. This mechanism keeps agents from all making big moves at once – only those with enough tokens and truly high panic will pay the price.

The **Universal Economy** thus provides a game-theoretic layer to the optimization. It prevents all agents from naively doing the same thing because they have to **budget their exploratory moves**. Successful agents are implicitly encouraged to exploit (they have the tokens to keep digging where they found success), while struggling agents either have to accept small steps or hope for a token injection by some means (which could come from the economy's periodic interventions). This creates a self-organizing exploration/exploitation balance.

Dynamic Hyperparameter Adaptation

Traditional algorithms often have fixed hyperparameters (like learning rates, evaporation rates, etc.) chosen by the user. UMACO, by contrast, **adapts key hyperparameters on the fly** based on the PAQ state and topological cues. This reduces the burden of tuning and allows the algorithm to **respond to different phases of the search** (exploration vs exploitation, etc.) autonomously.

The main parameters that UMACO adjusts dynamically include:

- **Pheromone Update Coefficients (α, β, ρ):** In ant-colony terms, α (alpha) typically scales the influence of pheromone trails when constructing new solutions, β scales the influence of heuristic or problem-specific information, and ρ (rho) is the evaporation rate (how quickly pheromones evaporate). In UMACO:
- `alpha` is treated as a complex number, where the real part scales how much pheromone influences the next update step. UMACO updates `alpha`'s real part based on panic and anxiety: e.g., `alpha_real = panic_mean * anxiety_amplitude`. So when panic and anxiety are high (crisis), the algorithm increases alpha, meaning it will inject more pheromone-driven momentum in the next moves (or potentially add more bias to pheromone updates). The imaginary part of alpha could be used for a phase shift in updating imaginary pheromone components (though by default it might remain 0).
- `beta` is adjusted using the persistent entropy of the pheromone field (from TSF analysis). The code sets `beta = 0.1 * entropy` (with some fallback decay if the homology computation fails). This means if the pheromone distribution is very complex, beta goes up (giving more weight to the learned pheromone landscape in guiding agent moves); if the distribution is simple, beta goes down (agents rely relatively more on randomness or any built-in problem heuristic).

- `rho` (evaporation rate or analogous parameter) is adjusted based on recent "momentum" in the pheromone field. UMACO monitors the norm of the **covariant momentum** (an internal complex matrix tracking recent pheromone change direction). If the momentum norm is high (pheromones are changing a lot), it might reduce evaporation to preserve those changes; if

momentum is low, it could increase evaporation to shake things up. In code, we see `rho = 0.9 * rho + 0.1 * exp(-momentum_norm)`, so if momentum_norm is large (fast changes), exp(-large) is very small, making rho tend toward a smaller value (less evaporation dampening the changes less). If momentum is small (steady state), exp(-small) ~ 1, so rho tends back up (more evaporation to encourage new exploration).

- **Crisis Triggers:** Some hyperparameters are effectively thresholds for the PAQ triggers:

- `partial_reset_threshold` – number of iterations of stagnation after which a partial pheromone reset occurs. If the internal `stagnation_counter` (how many consecutive iterations without improvement) exceeds this, the pheromone matrix will partially reset (low values set to baseline as described in the pheromone system).
- `quantum_burst_interval` – sets a schedule for forced quantum bursts. Even if panic is not extremely high, every `N` iterations the solver will do a quantum burst as a safety net. This interval itself could be dynamic or configured per domain (for example, one might choose a shorter interval for extremely rugged landscapes).

- **Neurochemical Analogs (in advanced use-cases):** In the extended "MACO+" configuration for things like LLM training (discussed later), UMACO introduces analogs of neurochemicals which essentially are additional hyperparameters that modulate behavior:

- *Myrcene factor* (stability control) – dampens large oscillations in pheromone when anxiety is high.
- *Limonene factor* (creativity boost) – scales the magnitude of quantum bursts injected when stagnation is detected.
- *Pinene factor* (pathway clarity) – adjusts how strongly pheromone trails are reinforced versus flattened based on variance (ensuring one path doesn't overshadow others too quickly).
- *Linalool factor* (anxiety management) – controls smoothing of the pheromone's imaginary (exploratory) component to calm the system if anxiety goes too high.

These factors are part of the **NeuroPheromoneSystem** in the advanced context and we will see how they are applied in the code (e.g., exponential dampening by myrcene on real pheromone values, or multiplying the burst pattern by limonene).

The dynamic adaptation of these parameters is **tightly integrated with PAQ and TSF**. The solver continually monitors its state each iteration (panic level, anxiety amplitude, entropy of pheromones, improvement or stagnation status) and tweaks α, β, ρ, etc. accordingly. This means UMACO can, for example: - Start in a highly exploratory mode (if everything is flat and uncertain, keep evaporation high and pheromone influence low). - Gradually shift to exploitative mode as structures appear in pheromone (increase β, lower random noise). - If stuck (panic high), increase α and do bursts to explore radically new territory. - If one agent is dominating (others stagnating), maybe economy triggers a redistribution (a kind of "quantum burst" in economy, described later) to give lagging agents a chance and avoid premature convergence.

All these adaptations happen without user intervention, making UMACO robust across problem types with minimal tuning. The user mainly needs to provide appropriate domain representations (how solutions map to pheromone matrix) and a loss/objective function; UMACO's internal regulators handle the rest.

Having covered the conceptual pieces, we now move on to the **module-by-module documentation** of UMACO's implementation. This will tie the above concepts to actual classes, methods, and their usage in the codebase.

Core Components and Modules

UMACO's code is organized into a Python package (e.g., `umaco` module) with core classes, and a set of example scripts demonstrating domain-specific usage. In this section, we document each core component (class or function) in the framework, explaining its role and API. Think of this as the developer reference for UMACO's internals.

UMACO9 Solver and Configuration

UMACO9 is the main solver class in the core module (in `umaco/Umaco9.py`). The "9" denotes the version; it integrates all the core ideas described earlier. This is the class you instantiate to run an optimization.

- **Class** `UMAC09Config`**:** a configuration dataclass that holds hyperparameters for the UMACO9 solver. Its fields include:
- `n_dim` (int): The dimensionality of the pheromone matrix (and problem space). For many problems this might equal the number of variables or degrees of freedom. The pheromone matrix will be n_dim x n_dim.
- `panic_seed` (np.ndarray): An initial matrix to seed the Panic Tensor. Typically this is a starting pheromone-like matrix or a representation of an initial guess. It must be of shape (n_dim, n_dim). The UMACO9 constructor will raise an error if not.
- `trauma_factor` (float): Initial scaling factor for the anxiety wavefunction. Essentially how "traumatized" the system starts – higher means starting with more anxiety baseline.
- `alpha` (float): Base value for α, the pheromone influence parameter. This will be converted to a complex number internally. (Initial imaginary part is 0, but the real part will change during runtime with PAQ).
- `beta` (float): Base value for β, the weight on pheromone vs heuristic influence. Also adapted via persistent entropy.
- `rho` (float): Base value for ρ, controlling pheromone evaporation or similar dampening. Also adapted dynamically.
- `max_iter` (int): Maximum number of iterations for the optimize loop.
- `target_entropy` (float): A target value for the persistent entropy of the pheromone field. The algorithm injects a bit of noise to the pheromone matrix if the entropy strays too far from this target, to maintain diversity.
- `partial_reset_threshold` (int): The stagnation count after which a partial pheromone reset is triggered. E.g., if no improvement for this many iterations, do a partial reset.
- `quantum_burst_interval` (int): The fixed interval (in iterations) for forced quantum bursts. For example, if 100, then every 100th iteration a burst occurs regardless of panic.

- **Class** `UMAC09`**:** the solver that uses the above config along with an economy and pheromone system. Key aspects and methods of `UMAC09`:

- **Constructor** `__init__(config: UMACO9Config, economy: UniversalEconomy, pheromones: NeuroPheromoneSystem)`**:** Initializes the solver. It:
    - Sets up the **PAQ core**: creates `self.panic_tensor` as a Cupy GPU array (same shape as config.panic_seed, initialized with the given seed values) and `self.anxiety_wavefunction` as a complex matrix of shape (n_dim, n_dim) initialized to zero then scaled by `trauma_factor`. These represent the global panic and anxiety states.
    - Initializes the **TSF**: stores the `pheromones` object (the NeuroPheromoneSystem passed in) and creates a `covariant_momentum` matrix (same shape) initialized with a small imaginary constant (e.g., 0.01j everywhere). This momentum will accumulate changes in pheromones each iteration.
    - Sets up initial **hyperparameters**: e.g., `self.alpha = alpha + 0j` as a complex64, `self.beta = beta`, `self.rho = rho` from config.
    - Stores iteration counters and histories: `max_iter`, `target_entropy`, an empty list `quantum_burst_history`, an empty `panic_history` list, etc. Also sets `stagnation_counter = 0`, `best_score = -inf` (to track best performance seen), and a `burst_countdown = quantum_burst_interval` that counts down to the next scheduled burst.
    - Prepares **topology tools**: creates instances of `Rips()` and `PersistenceImager()` from the Ripser/persim libraries for persistent homology calculations. Also `self.homology_report = None` initially (will hold latest homology result).
    - Saves the reference to the passed-in `economy` (UniversalEconomy) for multi-agent interactions.
- **Method** `panic_backpropagate(loss_grad: cp.ndarray)`**:** Part of PAQ core. It updates the panic tensor given a "gradient" of the loss. Internally, it multiplies the absolute gradient by the log of `(1 + |anxiety_wavefunction|)` to modulate it, then mixes this into the panic tensor (e.g., 85% previous panic + 15% new tanh of that magnitude). This means if the loss gradient is large in some area and anxiety is non-negligible, panic will increase there. This method is called each iteration with an approximation of loss gradient (even if a true gradient is not available, UMACO uses a heuristic one as seen in optimize).
- **Method** `quantum_burst()`**:** Executes the quantum leap mechanism. It performs:
    - SVD on the real part of the pheromone matrix: `U, S, V = svd(pheromones.real)`.
    - Constructs a low-rank approximation using the top singular values (e.g., top 3 components): `structured = U[:, :k] @ diag(S[:k]) @ V[:k, :]`.
    - It then uses the norm of the panic tensor and the mean magnitude of the anxiety wavefunction to scale this "burst". Essentially, the more panic and anxiety, the stronger the burst.
    - The pheromone matrix might then be perturbed or replaced by some combination of random noise and this structured approximation (in code, it forms a burst pattern and likely adds it to the pheromone field).
    - This method is typically called when panic or anxiety go beyond thresholds, or on schedule. After calling `quantum_burst()`, the solver resets some counts (like it will reset the `burst_countdown` and possibly log the event).
- **Method** `persistent_homology_update()`**:** Performs a persistent homology computation on the current pheromone field. It takes the real part of `pheromones` as input, runs `self.rips.fit_transform()` to get persistence diagrams (which capture topological

features at multiple thresholds), and stores the result in `self.homology_report`. This is mostly for analysis/logging; the more direct use is that `persistent_entropy` might be computed from those diagrams to update beta.

- **Method** `_update_hyperparams()`**:** Adjusts α, β, ρ as described:
  - Calculates `p_mean = mean(panic_tensor)` and `a_amp = mean(|anxiety_wavefunction|)`.
  - Updates `alpha.real = p_mean * a_amp` (complex α gets a new real part from the product of average panic and average anxiety magnitude; the imaginary part of α remains whatever it was, usually 0).
  - Calculates the norm of `covariant_momentum`; updates `rho = 0.9*rho + 0.1*exp(-mom_norm)` (so more momentum -> smaller rho).
  - Grabs the real pheromone field and tries to compute persistent entropy: if successful, sets `beta = 0.1 * entropy`. If the computation fails (e.g., missing dependencies or edge cases), it falls back to decaying beta by 1% (`beta *= 0.99`) to slowly reduce pheromone reliance.
- **Internal utilities:** `_symmetrize_clamp()` which makes the pheromone matrix symmetric and non-negative (averaging it with its transpose, zeroing diagonal, clamping negatives to 0). This is used to enforce a valid pheromone interpretation (for instance, if pheromones represent edge trails in an undirected space, symmetry makes sense; clamping avoids negative pheromone which might be nonsensical in some interpretations). `_trigger_stagnation_reset()` simply calls the pheromone system's partial reset and resets the stagnation_counter.
- **Method** `optimize(self, agents: List[UniversalNode], loss_fn: Callable[[np.ndarray], float]) -> (np.ndarray, np.ndarray, List[float], Any)`**:** This is the main optimization loop. You pass in a list of agent objects and a loss function (which computes the objective value for a given solution matrix or vector). The optimize method returns a tuple of:
  - final pheromone matrix (real part),
  - final pheromone matrix (imaginary part),
  - panic_history (a list of average panic values per iteration),
  - homology_report (persistent homology result from the final iteration, or possibly a sequence of such reports).

  Inside `optimize`, for each iteration `i` from 0 to `max_iter - 1`: - It extracts the current solution representation from the pheromone matrix. Typically this might be the real part or some interpretation of it. In the example, they often use `real_part = asnumpy(pheromones.real)` and pass that to `loss_fn`. - Compute `loss_val = loss_fn(real_part)`. This is the objective value for the current state. - It creates a dummy gradient `grad_approx` (in the generic version, since we might not have an analytic gradient, they just use a matrix of ones scaled by the loss value). This is a heuristic: a rough signal to feed into panic update. - Calls `panic_backpropagate(grad_approx)` to update the panic tensor. - Appends the current average panic (`mean(panic_tensor)`) to `panic_history`. - Checks panic/anxiety thresholds: if `mean(panic) > 0.7` or `||anxiety_wavefunction|| > 1.7` (thresholds chosen in config), it triggers `quantum_burst()` immediately. - Calls `persistent_homology_update()` to analyze the pheromone field's topology at this iteration. - Calls `_update_hyperparams()` to tweak α, β, ρ based on the new panic, anxiety, and homology info. - Updates pheromones based on the current momentum: `pheromones +=`

`alpha.real * covariant_momentum` (so the pheromone matrix is nudged in the direction of the momentum scaled by α's real part – essentially applying some velocity to pheromone changes). - Symmetrize and clamp pheromones (maintain consistency). - Compute current entropy of pheromones and if it's too far from target_entropy, add a small random imaginary noise (`0.01j * normal(...)`) to the pheromone matrix. This reintroduces exploration diversity if things are too orderly or too random. - Compute a **score** for this iteration: e.g., `current_score = 1/(1+loss_val)` (higher means better). If this is the best_score seen so far, update best_score and reset stagnation_counter. If not improved, increment stagnation_counter and if it hits `partial_reset_threshold`, call `_trigger_stagnation_reset()` (partial reset of pheromones). - Countdown to scheduled bursts: decrement `burst_countdown` each iteration, and if it hits 0, trigger a `quantum_burst()` and reset the countdown to the interval. - **Agent interaction:** Update the economy each iteration via `economy.update_market_dynamics()`. Compute a `scarcity` measure (like mentioned, a blend of pheromone mean and economy market value). Then for each agent in `agents` list, call `agent.propose_action(loss_val, scarcity)`. This effectively runs the multi-agent token update as described before: each agent decides its required resources based on panic, attempts to buy, and gets rewarded. This closes the loop between the solver's state and the agent economy: the loss outcome of this iteration influences agent rewards and therefore their token balances for the next iteration. - Loop continues until max_iter or some break condition (the current implementation doesn't include an explicit break on convergence; it runs full iterations or until externally stopped). - Returns the final pheromone matrix (real and imaginary parts) and logs.

Using UMACO9: To use this core solver in practice, you typically: 1. Create an `EconomyConfig` (described below) and instantiate a `UniversalEconomy`. 2. Create a `PheromoneConfig` and instantiate a `NeuroPheromoneSystem` (core version). 3. Prepare a `UMACO9Config` with appropriate dimensions and initial settings (panic_seed etc.). 4. Instantiate `UMACO9` with those config, economy, and pheromone system. 5. Create a list of `UniversalNode` agents (with their NodeConfig). 6. Call `solver.optimize(agents, loss_fn)` where `loss_fn` is a function you provide to evaluate the objective given a representation of the current state (usually derived from pheromone matrix). 7. Read the results (the final pheromone values, etc.) to extract the solution.

We will see a concrete example of these steps in the **Examples** section (e.g., the Rosenbrock function optimization example sets these up). First, let's document the other core classes referenced here: the economy, nodes, and pheromone system.

UniversalEconomy and Agents (Node) Classes

The economy and agent classes are defined in the core module as well (under `umaco/Umaco9.py` in the unified code).

- **Class** `EconomyConfig`**:** a dataclass for economy parameters:
- `n_agents` (int): Number of agents in the system.
- `initial_tokens` (int): Starting token balance for each agent.
- `token_reward_factor` (float): A scaling factor for how many tokens to give per unit of performance. Higher means more generous rewards.
- `min_token_balance` (int): A floor on agent tokens – if an agent's balance goes below this, it will be bumped up to this minimal value (so agents never completely run out).
- `market_volatility` (float): Standard deviation of random fluctuations to apply to market value each update. Introduces randomness in resource pricing.

- `inflation_rate` (float): Rate at which to reduce token value over time (if not in a crisis), to avoid runaway token growth.

- **Class** `UniversalEconomy`**:** manages the token accounts and market dynamics for the agents. Main methods and properties:

- `__init__(config: EconomyConfig)`**:** Initializes token balances for each agent (a dict mapping agent_id -> initial_tokens) and an empty performance history list for each agent. Sets initial `market_value = 1.0` (baseline resource value) and copies volatility & inflation from config.
- `buy_resources(node_id: int, required_power: float, scarcity_factor: float) -> bool`**:** Simulates an agent purchasing resources. The cost is computed as `cost = int(required_power * 100 * scarcity_factor)`. This means the more resources requested, and the higher the scarcity, the more tokens it costs. The method checks if the agent has at least that many tokens:
    - If yes, it deducts the cost from the agent's tokens. It also ensures the agent doesn't go below min_token_balance – if after purchase tokens < min balance, it resets the agent's tokens to min balance (so an agent can effectively spend down only to a point).
    - Returns True if purchase succeeded, False if not enough tokens.
- `reward_performance(node_id: int, performance: float)`**:** Rewards an agent after an iteration. It calculates `reward = int(performance * 100 * token_reward_factor)` and adds that to the agent's token count. Again enforces min_token_balance floor (if somehow the addition made it lower, which normally shouldn't happen since we add tokens, but min_balance is more relevant after purchases). Also appends the performance score to that agent's performance history.
- `update_market_dynamics()`**:** Simulates a market fluctuation at each step. It draws a random `market_change ~ Normal(0, market_volatility)` and updates `market_value *= (1 + market_change)`. It then clamps market_value between 0.2 and 5.0 to avoid extreme crashes or spikes. (So market_value hovers around 1.0 with some noise, but could as low as 0.2 or high as 5.0 in rare extremes). This market_value can indirectly affect scarcity as used in `buy_resources`. (In the extended economy, market dynamics are more complex; here it's just random noise).

- `get_token_distribution() -> Dict[int, float]`**:** Utility to see what fraction of total tokens each agent has. It sums all tokens and returns a dict of agent -> (agent_tokens/total). This can be useful for monitoring if one agent is monopolizing tokens or if distribution is even.

- **Class** `NodeConfig`**:** dataclass for agent (node) configuration:

- `panic_level_init` (float): Initial panic level for an agent. Typically a small value like 0.2 (20%).

- `risk_appetite` (float): A factor (0 to 1 or more) controlling how boldly the agent uses resources. High risk_appetite means the agent will request more resources when under panic. Low means even if panicking it will be conservative.

- **Class** `UniversalNode`**:** represents an agent in the core UMACO solver. Each agent could be thought of as an ant or a particle with its own strategy. Key parts:

- `__init__(node_id: int, economy: UniversalEconomy, config: NodeConfig)`**:** Sets the agent's ID, a reference to the shared economy, and its config. Initializes:
    - `performance_history` (list of float) empty.

    - `panic_level` to `config.panic_level_init` (starting panic).
    - (It could also have other state if extended, but in the base version these are the main ones.)
- `propose_action(current_loss: float, scarcity: float)`**:** This is called each iteration by the solver to let the agent react to the current state. What it does:
    - Computes a performance metric `perf = 1.0 / (1.0 + current_loss)`. So if loss is low (good solution), perf is closer to 1; if loss is high, perf is near 0.
    - Appends this `perf` to its performance_history.
    - If there are at least 3 entries in history, it checks the trend: `trend = recent_perf - perf_from_two_iterations_ago`. If the trend is negative (performance got worse), it increases panic: `panic_level = max(0.05, min(0.95, panic_level - trend))`. In other words, if the agent's performance is declining, it gets more panicked (panic goes up, since trend will be negative minus negative becomes plus). If performance improved, trend is positive, it reduces panic a bit.
    - It then decides how much resources to request. In this base version, `required_power = 0.2 + 0.3 * panic_level * risk_appetite`. So baseline 0.2, plus up to 0.3 more scaled by how panicked and how risk-seeking it is. This `required_power` is an abstract measure (it could correlate with, say, a fraction of GPU or a number of candidate solutions to evaluate, etc., depending on context).
    - Calls `success = economy.buy_resources(node_id, required_power, scarcity)`. If this returns False (not enough tokens to cover cost), the agent interprets that as it couldn't get what it wanted, and it increases its panic: `panic_level *= 1.1` (10% more panic).
    - Regardless of success, it then calls `economy.reward_performance(node_id, perf)` to get rewarded for its performance this iteration. This will increase its tokens if `perf` was significant.
    - (In base version, propose_action doesn't return anything. It just updates internal state and economy. In an extended version, an agent might actually propose a change or action to the solver; but base UMACO doesn't use agent proposals to directly adjust solution, the influence is indirect via tokens and pheromones.)

In the core UMACO loop, the agents influence the search indirectly: by buying resources (which could conceptually allow them to explore more – although in the base code, nothing explicitly stops an agent from exploring if it fails to buy resources, but we can imagine that in a more complex integration, failing to buy might mean the agent cannot make a large change). More concretely, their main effect is through the **token distribution** – this might not immediately affect the pheromone field in the base optimize (since all agents contribute to the same pheromone field uniformly). However, token distribution would matter if the algorithm were extended to have each agent contribute differently or to run agents in parallel where resource availability could throttle some agents.

Even in the base single-threaded loop, the economy adds a **feedback loop**: if the solution quality improves, all agents get rewarded (since performance is global here, though one could tie performance to individual agents if each agent explored a different solution). If stagnation occurs, agents that aren't improving much will just keep minimal tokens and high panic, so once a random event or quantum burst gives them an opportunity, they might spend big tokens (if they have any) or after a token redistribution they will jump.

Note: In more advanced usage (like the LLM fine-tuning scenario), each agent might focus on a different aspect (we'll see "EnhancedCognitiveNode" in a moment) and propose changes; in that scenario, the economy's token and trade mechanisms become very important to decide whose proposals to accept. But for core UMACO, think of the economy as a mechanism to ensure multiple

agents get a chance and to simulate a resource constraint that prevents too many large moves simultaneously.

NeuroPheromoneSystem (Core Pheromone Field)

The **NeuroPheromoneSystem** in the core context is a simpler version of the pheromone field manager (contrasted with an enhanced version for specialized use). It is defined in the core module as:

- **Class** `PheromoneConfig` **:** configuration for the pheromone system:
- `n_dim` (int): dimension of the pheromone matrix (should match UMACO9Config.n_dim).
- `initial_val` (float): the initial value to fill in the pheromone matrix (e.g., 0.3 everywhere to start, as a baseline pheromone concentration).
- `evaporation_rate` (float): how much pheromone evaporates each update (0 to 1 fraction). For example, 0.1 means pheromone values retain 90% of their value each iteration (losing 10%). Higher evaporation leads to faster forgetting of old paths.

- **Class** `NeuroPheromoneSystem` **(core version):** manages a 2D pheromone matrix (complex) and related operations:

- `__init__(config: PheromoneConfig)` **:** Initializes:
  - `self.pheromones` : a Cupy complex64 array of shape (n_dim, n_dim), filled with `initial_val` (every entry = initial_val + 0j).
  - `self.pathway_graph` : a Cupy float32 array of same shape, initialized with small values (e.g., 0.01 for all entries). This can be thought of as an auxiliary matrix tracking the frequency or strength of connections visited.
  - `self.evaporation_rate` : set from config.
  - (The core version does not explicitly track panic or anxiety per agent; it's mostly a global field. The extended version will integrate with anxiety wavefunction.)
- `deposit_pheromones(paths: List[List[int]], performance: float, intensity: float)` **:** In the base code, this method expects agents to provide the paths they took and a performance score. It then:
  - Evaporates the entire pheromone matrix: `pheromones *= (1 - evaporation_rate)` (reducing all values by the evaporation factor).
  - Computes a `deposit_amt = intensity * (performance ** 1.3)` (just a formula to scale deposit by performance nonlinearly).
  - Iterates over each path in `paths` : each path is a sequence of indices (perhaps representing a route or a selection of variables). For each consecutive pair `(a, b)` in a path, it increases `pheromones[a, b] += deposit_amt` and also increases `pathway_graph[a, b] += 0.1 * performance` (so the pathway graph accumulates some smaller weight of performance).
  - This effectively reinforces those transitions in the pheromone field.
  - (If an agent only provides a single point or trivial path, the code guards that if `len(path) < 2` it continues without depositing, since no transition exists.)
- `partial_reset(threshold_percent: float = 30.0)` **:** Implements a partial pheromone reset to fight stagnation:
  - It flattens the absolute values of the pheromone matrix to an array and finds a cutoff value that is the `threshold_percent` -th percentile (e.g., 30th percentile of pheromone magnitudes).

- All pheromone entries below that cutoff (the weakest 30% of trails) are reset to a small baseline (0.01) to "forget" them and allow new exploration on those edges.
- It also halves the corresponding entries in `pathway_graph` (for those weak pathways) to reduce their influence.
- This way, very faint pheromones (likely untraversed or long not used) are wiped, preventing them from cluttering or from never being tried again (since we set them to a small positive baseline rather than truly zero, they're still available to be chosen in future exploration).

The core `NeuroPheromoneSystem` essentially provides a global memory that *all agents share*. It doesn't differentiate contributions per agent – all deposits go into one matrix. The term "neuro" hints that we consider this like a brain's synaptic weights that adapt as experiences accumulate. In the core design, it's fairly straightforward: deposit pheromone on used paths, evaporate globally, occasionally reset weakest ones.

Complex values interpretation: In the base deposit, we treated everything as if real-valued (notice we used `pheromones = cp.ones(..., dtype=cp.complex64)`, but deposit adds a real number to it). The imaginary part in the core isn't explicitly utilized. However, the UMACO9 solver does manipulate the imaginary part by adding noise or momentum as we saw (`pheromones += 0.01j * noise` in certain situations). The idea is that the imaginary part of pheromones can carry a sort of *momentum or oscillation information*, whereas the real part is more like a conventional pheromone concentration. In more advanced usage, one could interpret the imaginary component as "negative pheromone" (repellent) or alternate route markers.

To the developer: when using the core UMACO9, you typically won't interact with NeuroPheromoneSystem methods directly except perhaps to inspect or reset. The `optimize()` loop handles evaporation and deposit calls internally through agent actions or its own logic. But it's important to understand that if you wanted to plug in a custom domain, you might adapt how `deposit_pheromones` works or how a solution is read from `pheromones`.

At this point, we have covered the core classes: **UMACO9 (solver), UniversalEconomy, UniversalNode, and NeuroPheromoneSystem** along with their configs. These constitute the generic framework that is solver-agnostic.

Next, we will discuss the **extended modules** and classes that are provided to show how UMACO can be specialized for particular complex tasks – such as fine-tuning a Large Language Model (LLM). These demonstrate how you can build on the core to incorporate domain-specific logic while still following the UMACO architecture.

LLM Optimization Extension (MACAO in `maco_direct_train16.py`)

One of the powerful features of UMACO is that you can plug in domain-specific behaviors by extending the economy, pheromone system, and nodes. The repository includes a module (in `umaco/maco_direct_train16.py`) which implements **MACO for LLM training** – in the thesis context called "MACO-LLM" or sometimes referred to as **MACAO** (Multi-Agent Cognitive Architecture with Organic pathways). This is essentially UMACO tailored to optimizing the training process of a language model. We will outline the key components of this extension:

Class `MACAOConfig`: This is an extended configuration class that inherits the general structure of EconomyConfig, PheromoneConfig, and adds many domain-specific parameters. It's not a subclass but a new dataclass aggregating everything needed for the LLM scenario. Key fields in `MACAOConfig`: -

General training settings: `model_name` (e.g., a HuggingFace model identifier), `output_dir` for saving results, `training_data_file` path, etc. - **Economy parameters:** `n_agents`, `initial_tokens`, `token_reward_factor`, `min_token_balance`, `market_volatility`, `inflation_rate`. (These mirror EconomyConfig but possibly with different default values tuned for LLM training. There are also fields like `enable_trading` (bool) to turn on/off direct token trading among agents, `trading_fee`, and `trade_offer_lifetime` which are used in the enhanced economy.) - **Pheromone/Environment parameters:** `initial_pheromone` (like initial_val), `evaporation_rate`, `pheromone_intensity` (scale of deposits – something the basic config didn't explicitly have as a separate field). - **Neurochemical factors:** `myrcene_factor`, `limonene_factor`, `pinene_factor`, `linalool_factor` – as discussed, these tune stability, creativity bursts, clarity, and smoothing in the neuropheromone dynamics. - **Dynamic pathway parameters:** `noise_std` (std dev for exploratory noise in pheromone updates), `target_entropy`, `partial_reset_threshold`, `quantum_burst_interval` – analogous to UMACO9Config but possibly different defaults (for example, they might use a larger interval for quantum bursts in LLM training since training is slow). - **LLM training hyperparams:** `batch_size`, `grad_accum_steps`, `learning_rate`, `lora_rank`, `lora_alpha`, `num_epochs`, `max_seq_length` – these relate to how the LLM is fine-tuned (e.g., LoRA parameters for low-rank adaptation, learning rate schedule, etc.). - **Bounds for proposals:** `lr_min`, `lr_max`, `lora_dropout_min`, `lora_dropout_max` – define the range within which agents can propose changing these hyperparameters. - **Visualization and Logging:** flags like `enable_visualization`, `visualization_interval` (how often to record or plot economy state), `export_economy_data`, and WandB (Weights & Biases) integration settings: `wandb_project`, `wandb_run_name`, `log_interval` for how often to log.

This config essentially collects everything needed to coordinate an LLM training run with MACO in the loop.

Class `EnhancedQuantumEconomy`**:** This extends the concept of UniversalEconomy with additional features specifically for the LLM scenario: - It uses `MACAOConfig` for initialization rather than EconomyConfig. - **Token accounting:** It still creates `tokens` dict and `performance_history` for each agent (these may store tuples of more complex performance metrics). - **Market and resource tracking:** It introduces `resource_scarcity` (initial 0.5) to explicitly track resource usage (like GPU memory usage, etc.). It also keeps lists for `outstanding_offers` and `completed_trades` (to implement a token trading market between agents), and an `agent_specializations` dict. - **History logs:** It maintains a `history` dict to record time series of token distribution, market value, resource scarcity, trade volume, and agent performance at each time step (these are used for visualization). - **Performance metrics:** It adds fields like `initial_loss` (to remember the baseline loss when training started), `loss_ewma` (exponential weighted moving avg of loss for stability in reward calc), and `perplexity_history` (since in language model training, perplexity is a key metric). - **Agent specialization assignment:** In `__init__`, it calls `_initialize_specializations()`. This randomly assigns each agent a "primary" and "secondary" specialization from categories such as `"computation"`, `"memory"`, `"optimization"`, `"exploration"`, `"exploitation"`, `"communication"`. Each agent gets a slight efficiency and production factor. These specializations might influence how their actions are evaluated (for example, an agent specialized in "memory" might get certain benefits when memory is a bottleneck). - **Resource state monitoring:** Method `_get_resource_state()` checks actual system resources: e.g., GPU memory usage and utilization (using `torch.cuda` stats) and CPU load. It returns a dict with values 0–1. The economy uses this to decide how scarce resources currently are (for example, if GPU memory is 90% used, then memory is very scarce). - **Market update (override):** `update_market_dynamics(global_step: int)`: This is more complex than the base version: - It updates the resource_state by calling `_get_resource_state()`. - It still applies a random market_change, but adds factors: a

`scarcity_effect = 0.1 * (resource_scarcity - 0.5)` (so if resources are more scarce than neutral 0.5, it nudges market up or down accordingly), and a `velocity_effect = -0.01 * (trade_volume / n_agents)` (so if there have been many trades recently, it slightly lowers the market as if increased liquidity). - Updates `market_value *= (1 + market_change + scarcity_effect + velocity_effect)` and clamps to [0.2, 5.0]. - It checks total token supply: if total tokens in circulation drop below 70% of initial (maybe many tokens got spent without equivalent rewards), it triggers an **emergency stimulus**: calculates how many tokens to add to bring it back to 70%. It then distributes those tokens either evenly or preferentially to agents below average token balance (this prevents a depression where all agents run out of tokens). This is logged as `economy/emergency_stimulus`. - Otherwise (if token supply is healthy), it applies normal inflation by scaling down each agent's tokens by a small factor (ensuring no runaway inflation if agents keep accumulating). - It records the current state by `_record_current_state(step)`, which appends to the history arrays (token distribution, market value, etc. for plotting). - Logs key economy metrics to WandB if enabled (market value, scarcity, token supply, etc.).

- **Resource trading between agents:**
- `buy_resources(node_id: int, required_power: float) -> bool`: Overridden. It factors in agent specialization and actual resource availability:
    - It updates resource_state (e.g., how much GPU is free).
    - Adjusts the required_power by the agent's efficiency (specialization): e.g., if agent's efficiency is >1, they effectively need more power for same action (less efficient), if <1, they get more out of less power.
    - Determines available_power = 1 - GPU usage (free fraction). If the agent is asking for more than what's available, that's inherently a problem.
    - Calculates a base cost: `base_cost = int(adjusted_power * 350)` (they chose a different scale factor here).
    - Computes a `scarcity_multiplier = exp(2 * resource_scarcity) - 0.5` – if resources are scarce (near 1), exp(2*1) ~7.4, so big multiplier; if abundant (0), exp(0) =1, minus 0.5 = 0.5, so costs are lower.
    - `market_multiplier = market_value` (current economy state).
    - Cost = base_cost * scarcity_multiplier * market_multiplier (rounded to int).
    - If the agent's primary specialization matches the resource under pressure (like primary=="memory" and GPU memory usage is high, or primary=="computation" and GPU util is high), it gives a 20% discount on cost (because the agent is specialized in the scarce resource, it's more "trusted" to utilize it effectively).
    - Then if `available_power >= adjusted_power` (enough resource physically free) and the agent has tokens >= cost:
    - It ensures after spending cost, agent won't go below min balance (if it would, it *does not* auto-bump here, instead it prevents them from spending below that – see below trade).
    - If spending would violate min balance, it doesn't deduct but instead calculates how many more tokens needed and creates a **trade offer**: calls `_create_trade_offer(agent, needed, "need_tokens")`. This logs that agent needs X tokens (maybe another agent can supply).
    - If all good, it deducts cost from tokens and logs purchase success.
    - If either resource not available or tokens insufficient:
    - If insufficient tokens but resource was available, it also creates a trade offer for the missing amount.
    - Otherwise logs cannot buy.
    - Returns True if purchase succeeded, False if not (possibly triggering trades).
- `_create_trade_offer(agent_id, amount, offer_type)`: adds an entry to `outstanding_offers` list indicating this agent either needs tokens or offers tokens.

`offer_type` is "need_tokens" or "offer_tokens". It records an exchange_rate (they set 1.0 for need, 1.3 for offering tokens, meaning a seller might want 1.3x payback maybe). It timestamps the offer.

- `process_trades()`: This tries to match "need_tokens" offers with "offer_tokens" offers. It goes through outstanding_offers, pairs up needs with offers from different agents, and transfers tokens accordingly:
    - For each match, it moves the minimum amount from seller to buyer, logs the trade, and reduces the offers' amounts. It increases `total_trades` count.
    - It removes offers that have been fully satisfied or expired (older than `trade_offer_lifetime` steps).
    - This allows agents with excess tokens to effectively lend or give tokens to those who are short, perhaps in exchange for some implicit future favor (in a real economy, exchange_rate would matter, but here it's simplified).

- `reward_performance(node_id, loss_val, influenced_training=False, loss_improved=False, previous_loss=None, agent_type=None, gradient_norm=None) -> (int, float)`: This is an overridden reward function that uses more complex logic:

    - It calls `calculate_performance()` which likely computes a performance score using loss, agent's focus type, gradient norm, etc., and an improvement metric. For example, it might combine raw loss and how much better it is than previous loss, or for a 'learning_rate' agent use some special metric.
    - If `previous_loss` is given and `loss_improved` wasn't explicitly true, it sets `loss_improved = (previous_loss > loss_val)` – basically check if loss decreased.
    - It computes a base_reward = tanh(performance) * 50 * token_reward_factor. (Using tanh means performance is bounded between -1 and 1 output, so reward is bounded and grows sublinearly).
    - It gets agent's specialization production multiplier.
    - It sets `market_multiplier = 0.3 + 0.7 * min(1.5, max(0.5, market_value))` – ensures market factor is between 0.3 and 1.35 roughly (not letting market crashes or booms overly penalize/reward).
    - If the agent's proposal **influenced_training** (meaning the agent's suggestion was actually applied this iteration, e.g., its LR change was used):
    - If and it led to `loss_improved`, they give a 1.5x bonus on top of base reward.
    - If it was used but did not improve loss, they still give reward but with a 0.7 factor (so it gets some reward but less, perhaps to not discourage trying).
    - The reward = base_reward * production_multiplier * market_multiplier * improvement_factor, then cast to int.
    - Add tokens to agent by that reward amount.
    - Log whether it was a full reward or partial.
    - If the agent's proposal was not used (`influenced_training=False`):
    - It gives a small participation reward = 10% of what it would be: `0.1 * base_reward * production * market`.
    - So agents still get a token trickle for being present even if their suggestions weren't applied, to keep them in the game.
    - Add tokens accordingly and log.
    - Enforce min_token_balance.

- Append to `performance_history[node_id]` a tuple (performance, reward_given, improvement_metric).
- Anti-hoarding measure: if an agent's tokens exceed 1.6 * initial_tokens, it considers that excess and automatically creates a trade offer to **sell** 25% of the excess. This encourages redistribution: a very rich agent will offer tokens out, which `process_trades` may give to poorer agents.
- Returns `(reward, performance)`.

• Other methods:

- `trigger_quantum_burst()`: Different from solver's quantum burst, this is an **economic quantum burst**. If resource pressure is extremely high (in the training loop they check if GPU usage > 0.9 and then call this), it does a drastic token redistribution:
- Logs it (and optionally to WandB).
- It computes total_tokens and sets aside 30% to redistribute equally (base_allocation), 50% to distribute proportional to performance scores, 20% for specialization-based distribution.
- It calculates a weighted recent performance for each agent (with more weight on more recent results using a decay factor). Then uses that to allocate the performance_allocation portion – agents who performed better recently get more of that share.
- It gives specialization bonuses: e.g., if memory is scarce and an agent's primary is memory, it gets a boost in specialization score; similarly for computation if GPU util high, etc. Then the specialization_allocation is split by those scores.
- It constructs a new token distribution: each agent gets base_allocation (equal share) + perf_share + spec_share (from those calculations).
- Replaces `self.tokens` with this new distribution (basically a partial reset of the economy leveling the playing field but still rewarding some).
- Lowers market_value slightly towards 1 (simulate stabilization) and clears outstanding_offers (market resets).
- `get_resource_pressure()`: returns the max of current GPU memory, GPU utilization, and CPU utilization. This is used to decide if a quantum burst is needed (threshold > 0.9, as seen in training loop).
- Visualization methods: `visualize_economy()` and `visualize_current_state()` to plot the data collected in history (token distribution over time, market conditions, performance per agent, etc.). These produce Matplotlib charts for analysis and can save to file.

In essence, **EnhancedQuantumEconomy** turns the simple token economy into a full mini-market: - Agents can trade tokens with each other (so a struggling agent can get tokens from a successful agent if that agent decides to sell, which might simulate an agent focusing on exploitation selling resources to one focusing on exploration, etc.). - There's regulation: if tokens concentrate too much or total tokens drop, the system injects or redistributes tokens (like central bank interventions). - The reward system is more nuanced, giving bonuses when an agent's suggestion was actually useful (aligning incentives with actual problem improvement). - It monitors actual hardware usage to adjust costs, which ties the algorithm's behavior to real resource constraints (important in training where you can't exceed GPU memory).

Class `EnhancedCognitiveNode`**:** This extends the UniversalNode concept for the LLM case: - **Initialization:** In addition to `node_id, economy, config`, it defines focus areas: `['learning_rate', 'regularization', 'architecture', 'data_focus']`. It assigns each agent a focus by taking `focus = focus_areas[node_id % len(focus_areas)]`. So with 8 agents,

you'd get 2 agents per focus (if more agents, it cycles through). - The focus determines what kind of proposals the agent will make: - *learning_rate* agent might suggest new learning rates. - *regularization* agent might tweak weight decay or dropout. - *architecture* agent might propose changes to LoRA rank, alpha, or which layers to fine-tune. - *data_focus* agent might suggest data-related changes like batch size, sequence length, or sampling. - It calls `_initialize_strategy()` which returns an initial strategy dict depending on focus. For example: - If focus == 'learning_rate': strategy might start with `{'lr': config.learning_rate, 'lr_decay': 'cosine', 'warmup_steps': 100}` (meaning it will consider proposals around LR schedule). - 'regularization': might store `{'weight_decay': 0.01, 'dropout': 0.1, 'lora_dropout': 0.05}` as baseline. - 'architecture': might store current LoRA config like `{'lora_rank': config.lora_rank, 'lora_alpha': config.lora_alpha, 'target_modules': [list of transformer modules]}` indicating which parts of model to fine-tune. - 'data_focus': might store things like `{'batch_priority': 'default', 'sequence_length': config.max_seq_length, 'augmentation': 0.0}` (these could influence data pipeline). - It sets up additional attributes: - `performance_history` (empty list), - `panic_level = 0.2` initial (similar concept but here not heavily used like in UniversalNode, since economy covers panic through tokens mostly, but the agent might still have a notion of panic if needed). - `last_resource_request = 0.0`, `trade_history = []`, - Randomize some behavioral traits: `risk_appetite`, `cooperation_tendency`, `innovation_drive` – these can influence how aggressively it trades or changes proposals. - For tracking training dynamics: `last_loss = None`, `initial_loss = None` (to see improvement), `last_lr_direction = 0` (was last LR change up or down), `consecutive_improvements = 0`, `consecutive_regressions = 0`.

- `propose_update(current_loss, iteration, previous_loss=None, gradient_norm=None) -> Dict[str, Any]`: This is where the agent suggests a change based on the current training metrics. In this method:
- If `initial_loss` is not set, it sets it to `current_loss` (so each agent remembers the starting loss).
- It calculates `improvement = (previous_loss - current_loss) / previous_loss` if previous_loss provided, and sets `loss_improved = improvement > 0`.
- It updates its counters:
    - If loss_improved: increment `consecutive_improvements`, reset regressions.
    - If not improved: reset improvements, increment `consecutive_regressions`.
- These counters can be used to gauge if the agent's focus area is consistently helping or hurting.
- The proposal strategy likely goes:
    - If focus is 'learning_rate':
    - The agent might consider adjusting the LR. It knows the current global LR (from the scheduler perhaps).
    - It might use `consecutive_improvements` or `regressions` to decide whether to raise or lower LR. E.g., if consecutive improvements > some threshold, maybe decrease LR to fine-tune slower, if regressions stacking up, maybe the LR is too low and we're plateauing, so increase a bit.
    - It could also consider `last_lr_direction` – if it changed LR upward last time and things improved, it might continue that direction, etc. This particular code might implement something like: if loss keeps improving, gently exploit that strategy; if it worsens, reverse course.
    - The output would be a dict like `{'lr': new_value}`.
    - If focus is 'regularization':
    - Maybe check if loss improved – if not, perhaps increase regularization (like up weight decay or dropout) to prevent overfitting, or if accuracy dropped, maybe decrease regularization if underfitting.

    - Could output proposals like `{'weight_decay': X, 'dropout': Y}` adjustments.
    - If focus is 'architecture':
    - Could propose adjusting LoRA rank or which layers to train. For example, if model seems not learning (loss stagnating), propose increasing LoRA rank or enabling more layers to train (bigger capacity).
    - Or if doing well, maybe focus on fewer layers (to speed up)? This is quite domain-specific.
    - Proposal example: `{'lora_rank': 16, 'lora_alpha': 32}` (just as a suggestion).
    - If focus is 'data_focus':
    - Could suggest increasing max_seq_length if model might benefit from longer context, or adjusting how batches are sampled, etc.
    - E.g., if loss is not improving maybe try different data augmentation or focus on harder examples.
    - Could propose `{'batch_priority': 'hard_first'}` or `'sequence_length': smaller (for faster training if hitting memory limits)`.
- The method likely uses the agent's random traits too (innovation_drive might randomize proposals occasionally to try something new).

- The returned dict is the **proposal**. The training loop will then decide if and how to apply these proposals.

- The `EnhancedCognitiveNode.propose_update` method documentation string notes:

    > "Enhanced proposal with: - Consecutive improvement/regression tracking - Loss-aware LR changes - Memory of successful directions"

So indeed it uses those counters to inform how to adjust.

In the training loop (we'll see in the example section), after computing the loss for a batch, they iterate through each `EnhancedCognitiveNode` in the list: - They call `proposal = node.propose_update(loss_val, global_step, previous_loss=loss_ema, gradient_norm=gradient_norm)`. - Then they check the proposal: - If `node.focus == 'learning_rate'` and proposal has an 'lr', they compare it with current LR. If the proposed LR differs by more than 3%, they actually apply it (update the optimizer's learning rate in all parameter groups). They then mark `proposal_used = True` for that agent and log the change. - They might add similar checks for other focus types (the provided example code explicitly handles only learning_rate changes for direct application, likely because adjusting LR on the fly is straightforward; other proposals like dropout might require reconfiguring the model or scheduler and thus are not applied in the middle of a training epoch, or they could be queued). - After that, they call `economy.reward_performance(node_id, loss_val, influenced_training=proposal_used, loss_improved=(loss_ema > loss_val), previous_loss=loss_ema, agent_type=node.focus, gradient_norm=grad_norm)` to give reward. This uses the EnhancedQuantumEconomy's logic, giving a big reward if proposal was used and helped, etc., as we discussed. - They also accumulate some dummy `agent_paths` and `performances` to update pheromones: in the code we saw, they did

```
path = [node_id, (node_id + 4) % 64]
agent_paths.append(path)
performances.append(perf)
```

This is a bit artificial – they map each agent to a "path" on a 64x64 pheromone grid for deposit. It seems they picked 64 as a fixed dimension for pheromone in LLM (maybe an arbitrary latent space size), and they deposit pheromone in a trivial way just to demonstrate the idea (each agent deposits between itself and itself+4 mod 64). This ensures pheromones get updated even though LLM doesn't have an obvious 2D space. It's a way to represent which agent's actions were effective: if performance was high, those agents deposit more pheromone on their index connections. Over time, maybe certain agent indices (e.g., those focusing on LR vs architecture) will have stronger pheromone, indicating that area of focus is fruitful.

Class `NeuroPheromoneSystem` **(Enhanced version):** In the LLM module, they actually define another `NeuroPheromoneSystem` class (shadowing the core one, since they import it from maco_direct_train16). This enhanced NPS manages pheromones with neurochemical factors: - **Initialization:** likely `__init__(config: MACAOConfig, dimensions: int)` rather than PheromoneConfig. It sets up: - `pheromone_tensor`: complex matrix (dimensions x dimensions) filled with `config.initial_pheromone`. - `anxiety_wavefunction`: a complex vector of length `dimensions` (in their code snippet it was cp.zeros(dimensions)) – this might track anxiety per dimension or something. - `panic_tensor`: a float vector of length `dimensions` (not matrix like before, perhaps treating each dimension as a node with panic). - `pathway_graph`: float matrix same shape, filled with 0.01. - `stagnation_counter = 0`, `best_performance = 0.0`, `quantum_burst_countdown = config.quantum_burst_interval`. - `update_anxiety(current_performance: float, target: float = 0.7)`**:** Adjusts the anxiety wavefunction based on how current performance compares to a target (maybe 0.7 of max). - It computes `performance_gap = target - current_performance`. - `anxiety_factor = tanh(performance_gap * 3)` – so if performance is below target, this yields a positive number up to ~tanh(>0) ~ 1 for big gaps, meaning high anxiety; if performance is above target, gap is negative, tanh is negative, maybe indicating relaxed state. - It updates `anxiety_wavefunction.real = 0.8 * old_real + 0.2 * anxiety_factor`. So it smooths anxiety's real part, injecting 20% of the new factor. - If stagnation_counter > 10 (no improvement for a while): it calculates `creativity_potential = tanh(stagnation_counter / 20)`, and updates `anxiety_wavefunction.imag = 0.7 * old_imag + 0.3 * creativity_potential`. So if stagnation builds, the imaginary part (creative anxiety) increases, encouraging those quantum leaps. - Else (not stagnating), it decays `anxiety_wavefunction.imag *= 0.9` (reduce imaginary anxiety when things are moving). - `apply_neurochemical_effects()`**:** This applies the four factors: - `stability_factor = exp(-|anxiety_wavefunction.real| * myrcene_factor)`. This will be a matrix scaling factor (the code multiplies pheromone_tensor.real by this). If anxiety real is high, exp(-high * factor) is small -> it *shrinks the real pheromone values*, thus preventing extreme buildup (stabilizing). - `pheromone_tensor.real *= stability_factor` (elementwise). - `creativity_factor = tanh(|anxiety_wavefunction.imag| * limonene_factor)`. If anxiety imaginary (creative tension) is high, this tanh -> close to 1. If low anxiety, factor small. They check `if mean(creativity_factor) > 0.6`, then: * `pheromone_tensor += generate_quantum_burst() * creativity_factor`. That is, if the system has built up a lot of creative pressure, it triggers a burst in the pheromone field itself. The `_generate_quantum_burst()` in this class creates a random complex perturbation with some SVD-structured component (they use top 1/4 singular vectors to structure it). Then they blend 30% random, 70% structured (see code above). This is added to pheromone_tensor scaled by creativity_factor elementwise – so only dimensions with high imaginary anxiety get a strong burst. - `pathway_clarity = exp(-var(pheromone_tensor, axis=0) * pinene_factor)`. They compute variance of pheromone along one axis (var of each column perhaps) – if variance is high, meaning pheromone values differ a lot across some dimension (some very strong, some very weak trails), then exp(-high*pinene) is small -> they then* `pheromone_tensor *= pathway_clarity` *(likely*

broadcasting across one axis). This effectively reduces pheromone intensity on dimensions with too high variance, attempting to "level" the pheromone distribution a bit. This promotes clarity by not letting one pathway dominate ridiculously – think of pinene as trimming extreme pheromone values to keep a variety. - `smoothing_factor = tanh(|anxiety_wavefunction.imag| * linalool_factor)`*. Based on imaginary anxiety again, it scales the imaginary part of pheromones:* `pheromone_tensor.imag *= smoothing_factor`*. If anxiety (imag) is high, tanh might ~1 so it doesn't change much; if anxiety is moderate, this might reduce the imaginary signals somewhat, possibly preventing overreaction. Essentially linalool calms the "crazy" part of pheromone updates. -* `_generate_quantum_burst()`*: (we saw earlier) generates a complex random matrix (normal dist 0.1) and tries an SVD of current pheromone_tensor to produce a low-rank structured component. Combines them (30% random, 70% structured). This is used in apply_neurochemical_effects. -* `deposit_pheromones(agent_paths, performances)`*: Enhanced deposit that takes multiple agents' contributions at once: - It evaporates the pheromone:* `pheromone_tensor *= (1 - evaporation_rate)`*. - For each (path, perf) in parallel (they use zip), it calculates a deposit amount =* `pheromone_intensity * (perf ** 1.5)`*. - Then similar loop: for each consecutive pair in the path, adds deposit_amount to pheromone_tensor and a bit (0.1*perf) to pathway_graph. - It also caps the pheromone values: if the maximum absolute pheromone > 10, scale the whole matrix down to keep it in range [–10, 10]. This prevents explosion of pheromone values. -* `get_next_actions(current_positions: List[int]) -> List[int]`**:** This seems to be a helper to simulate an agent moving in the pheromone graph. Given current positions (like if each agent is at some index), it calculates probabilities from pheromone strengths and pathway strengths, adds some noise, and chooses a next position for each. This could be used if implementing an actual traversal of a graph by agents. It's not used in the training context, but perhaps was used in e.g. the SAT or TSP context. - `check_stagnation(current_performance: float) -> bool`**:** Checks if performance improved: - If current > best_performance: update best_performance, reset stagnation_counter, return False (not stagnating). - Else increment stagnation_counter. - If stagnation_counter >= partial_reset_threshold: * call `_apply_partial_reset()` (similar idea to core partial_reset but implemented differently here), * reset stagnation_counter, return True (signaling a reset happened). - Else return False. - `_apply_partial_reset()`**:** This one is more complex: - It flattens pheromones, sorts by magnitude, and picks `len(sorted)*0.2` as reset_count (20% smallest values perhaps). It resets those lowest pheromone entries to a random small value around initial_pheromone * (0.5–1.0). So it clears out the weakest 20% of trails. - It then reduces weak pathways in pathway_graph: finds a threshold at 30th percentile, halves all those weaker connections. - It introduces some new random connections in pathway_graph (like adding random edges with moderate weight) to simulate exploring new routes. - So it's like: prune weak info, dampen some of the rest, and sprinkle a bit of novelty. -* `check_quantum_burst() -> bool`**:** Decrements quantum_burst_countdown, and if it hits 0: - Calls `_apply_quantum_burst()` (which logs and adds a burst pattern scaled by limonene as we saw, and also dampens anxiety wavefunction real by factor 0.2 to simulate relief). - Resets countdown to quantum_burst_interval. - Returns True (indicating a burst happened). Otherwise returns False.

So the enhanced NeuroPheromoneSystem works closely with the neurochemical factors and ties into the agent performance feedback. In the LLM training loop, after each batch: - They call `pheromone_sys.deposit_pheromones(agent_paths, performances)` where each agent got a dummy path and a performance metric. This updates the pheromone_tensor. - Then `pheromone_sys.update_anxiety(current_perf)` (here they feed maybe overall performance or one agent's? The code uses 1/(1+loss) as performance in calls). - Then `pheromone_sys.apply_neurochemical_effects()` to adjust pheromones with factors and possibly do a burst if creative anxiety was high. - Then `if pheromone_sys.check_stagnation(current_perf): log partial reset event`. - `if pheromone_sys.check_quantum_burst(): log quantum burst event`. - If

economy.get_resource_pressure() high: `economy.trigger_quantum_burst()` and log (this resets tokens if GPU is maxed for example).

We can see how all pieces come together: the pheromone field tracks something like which agents (or which strategies) have been successful (through agent_paths/performances deposit). This is a proxy since in LLM training we don't have a spatial path; they made the agent index a "dimension" in pheromone space. The persistent homology aspect might be less emphasized here (they didn't explicitly call a homology update in the training loop, likely omitted for speed in that context), but they kept the infrastructure for partial resets and bursts.

QuantumMACO: While not a class, the term "QuantumMACO" could be used to describe this whole integration of quantum economy and cognitive agents for an ML context. In conversation we would just call it the enhanced MACO for LLM.

To avoid confusion: UMACO (universal) is the framework; MACAO (with an extra A) refers to this particular adaptation with "Organic pathways" (the neurochemistry analogy) for LLM. But conceptually, both are the same framework applied to different domains.

Examples and Use Cases

To illustrate how UMACO can be applied to various optimization problems, the repository provides a number of example scripts under the `examples/` directory. These examples cover domains from simple mathematical functions to complex real-world tasks. We will walk through three main examples that demonstrate UMACO's universality: a general function optimization, a SAT solver scenario, and an LLM fine-tuning scenario. We will then briefly mention other provided examples.

Example 1: General Function Optimization (Rosenbrock 2D)

Script: `examples/basic_optimization.py`

This example demonstrates using UMACO to minimize a simple 2D function – the Rosenbrock function. The Rosenbrock is a classic non-convex test function with a global minimum at (1,1) and a tricky curved valley. We show that UMACO can find this minimum.

Summary: The script sets up a 64×64 pheromone matrix (mostly unused dimensions except the first two for x and y), runs UMACO for 500 iterations, and prints the found (x, y).

Steps: 1. **Define the objective function** `rosenbrock(x)`:

```
def rosenbrock(x):
    # f(x,y) = (1 - x)^2 + 100*(y - x^2)^2
    if x.ndim == 2:
        # If x is a pheromone matrix, we take diagonal entries as coordinates
        n = min(x.shape[0], x.shape[1])
        a = np.diag(x)[:n]
        if len(a) < 2:
            return np.sum(a**2)
        x, y = a[0], a[1]
    else:
```

```
        x, y = x[0], x[1]
    return (1 - x)**2 + 100 * (y - x**2)**2
```

This is written to handle both a direct vector or a pheromone matrix input. For a pheromone matrix (64x64), it takes the diagonal entries as candidate solution values (assuming the solution is encoded along the diagonal).

1. **Initialize UMACO components:**

```
economy_config = EconomyConfig(n_agents=8, initial_tokens=250,
token_reward_factor=3.0)
economy = UniversalEconomy(economy_config)

n_dim = 64
pheromone_config = PheromoneConfig(n_dim=n_dim, initial_val=0.3,
evaporation_rate=0.1)
pheromones = NeuroPheromoneSystem(pheromone_config)

initial_values = np.zeros((n_dim, n_dim))
initial_values[0, 0] = -0.5  # initial guess for x
initial_values[1, 1] = -0.5  # initial guess for y

config = UMACO9Config(
    n_dim=n_dim,
    panic_seed=initial_values + 0.1,  # start panic tensor around small
values
    trauma_factor=0.5,
    alpha=0.2, beta=0.1, rho=0.3,
    max_iter=500,
    quantum_burst_interval=50
)
solver = UMACO9(config, economy, pheromones)
agents = [UniversalNode(i, economy, NodeConfig()) for i in range(8)]
```

We choose 8 agents, each starting with 250 tokens. The pheromone matrix is 64×64 with initial value 0.3 everywhere. We set the first two diagonal elements to -0.5 to start near (-0.5,-0.5) in solution space (some initial guess). Alpha, beta, rho are given initial values (these will adapt anyway). We set quantum bursts to happen every 50 iterations at most.

2. **Run the optimization:**

```
pheromone_real, pheromone_imag, panic_history, homology_report =
solver.optimize(agents, rosenbrock)
```

This runs the loop described earlier. Each iteration, `rosenbrock` is called with the current pheromone matrix (the solver passes the numpy version of pheromones.real). The multi-agent economy is updated too.

3. **Results extraction:**

```
final_x = pheromone_real[0, 0]
final_y = pheromone_real[1, 1]
final_value = rosenbrock(np.array([final_x, final_y]))
print(f"Final solution: x = {final_x:.6f}, y = {final_y:.6f}")
print(f"Final function value: {final_value:.6f}")
print("Global minimum (expected): x = 1.0, y = 1.0, f(x,y) = 0.0")
```

   It takes the solution encoded on the diagonal (0,0 and 1,1 positions). Ideally, these will have moved close to 1.0 by end. It also prints the known global min for comparison.

4. **Plotting** (optional): The script then produces several plots:

5. Panic level over iterations (to see if panic spikes and then resets).
6. Heatmap of final pheromone matrix real part.
7. Bar chart of final token distribution among agents (to see which agents ended up with more tokens; possibly indicating which agent contributed most).
8. Contour plot of the Rosenbrock function (log scale) with the UMACO solution and true min marked.

Running this example, you would see that UMACO indeed converges the (x,y) towards (1,1). The panic history might show spikes at times (and drops after quantum bursts or resets). The token distribution might not be uniform – maybe one or two agents got higher token shares, indicating they were more successful. But because we haven't differentiated agents' search (they all effectively work on same solution in this simple setup), token differences might just be due to minor timing differences.

This example highlights that **even for a simple continuous optimization, UMACO can be applied**. You don't need gradient descent; UMACO treats it as a search problem in a high-dimensional "pheromone" space and finds the optimum via its multi-agent strategy.

Example 2: Boolean Satisfiability (SAT) Solver

Script: `examples/ultimate_zvss-v4-n1.py` (and related, possibly the `macov8no-...` scripts)

This example (called "Ultimate ZVSS" in the code) demonstrates UMACO tackling a Boolean SAT problem. Specifically, it sets up a GPU-accelerated SAT solver that uses multiple ant-colony (MACO) strategies internally, plus some integration with an external SAT solver for validation. The acronym ZVSS might refer to some self-optimizing SAT approach (perhaps "Z3 or Minisat with MACO synergy" or an internal project name).

Summary: UMACO in this context is used to guide clause satisfaction search. Agents correspond to multiple ant colonies solving the SAT, pheromones correspond to assignments of boolean variables that satisfy many clauses, etc. The system uses partial solutions and a dynamic adjustment of parameters for each colony.

While the example code is complex, here are key points: - It can generate a random 3-SAT instance or take a CNF file as input. - It uses `MACOZVSSConfig` and `MACOZVSSSystem` classes (found in code around line 1743+ for config, 1922+ for system). These presumably set parameters like number of ants, alpha, beta, etc. for SAT solving specifically, possibly distributing the problem across multiple

pheromone matrices (one per colony or per variable group). - It references an external solver (Minisat) path in the config. After UMACO finds an assignment (or partial assignment), it may call Minisat to verify if it's a full solution. - The algorithm likely uses pheromone to represent how often a variable is set True or False in good partial solutions. Agents try flipping variables (like ants trying different true/false assignments) and deposit pheromone on satisfying assignments of clauses. - The economy in SAT context might allocate agents to focus on different clause subsets or different variable subsets if using multi-colony approach. - The output indicates it prints things like: - "Generated random 3-SAT instance:", - "Starting GPU-MACO SAT Solver...", - Solution analysis and a status (SAT/UNSAT) with runtime.

The **UMACO advantage for SAT**: Instead of brute force or classic DPLL, it performs a stochastic search enhanced by pheromone memory of good variable assignments. The PAQ triad helps escape early dead-ends by quantum bursts (which could mean randomizing a portion of the assignment when stagnated). The economy ensures not all agents try the same assignment; some might specialize in satisfying certain clauses (like specialization could be by clause or variable region, though unclear if implemented explicitly here).

One interesting note: the config mentions a `cryptanalysis parameters` and an example `MACOCryptoOptimizer` as well, but focusing on SAT: The result is that UMACO's SAT solver was able to handle instances that cause traditional solvers to struggle or time out, by continuously adapting its search (e.g., if many clauses remain unsatisfied, panic triggers a burst that completely changes a part of the assignment). It also dynamically adjusts how much it favors previous assignments (beta) based on persistent entropy of the assignment space encountered.

Usage: To use the SAT example, one would run the script specifying either a CNF file or letting it generate one. The MACOZVSSSystem likely has a `main()` that parses arguments:

```
parser = argparse.ArgumentParser(description="MACO Ultimate Edition (ZVSS
Self-Optimization) SAT Solver")
parser.add_argument("--cnf", help="Path to DIMACS CNF file")
parser.add_argument("--timeout", type=int, default=3600)
...
```

It can run with GPU support via Cupy for speed.

This example underscores UMACO's capability in discrete optimization and NP-complete problems. The combination of multiple colonies (agents), pheromone guidance, and quantum resets is akin to running many incomplete SAT solvers in parallel that share information about good variable assignments.

Example 3: LLM Fine-Tuning (MACO-LLM)

Script: `examples/llm_training.py`

This example integrates UMACO (as MACAO) into the fine-tuning process of a language model. It optimizes the training hyperparameters and possibly the training process itself using multi-agent proposals. The scenario described in the thesis case study was fine-tuning a model with limited VRAM (e.g., 7B parameter model on 8GB GPU) and using MACO to dynamically adjust things like learning rate, LoRA settings, etc., to achieve better/per faster fine-tuning.

Summary of what the script does: - It takes a base model (like `TheBloke/CodeLlama-7B-Instruct-GPTQ` in the SquadMACAOConfig example) and sets up a training pipeline (with transformers, datasets, etc.). - It loads a dataset (provided via `--data_file`, e.g., a JSONL of training data). - It uses `MACAOConfig` to configure the training run, possibly loading from a JSON if given or constructing one from CLI args. - It sets up the model for training (using bitsandbytes for 4-bit quantization and PEFT for LoRA adapters because of limited VRAM). - It initializes the **MACO components** for LLM:

```
economy = EnhancedQuantumEconomy(config)
pheromone_sys = NeuroPheromoneSystem(config, dimensions=64)
nodes = [EnhancedCognitiveNode(i, economy, config) for i in
range(config.n_agents)]
```

So if n_agents=8, we get 8 EnhancedCognitiveNodes with focuses [LR, Reg, Arch, Data, LR, Reg, Arch, Data] (repeating). The pheromone system uses 64 dimensions arbitrarily (maybe just a default for internal matrix). - It then enters a standard training loop for `config.num_epochs`. Each step: - A batch is loaded and fed to the model to get loss. - It accumulates gradients for `grad_accum_steps`. - Every few steps (once gradients accumulated): - It computes `gradient_norm` for logging. - Performs optimizer step and scheduler step. - Logs current loss, learning rate, etc. - Every certain interval (log_interval), it invokes the MACO logic: * `economy.update_market_dynamics(global_step)` – update economy with current step context. * `economy.process_trades()` – allow agents to exchange tokens if enabled. * Agent proposals:

```
for i, node in enumerate(nodes):
    proposal = node.propose_update(loss_val, global_step,
previous_loss=loss_ema, gradient_norm=gradient_norm)
    proposal_used = False
    if node.focus == 'learning_rate' and 'lr' in proposal:
        # if LR change is significant
        ... apply new lr ...
        proposal_used = True
    # (could handle other focus types similarly if implemented)
    # Reward the agent
    reward, perf = economy.reward_performance(node.node_id, loss_val,
influenced_training=proposal_used, loss_improved=(loss_ema > loss_val if
loss_ema else False), previous_loss=loss_ema, agent_type=node.focus,
gradient_norm=gradient_norm)
    # Prepare for pheromone deposit
    path = [node.node_id, (node.node_id + 4) % 64]
    agent_paths.append(path); performances.append(perf)
```

* Then update pheromone:

```
pheromone_sys.deposit_pheromones(agent_paths, performances)
pheromone_sys.update_anxiety(1.0 / (1.0 + loss_val))
pheromone_sys.apply_neurochemical_effects()
if pheromone_sys.check_stagnation(1.0 / (1.0 + loss_val)):
    ... log partial reset ...
if pheromone_sys.check_quantum_burst():
```

```
    ... log quantum burst ...
if economy.get_resource_pressure() > 0.9:
    economy.trigger_quantum_burst()
    ... log economy burst ...
```

* If visualization is enabled, maybe every N steps they call `economy.visualize_economy()` or record data.

- They also log training accuracy occasionally and send to WandB if used.

- After training, it would save the fine-tuned model or at least the output_dir is prepared.

How UMACO helps here: During training, typically one might have a fixed learning rate schedule, fixed dropout, etc. Using MACO: - The **learning_rate agents** can adjust the LR on the fly. For example, if loss plateaus, an agent might suggest raising LR slightly (maybe the schedule was too conservative). If loss is oscillating or spiking, an agent might suggest lowering LR more quickly. This dynamic tuning can potentially reduce training time or prevent divergence. - The **regularization agents** might tweak weight decay or dropout to combat overfitting or underfitting signals as training progresses (though applying such changes mid-training is non-trivial, the infrastructure is there). - The **architecture agents** could decide to increase LoRA rank or the number of trainable parameters if it detects under-capacity (if loss is not improving at all, perhaps model needs more freedom), or reduce them to save memory if not utilized. - The **data_focus agents** might shuffle or prioritize certain data when progress stalls (not implemented explicitly above, but conceptually they could). - The **economy** in this context ensures not all agents make extreme changes constantly. If an agent's suggestion consistently improves loss (like a good LR schedule), that agent will earn more tokens and thus continue to have influence (it can "afford" to keep making proposals that are costly if we imagine proposals as resource usage). If another agent's changes are mostly not used or not helpful, they get minimal reward (maybe just participation) and might effectively play a smaller role as it can't "buy" its changes (metaphorically). - The **pheromone field** being updated with agent performance can serve as a long-term memory of which agent focuses (or which strategies) have been beneficial. Although in this example it's indirectly used, one could imagine analyzing the pheromone matrix after training: a high pheromone concentration on certain agent indices might indicate those aspects (like learning rate vs regularization) were key to success. This could guide future runs or just serve diagnostic purposes. - The **PAQ triad** via the neuropheromone system monitors the training process globally: - If training stagnates (loss not improving over many steps), the stagnation_counter triggers partial resets. A partial reset in this context might not reset model weights (that would be counterproductive), but it resets pheromone trails and thereby perhaps resets some internal diversity measure. In practice here, `_apply_partial_reset` in pheromone system will introduce new random connections and reduce over-exploited pheromone trails. This could indirectly cause agents to try more diverse proposals (since the memory of what worked gets partially wiped, agents might branch out again). - If anxiety and panic grow (e.g., loss is very stubborn or starts increasing), `check_quantum_burst` will inject a quantum burst in pheromones and also drastically reduce anxiety. The quantum burst in pheromones might shuffle which agents or parameter combos are considered – effectively a big exploration jump: maybe try a completely different combination of hyperparameters (since pheromone influences agent decisions if we had agents pick moves based on pheromone, though in current code it's not explicitly picking actions from pheromone). - The economy's `trigger_quantum_burst` on high resource pressure might reallocate tokens if, say, one agent had monopolized them but now the situation changed (e.g., maybe one focus was important early training, but later another focus needs more attention – by redistributing tokens, you give other agents a chance to propose major changes).

Outcome: Using MACO-LLM, the training might achieve: - More stable or faster convergence: Because if something starts to go wrong, the system self-corrects (LR too high causing divergence -> an agent lowers it quickly; LR too low causing stagnation -> another agent raises it; etc.). - Possibly better final model: by exploring hyperparam combos, it might avoid bad local minima in hyperparameter space (like a learning rate that gets stuck in suboptimal loss). - The example logs would show the dynamics: - Token distribution shifts (maybe LR agent gaining tokens early, regularization kicking in later). - Market conditions (maybe resource_scarcity rising as GPU fills and then economy injecting tokens etc.). - And the training metrics as usual (loss going down, accuracy going up).

This example truly shows UMACO as **a universal optimizer**: it's optimizing not just the model's weights (SGD does that) but the entire training process itself as an optimization problem (finding the best schedule/policy for hyperparameters to minimize loss).

Other Provided Examples

In addition to the above, the repository includes several other scripts, each showcasing UMACO in a different domain. While we won't deep-dive into each, here is a brief overview:

- `examples/TSP-MACO.py`**:** Solves a Traveling Salesman Problem instance using MACO. It defines a `TSP_MACO` class where cities are nodes and ants (agents) construct routes. Pheromones represent desirability of going from city A to B. The MACO framework helps find a near-optimal route. This is a more classic application akin to Ant Colony Optimization, but benefiting from UMACO's advanced features (dynamic params, etc.). If you run it, it will output an optimal or near-optimal route and cost.

- `examples/ultimate_pf_simulator-v2-n1.py`**:** Here "pf" likely stands for **Protein Folding**. This script uses MACO for optimizing a protein folding simulation or parameters of a force field. The code includes `ProteinFoldingConfig` and `ProteinFoldingOptimizer` classes. It indicates MACO was adapted to navigate the complex energy landscape of protein folding. Agents might correspond to different folding moves, pheromones might mark favorable conformations or sequence of moves. The optimizer could attempt to minimize energy or RMSD to known structure. This shows UMACO handling continuous high-dimensional bioinformatics problems.

- `examples/UmacoFORCTF-v3-no1.py`**:** The name hints at a **CTF (Capture The Flag) competition scenario** or cryptographic challenge. Inside, the class `MACOCryptoOptimizer` is mentioned, with an example targeting a 192-bit block cipher (SPEEDY-7-192). This suggests UMACO was used in a cryptanalysis context, e.g., to optimize a differential or find a key by heuristic search. The agents might represent different key subspace searches, pheromone indicating promising key bits. The code likely integrates with actual cipher encryption/decryption to test candidate keys. Using UMACO's adaptive search could find cryptographic keys or vulnerabilities faster than random search or purely analytical methods, by intelligently exploring the key space.

- `examples/NeuroPheromonebasicv5.py`**:** This seems to be an earlier demonstration focusing on the pheromone system itself, possibly in a neural network or graph context (there's a `SelfOptimizingBrainModel` in it). It simulates a network of neurons divided into regions, and uses MACOOptimizer to adjust weights or pathways. Essentially, it's a sandbox to show how a pheromone-based optimizer could self-organize connectivity in a graph. It may not solve a

single clear-cut optimization problem but demonstrates emergent behavior (hence “basic” in name).

Each of these examples underscores a different facet of UMACO: - TSP and SAT show **combinatorial optimization**. - Protein folding and cryptanalysis show **scientific/engineering optimization** where the search space is huge and rugged. - The “NeuroPheromone basic” example shows **emergent behavior in a complex system**, aligning with the philosophical angle of UMACO (combining computation with ideas from neuroscience and economy).

To run any example, ensure you have the required dependencies (some need Cupy, some need domain-specific libraries or data). The pattern to use them is similar: - Configure problem (via config classes or arguments). - Instantiate UMACO solver or relevant system class. - Call the appropriate method (optimize, run, train, etc.). - Observe the output or logs for results.

Conclusion

UMACO is a **universal, modular optimization framework** that can be bent to almost any problem domain. By abstracting the optimization process into cognitive agents, a shared stigmergic memory, and an economic coordination mechanism, it provides a rich platform for emergent problem-solving strategies. General software engineers can apply UMACO without delving into the low-level mathematics of each domain – instead, one configures the framework’s components (agents, pheromones, economy) and plugs in the domain-specific objective or dataset. The system then *self-organizes* the search for an optimum, dynamically adapting as it learns about the problem.

This developer guide covered: - **Concepts:** PAQ triad for self-adaptation, topological pheromones for memory, token economy for multi-agent coordination, and automatic hyperparameter tuning. - **Core Components:** We described in detail the classes and methods that implement these concepts, like `UMACO9`, `UniversalEconomy`, `UniversalNode`, and `NeuroPheromoneSystem`, including how they interact each iteration. - **Extended Components:** We saw how to extend UMACO for specific needs, exemplified by the MACAO LLM training extension (`EnhancedQuantumEconomy`, `EnhancedCognitiveNode`, etc.) which added features like agent specializations and token trading. - **Examples:** We walked through how to use UMACO in practice on different tasks, from finding minima of functions to solving SAT and tuning language models, without changing the fundamental algorithm – only configuring it to the task at hand.

With UMACO, one can approach a new optimization problem by: 1. Defining how to represent a candidate solution (and how it might map to a pheromone matrix or agent decisions). 2. Writing a loss or fitness function for the problem. 3. Choosing the number of agents and initial tokens (more agents for more complex or larger search spaces perhaps). 4. Setting basic hyperparameters (dimensions of pheromone, etc.) – though many can remain at defaults thanks to self-tuning. 5. Letting the solver run and observing results, adjusting configurations if needed.

The framework’s strength lies in its **generality and adaptability**. Because it has mechanisms to avoid stagnation and balance exploration/exploitation, it often finds good solutions with little manual parameter fiddling. And because agents operate in parallel (conceptually or literally), it can escape local optima better than single-trajectory methods.

As a developer, you can also extend UMACO: - Implement custom agent classes if your agents need to propose domain-specific moves. - Customize the pheromone update rules to better encode your problem’s structure. - Integrate domain heuristics as part of the “heuristic information” vs pheromone

(the beta parameter can weigh the pheromone vs a known heuristic). - Adjust the economy rules if, say, you want agents to have different roles or if resource constraints in your scenario have a specific meaning.

This guide should serve as a foundation to understanding and using UMACO. We encourage experimentation – the emergent behaviors can sometimes be surprising (in a good way), as the system finds creative ways to solve problems by combining its various subsystems. Whether you're optimizing code, tuning machine learning models, solving puzzles, or simulating complex adaptive systems, UMACO provides a flexible toolkit to approach the challenge in a novel multi-agent, cognitive way.
