Hierarchical Multi-Agent Architecture for Real-Time Codeforces Problem Solving with Chess-Inspired Strategic Abstraction

1 Abstract

This document proposes a hierarchical multi-agent system designed to solve Codeforces problems in real time while providing structured explanations inspired by chess strategic archetypes. The system integrates role-based agent hierarchy, session-scoped file memory, rating-based governance, and computational cost control. The objective is to improve reasoning robustness, reduce hallucination, and maintain cost efficiency.

2 Motivation

Competitive programming and chess share structural similarities:

- Adversarial reasoning
- Strategic pattern recognition
- Resource optimization under constraints
- Evaluation of alternative branches

The proposed system models problem solving as an organizational hierarchy analogous to corporate structures and chess decision processes.

3 System Overview

The architecture consists of:

1. Hierarchical agent roles
2. Session-scoped structured memory
3. Rating-based activation control
4. External execution verification
5. Token and computation cost control

4 Hierarchical Agent Roles

The system uses role specialization instead of monolithic reasoning.

4.1 Intern Agent

Responsibilities:

- Extract problem constraints
- Propose candidate paradigms
- Estimate rough time complexity

Output schema:

```
{
  "proposed_paradigm": "...",
  "estimated_complexity": "...",
  "confidence": 0.xx
}
```

4.2 Engineer Agent

Responsibilities:

- Refine candidate strategies
- Eliminate infeasible approaches
- Structure algorithm design

4.3 Senior Engineer Agent

Responsibilities:

- Prove correctness
- Identify corner cases
- Optimize complexity

4.4 Lead Agent

Responsibilities:

- Compare multiple viable strategies
- Select optimal trade-off
- Validate scalability

4.5 CEO Agent

Responsibilities:

- Final approval
- Confidence calibration
- Rating update
- Generate final structured explanation

Authority flows upward. Feedback may flow across roles, but finalization authority remains hierarchical.

5 Session-Scoped File Memory

Each problem operates inside an isolated session workspace:

```
/session_id/
    problem_spec.json
    constraint_analysis.json
    candidate_strategies.json
    decision_log.json
    proof_notes.md
    test_results.json
    final_solution.cpp
```

Key principles:

- Memory is isolated per problem
- Files are structured and version-controlled
- Agents interact only via file updates
- No cross-session contamination

6 Rating System

Each problem has a rating R_p.

Each agent tier has a rating R_a.

Expected success probability:

$$E = \frac{1}{1 + 10^{(R_p - R_a)/400}}$$

Rating update rule:

$$R_{new} = R_{old} + K(S - E)$$

Where:

- $S = 1$ if solved successfully
- $S = 0$ otherwise
- K is update constant

Hierarchy activation depends on problem rating. Lower-rated problems do not activate full stack.

7 Hallucination Mitigation

The system reduces hallucination through:

1. Structured JSON outputs
2. Explicit constraint files
3. External execution validation
4. Limited revision cycles
5. Confidence-based escalation

Execution validation is mandatory. Logical claims are verified through runtime testing.

8 Computation and Cost Control

8.1 Model Tiering

- Lower roles use smaller models
- Higher roles use larger models only when needed

8.2 Token Budgeting

Each role has a strict token limit. Outputs exceeding limit must be summarized.

8.3 Adaptive Escalation

Higher-tier agents activate only if:

- Confidence is low
- Complexity risk detected
- Verification fails

9 Comparative Strategy Evaluation

Multiple strategies may be proposed. Comparison is based on:

- Asymptotic complexity
- Empirical runtime
- Edge case robustness
- Confidence score

Only top-ranked strategy proceeds to final approval.

10 Chess-Inspired Strategic Abstraction Layer

The system optionally maps algorithm paradigms to chess archetypes.

Example mapping:

- Greedy $\rightarrow$ Initiative play
- Dynamic Programming $\rightarrow$ Positional accumulation
- Backtracking $\rightarrow$ Variation calculation
- Game Theory $\rightarrow$ Perfect adversarial play

This layer enhances pedagogical clarity but does not affect correctness verification.

11 Termination Criteria

A session concludes when:

- All test cases pass
- Confidence exceeds threshold
- Maximum revision cycles reached

Session memory is then archived or discarded.

12 Conclusion

The proposed system combines hierarchical reasoning, structured memory, rating governance, and execution validation to build a robust and cost-aware multi-agent solver for Codeforces problems. The design emphasizes convergence, verification, and computational efficiency while enabling optional strategic abstraction inspired by chess.

13 Formal System Model

13.1 Problem Definition

Let a Codeforces problem be defined as:

$$P = (I, O, C, T)$$

Where:

- I = Input specification
- O = Output specification
- C = Constraint set
- T = Tag set (algorithmic hints)

The objective of the system is to construct:

$$S = (A, \Pi, \Phi)$$

Where:

- A = Algorithm
- Π = Proof of correctness
- Φ = Complexity characterization

Such that A satisfies C and produces correct O for all valid I.

13.2 Hierarchical Agent Model

Define a set of agents:

$$\mathcal{H} = \{H_1, H_2, H_3, H_4, H_5\}$$

Where:

- H_1 = Intern
- H_2 = Engineer
- H_3 = Senior Engineer
- H_4 = Lead
- H_5 = CEO

Each agent H_i performs a transformation:

$$H_i : (P, M_i) \to (M_{i+1})$$

Where M_i represents the structured session memory at stage i.
Authority ordering is defined as:

$$H_1 \prec H_2 \prec H_3 \prec H_4 \prec H_5$$

Meaning decisions can only be finalized by higher-order agents.

14 Session Memory State Machine

Let the session state be:

$$\Sigma = (F, D, V)$$

Where:

- F = File state (problem, constraints, strategies)
- D = Decision log
- V = Verification status

State transitions follow:

$$\Sigma_0 \to \Sigma_1 \to \cdots \to \Sigma_k$$

Termination condition:

$$\Sigma_k \text{ is terminal if } V = \text{pass} \land \text{confidence} > \tau$$

Where τ is confidence threshold.

15 Computation Cost Model

Let:

$$C_{total} = \sum_{i=1}^{n} (t_i \cdot \lambda_i)$$

Where:

- t_i = tokens used by agent i
- λ_i = cost per token for model used by agent i

To maintain cost efficiency:

$$t_i \leq T_i^{max}$$

Where T_i^{max} is strict token budget per role.
Escalation occurs only if:

$$\text{confidence}_i < \theta_i$$

Where θ_i is escalation threshold.

16 Adaptive Activation Policy

Let problem rating be R_p.
Define activation function:

$$\alpha(R_p) = \begin{cases} \{H_1, H_2\} & R_p < 1200 \\ \{H_1, H_2, H_3\} & 1200 \leq R_p < 1800 \\ \{H_1, H_2, H_3, H_4\} & 1800 \leq R_p < 2200 \\ \mathcal{H} & R_p \geq 2200 \end{cases}$$

This ensures computational scaling based on difficulty.

17 Comparative Strategy Framework

Let candidate strategies be:

$$\mathcal{S} = \{S_1, S_2, ..., S_k\}$$

Each strategy is evaluated by:

$$E(S_i) = w_1 \cdot \text{complexity} + w_2 \cdot \text{runtime\_empirical} + w_3 \cdot \text{confidence}$$

Final selection:

$$S^* = \arg\min_{S_i \in \mathcal{S}} E(S_i)$$

Subject to correctness constraints.

18 Verification Framework

Verification consists of:

1. Sample test validation
2. Randomized test generation

3. Adversarial edge case synthesis

Let:

$$V = \bigwedge_{j=1}^{m} \text{Test}_j(A) = \text{pass}$$

Only if $V = \text{true}$ may finalization occur.

19 Chess Strategy Mapping Formalization

Define mapping:

$$\mathcal{M} : \mathcal{A} \to \mathcal{C}$$

Where:

- $\mathcal{A}$ = Algorithmic paradigms
- $\mathcal{C}$ = Chess strategic archetypes

Example mappings:

$$\text{Greedy} \to \text{Initiative}$$

$$\text{Dynamic Programming} \to \text{Positional Accumulation}$$

$$\text{Game Theory} \to \text{Perfect Adversarial Play}$$

This mapping is explanatory only and does not influence correctness.

20 Scalability Analysis

Worst-case multi-agent invocation cost:

$$O(k \cdot T_{avg})$$

Where:

- k = number of activated agents
- T_{avg} = average token usage per agent

With adaptive activation:

$$E[C_{total}] \ll C_{max}$$

Because higher-tier agents are invoked infrequently.

21 Failure Modes

- Reinforced internal hallucination
- Over-escalation on trivial problems
- Excessive token growth
- Premature convergence

Mitigation strategies include strict budgets, deterministic checks, and hard termination rules.

22 Future Work

- Reinforcement learning for agent promotion/demotion
- Self-play evaluation against historical Codeforces data
- Empirical benchmarking across rating bands
- Distributed execution framework

23 Experimental Evaluation Plan

Evaluation metrics:

- Solve rate by rating bucket
- Average token usage per rating
- Verification failure rate
- Escalation frequency
- Cost per solved problem

Benchmark datasets:

- Historical Codeforces problem sets
- Random unseen problem batches

Statistical significance measured using solve-rate comparison against single-agent baseline.

24 Orchestration Algorithm

The hierarchical workflow is coordinated by a central orchestrator.

24.1 High-Level Pseudocode

```
function SOLVE_PROBLEM(problem_id):

    P <- FETCH_FROM_CODEFORCES(problem_id)
    INIT_SESSION_WORKSPACE(P)

    active_agents <- ACTIVATE_BY_RATING(P.rating)

    for agent in active_agents:
        M <- LOAD_RELEVANT_FILES(agent)
        response <- agent.PROCESS(P, M)
        UPDATE_WORKSPACE(response)

        if VERIFICATION_FAILED():
            ESCALATE()
        if CONFIDENCE_HIGH() and VERIFICATION_PASSED():
            break

    if VERIFICATION_PASSED():
        FINALIZE_SOLUTION()
        UPDATE_AGENT_RATINGS()
    else:
        MARK_UNRESOLVED()

    RESET_SESSION()
```

24.2 Escalation Rule

Escalation is triggered if:

$$\text{confidence}_i < \theta_i \quad \vee \quad V = \text{fail}$$

Escalation depth is capped at d_{max}.

25 Real-Time Codeforces Integration

25.1 API Endpoints

The system interacts with the Codeforces API:

- `problemset.problems`
- `contest.list`
- `contest.standings`

25.2 Problem Fetch Model

Let:

$$P_{live} = (I, O, C, T, R_p)$$

Where R_p is problem rating.
The system fetches metadata and automatically classifies:

$$\hat{A}_0 = f(T, C)$$

Where f is initial paradigm predictor.

26 Deployment Architecture

26.1 Microservice Structure

The system can be decomposed into:

- Orchestrator Service
- Agent Pool Service
- Execution Sandbox
- Verification Engine
- Rating Manager
- Session Memory Manager

26.2 Service Interaction Flow

1. Orchestrator receives problem request
2. Session initialized
3. Agent invoked
4. Response stored in session memory
5. Code executed in sandbox
6. Verification result returned
7. Decision updated

26.3 Isolation Constraints

Each problem execution occurs in:

- Isolated container
- CPU-time limited sandbox
- Memory bounded environment

27 Complexity Bound on Multi-Agent Overhead

Let:

- k = number of active agents
- r = revision cycles
- T_{max} = max tokens per agent

Worst-case token cost:

$$C_{worst} = k \cdot r \cdot T_{max}$$

Given bounded k and r:

$$C_{worst} = O(1)$$

Relative to number of agents.
Thus, recursion depth does not grow unbounded.

28 Convergence Guarantee

Define a monotonic improvement function:

$$\phi(\Sigma_i) = (\text{confidence}, V)$$

Each escalation must satisfy:

$$\phi(\Sigma_{i+1}) \geq \phi(\Sigma_i)$$

Under partial ordering where:

- Verification failure ¡ Verification success
- Lower confidence ¡ Higher confidence

If no improvement occurs within r_{max} cycles, session terminates.

29 Promotion and Demotion Mechanism

Each agent H_i has rating R_i.

Promotion rule:

$$R_i > R_{threshold}^{i+1} \Rightarrow \text{eligible for promotion}$$

Demotion rule:

$$R_i < R_{threshold}^{i} \Rightarrow \text{restricted scope}$$

This enables adaptive specialization over time.

30 Confidence Calibration Model

Let:

$$\text{confidence} = g(\text{verification pass rate}, \text{complexity margin})$$

Example:

$$\text{confidence} = 0.5 \cdot \text{test\_pass\_ratio} + 0.3 \cdot \text{complexity\_margin} + 0.2 \cdot \text{historical\_accuracy}$$

Confidence threshold τ determines finalization.

31 Security and Integrity Considerations

- Prevent prompt injection via strict input parsing
- Validate API inputs
- Limit execution sandbox permissions
- Prevent cross-session contamination

32 Distributed Scaling Model

For large-scale deployment:

- Horizontal scaling of agent pools
- Queue-based orchestration
- Async verification pipelines
- Token budget monitoring service

Load balancing ensures high-rating problems receive priority compute.

33 Empirical Study Design

To evaluate system robustness:

33.1 Baseline Comparison

Compare against:

- Single-agent LLM baseline
- Deterministic template solver

Metrics:

- Solve rate by rating bucket
- Average cost per problem
- Escalation depth frequency
- Hallucination detection rate

33.2 Ablation Studies

Remove one component at a time:

- Remove hierarchy
- Remove verification
- Remove token cap
- Remove session isolation

Measure degradation.

34 Theoretical Positioning

This system lies at intersection of:

- Multi-agent reasoning systems
- Structured memory architectures
- Cost-aware LLM orchestration
- Educational abstraction frameworks

It represents a hybrid symbolic-neural hierarchical solver model.