```
"""
SAT ALGORITHM DISCOVERY SYSTEM - ENHANCED
Translated from Grimoire Codex to Python

NOW WITH:
- Real DPLL SAT solving algorithm
- Symbolic complexity analysis
- Formal solution verification
- Advanced meta-learning
"""

import random
import time
from dataclasses import dataclass, field
from typing import List, Dict, Tuple, Optional, Callable, Any, Set
from collections import defaultdict
from copy import deepcopy
import math

#
================================================================================
====
# SAT PROBLEM STRUCTURES
#
================================================================================
====

@dataclass
class SATInstance:
    """Represents a 3-SAT problem instance"""
    num_variables: int
    clauses: List[Tuple[int, ...]]

    @staticmethod
    def generate_random(num_vars: int, num_clauses: int, clause_size: int = 3) -> 'SATInstance':
        """Generate random SAT instance"""
        clauses = []
        for _ in range(num_clauses):
            vars_in_clause = random.sample(range(1, num_vars + 1), min(clause_size, num_vars))
            clause = tuple(random.choice([v, -v]) for v in vars_in_clause)
            clauses.append(clause)
        return SATInstance(num_vars, clauses)

    def verify_solution(self, assignment: Dict[int, bool]) -> bool:
```

```
        """Verify if assignment satisfies all clauses"""
        for clause in self.clauses:
            satisfied = False
            for literal in clause:
                var = abs(literal)
                if var in assignment:
                    if (literal > 0 and assignment[var]) or (literal < 0 and not assignment[var]):
                        satisfied = True
                        break
            if not satisfied:
                return False
        return True

#
================================================================================
====
# REAL DPLL SAT SOLVER
#
================================================================================
====

class DPLLSolver:
    """Real DPLL SAT solver with configurable heuristics"""

    def __init__(self, heuristics: List[str], strategies: List[str], parameters: Dict):
        self.heuristics = heuristics
        self.strategies = strategies
        self.parameters = parameters
        self.operation_count = 0
        self.decision_count = 0
        self.propagation_count = 0

    def solve(self, instance: SATInstance, max_operations: int = 10000) -> Tuple[bool,
Optional[Dict[int, bool]], int]:
        """Solve SAT instance using DPLL algorithm"""
        self.operation_count = 0
        self.decision_count = 0
        self.propagation_count = 0

        clauses = [set(clause) for clause in instance.clauses]
        assignment = {}

        result = self._dpll(clauses, assignment, instance.num_variables, max_operations)
```

```
    if result:
        return True, assignment, self.operation_count
    else:
        return False, None, self.operation_count

def _dpll(self, clauses: List[Set[int]], assignment: Dict[int, bool],
          num_vars: int, max_ops: int) -> bool:
    """Core DPLL recursive algorithm"""
    self.operation_count += 1

    if self.operation_count > max_ops:
        return False

    if not clauses:
        return True

    if any(len(clause) == 0 for clause in clauses):
        return False

    # Unit propagation
    if "unit_propagation" in self.strategies:
        while True:
            unit_clause = self._find_unit_clause(clauses)
            if unit_clause is None:
                break

            literal = next(iter(unit_clause))
            var = abs(literal)
            value = literal > 0

            assignment[var] = value
            clauses = self._propagate(clauses, literal)
            self.propagation_count += 1
            self.operation_count += 1

            if not clauses:
                return True
            if any(len(clause) == 0 for clause in clauses):
                return False

    # Pure literal elimination
    if "pure_literal_elimination" in self.strategies:
        pure_literal = self._find_pure_literal(clauses)
        if pure_literal is not None:
```

```
            var = abs(pure_literal)
            value = pure_literal > 0
            assignment[var] = value
            clauses = self._propagate(clauses, pure_literal)
            self.operation_count += 1
            return self._dpll(clauses, assignment, num_vars, max_ops)

        # Choose variable
        var = self._choose_variable(clauses, assignment, num_vars)
        if var is None:
            return True

        self.decision_count += 1

        # Try positive
        assignment_copy = assignment.copy()
        clauses_copy = deepcopy(clauses)
        assignment_copy[var] = True
        clauses_try = self._propagate(clauses_copy, var)

        if self._dpll(clauses_try, assignment_copy, num_vars, max_ops):
            assignment.update(assignment_copy)
            return True

        # Try negative
        assignment[var] = False
        clauses = self._propagate(clauses, -var)

        return self._dpll(clauses, assignment, num_vars, max_ops)

    def _find_unit_clause(self, clauses: List[Set[int]]) -> Optional[Set[int]]:
        for clause in clauses:
            if len(clause) == 1:
                return clause
        return None

    def _find_pure_literal(self, clauses: List[Set[int]]) -> Optional[int]:
        literal_count = defaultdict(int)
        for clause in clauses:
            for literal in clause:
                literal_count[literal] += 1

        for literal in literal_count:
            if -literal not in literal_count:
```

```
            return literal
    return None

def _propagate(self, clauses: List[Set[int]], literal: int) -> List[Set[int]]:
    new_clauses = []
    for clause in clauses:
        if literal in clause:
            continue
        elif -literal in clause:
            new_clause = clause - {-literal}
            new_clauses.append(new_clause)
        else:
            new_clauses.append(clause.copy())
    return new_clauses

def _choose_variable(self, clauses: List[Set[int]],
                     assignment: Dict[int, bool], num_vars: int) -> Optional[int]:
    unassigned = set(range(1, num_vars + 1)) - set(assignment.keys())

    if not unassigned:
        return None

    if not self.heuristics:
        return random.choice(list(unassigned))

    heuristic = self.heuristics[0]

    if heuristic == "random_variable_selection":
        return random.choice(list(unassigned))

    elif heuristic == "most_constrained_first":
        var_count = defaultdict(int)
        for clause in clauses:
            for literal in clause:
                var = abs(literal)
                if var in unassigned:
                    var_count[var] += 1
        if var_count:
            return max(var_count.items(), key=lambda x: x[1])[0]
        return random.choice(list(unassigned))

    elif heuristic == "activity_based":
        var_activity = defaultdict(float)
        for clause in clauses:
```

```
            clause_vars = [abs(lit) for lit in clause if abs(lit) in unassigned]
            activity_boost = 1.0 / max(len(clause), 1)
            for var in clause_vars:
                var_activity[var] += activity_boost
        if var_activity:
            return max(var_activity.items(), key=lambda x: x[1])[0]
        return random.choice(list(unassigned))

    else:
        return random.choice(list(unassigned))

#
================================================================================
====
# COMPLEXITY ANALYZER
#
================================================================================
====

class ComplexityAnalyzer:
    """Mathara + Fractala - Analyze algorithmic complexity"""

    def __init__(self):
        self.measurements = []

    def analyze_run(self, num_variables: int, operations: int, solved: bool) -> Dict:
        if num_variables == 0:
            return {'empirical_complexity': 'N/A'}

        n = num_variables

        linear_ratio = operations / n if n > 0 else 0
        nlogn_ratio = operations / (n * math.log2(n)) if n > 1 else 0
        quadratic_ratio = operations / (n * n) if n > 0 else 0
        exponential_base = operations ** (1/n) if n > 0 and operations > 0 else 0

        analysis = {
            'operations': operations,
            'variables': num_variables,
            'linear_ratio': linear_ratio,
            'exponential_base': exponential_base,
            'solved': solved
        }
```

```
        self.measurements.append(analysis)
        return analysis

    def estimate_complexity_class(self, recent_runs: int = 10) -> str:
        if len(self.measurements) < 3:
            return "INSUFFICIENT_DATA"

        recent = self.measurements[-recent_runs:]

        exp_bases = [m['exponential_base'] for m in recent if m['exponential_base'] > 0]
        if exp_bases:
            avg_base = sum(exp_bases) / len(exp_bases)
            if avg_base > 1.1:
                return f"O(~{avg_base:.2f}^n) EXPONENTIAL"

        linear_ratios = [m['linear_ratio'] for m in recent]
        if linear_ratios:
            avg_linear = sum(linear_ratios) / len(linear_ratios)
            return f"O(n) ratio: {avg_linear:.1f}"

        return "UNKNOWN"

#
================================================================================
====
# MEMORY & STATE
#
================================================================================
====

class MemorySubstrate:
    def __init__(self):
        self.state_history = []
        self.journey_log = []
        self.complexity_history = []

    def track_journey(self, event: str):
        self.journey_log.append({'event': event, 'time': time.time()})

    def record_complexity(self, analysis: Dict):
        self.complexity_history.append(analysis)
```

```
#
================================================================================
====
# ALGORITHM TEMPLATE
#
================================================================================
====

@dataclass
class AlgorithmTemplate:
    name: str
    heuristics: List[str]
    strategies: List[str]
    parameters: Dict[str, Any]
    fitness: float = 0.0
    success_count: int = 0
    total_runs: int = 0
    total_operations: int = 0

#
================================================================================
====
# GENERATION ENGINE
#
================================================================================
====

class GenerationEngine:
    def __init__(self, memory: MemorySubstrate):
        self.memory = memory
        self.templates_created = 0

        self.heuristic_pool = [
            "random_variable_selection",
            "most_constrained_first",
            "activity_based"
        ]

        self.strategy_pool = [
            "unit_propagation",
            "pure_literal_elimination"
        ]

    def generate_algorithm(self) -> AlgorithmTemplate:
```

```
        algorithm = AlgorithmTemplate(
            name=f"Algo_{self.templates_created}",
            heuristics=random.sample(self.heuristic_pool, random.randint(1, 2)),
            strategies=random.sample(self.strategy_pool, random.randint(1, 2)),
            parameters={'max_operations': random.randint(2000, 5000)}
        )
        self.templates_created += 1
        return algorithm

    def mutate_algorithm(self, base: AlgorithmTemplate) -> AlgorithmTemplate:
        mutated = AlgorithmTemplate(
            name=f"{base.name}_m{random.randint(0,99)}",
            heuristics=base.heuristics.copy(),
            strategies=base.strategies.copy(),
            parameters=base.parameters.copy()
        )

        mutation_type = random.choice(['heuristic', 'strategy', 'parameter'])

        if mutation_type == 'heuristic' and mutated.heuristics:
            mutated.heuristics[0] = random.choice(self.heuristic_pool)
        elif mutation_type == 'strategy' and mutated.strategies:
            mutated.strategies[0] = random.choice(self.strategy_pool)
        else:
            mutated.parameters['max_operations'] += random.randint(-500, 500)
            mutated.parameters['max_operations'] = max(1000,
mutated.parameters['max_operations'])

        return mutated

#
================================================================================
====
# EXECUTION ENGINE
#
================================================================================
====

class ExecutionEngine:
    def __init__(self, memory: MemorySubstrate):
        self.memory = memory
        self.complexity_analyzer = ComplexityAnalyzer()

    def execute_algorithm(self, algorithm: AlgorithmTemplate, instance: SATInstance) -> Dict:
```

```
        solver = DPLLSolver(algorithm.heuristics, algorithm.strategies, algorithm.parameters)

        start_time = time.time()
        solved, assignment, operations = solver.solve(instance,
algorithm.parameters['max_operations'])
        execution_time = time.time() - start_time

        verified = False
        if solved and assignment:
            verified = instance.verify_solution(assignment)

        complexity_analysis = self.complexity_analyzer.analyze_run(instance.num_variables,
operations, solved)
        self.memory.record_complexity(complexity_analysis)

        symbol = '✓' if verified else ('?' if solved else '✗')
        self.memory.track_journey(f"{algorithm.name}: {symbol} {operations} ops")

        return {
            'algorithm': algorithm.name,
            'solved': solved,
            'verified': verified,
            'operations': operations,
            'time': execution_time,
            'instance_size': instance.num_variables
        }

#
================================================================================
====
# META-LEARNING
#
================================================================================
====

class MetaLearningEngine:
    def __init__(self, memory: MemorySubstrate):
        self.memory = memory
        self.generation = 0
        self.fitness_history = []

    def evaluate_population(self, algorithms: List[AlgorithmTemplate], results: List[Dict]) ->
List[AlgorithmTemplate]:
        algo_results = defaultdict(list)
```

```
        for result in results:
            algo_results[result['algorithm']].append(result)

        for algo in algorithms:
            algo_results_list = algo_results[algo.name]
            if algo_results_list:
                verified_count = sum(1 for r in algo_results_list if r['verified'])
                total_ops = sum(r['operations'] for r in algo_results_list)
                avg_ops = total_ops / len(algo_results_list)

                verification_rate = verified_count / len(algo_results_list)
                ops_penalty = avg_ops / 10000

                algo.fitness = verification_rate - ops_penalty
                algo.success_count = verified_count
                algo.total_runs = len(algo_results_list)
                algo.total_operations = total_ops
            else:
                algo.fitness = 0.0

        return sorted(algorithms, key=lambda a: a.fitness, reverse=True)

    def evolve_generation(self, algorithms: List[AlgorithmTemplate], generator:
GenerationEngine) -> List[AlgorithmTemplate]:
        self.generation += 1

        next_gen = algorithms[:3].copy()

        while len(next_gen) < len(algorithms):
            parent = random.choice(algorithms[:3])
            child = generator.mutate_algorithm(parent)
            next_gen.append(child)

        avg_fitness = sum(a.fitness for a in algorithms) / len(algorithms)
        self.fitness_history.append(avg_fitness)

        return next_gen

#
==========================================================================
====
# ORCHESTRATOR
```

```
#
================================================================================
====

class SystemOrchestrator:
    def __init__(self):
        self.memory = MemorySubstrate()
        self.generator = GenerationEngine(self.memory)
        self.executor = ExecutionEngine(self.memory)
        self.meta_learner = MetaLearningEngine(self.memory)

        self.population_size = 8
        self.instances_per_gen = 5
        self.current_population = []

    def initialize(self):
        print("🌟 ENHANCED SAT ALGORITHM DISCOVERY SYSTEM")
        print("   Real DPLL + Formal Verification + Complexity Analysis")
        print("=" * 70)

        self.current_population = [self.generator.generate_algorithm() for _ in
range(self.population_size)]
        print(f"✓ Generated {self.population_size} algorithms\n")

    def run_generation(self, gen_num: int):
        print(f"{'='*70}")
        print(f"GENERATION {gen_num}")
        print(f"{'='*70}")

        test_instances = [SATInstance.generate_random(random.randint(10, 15),
random.randint(25, 40))
                          for _ in range(self.instances_per_gen)]

        all_results = []
        for algo in self.current_population:
            for instance in test_instances:
                result = self.executor.execute_algorithm(algo, instance)
                all_results.append(result)

        total = len(all_results)
        verified = sum(1 for r in all_results if r['verified'])
        avg_ops = sum(r['operations'] for r in all_results) / total

        print(f"\n📊 RESULTS:")
```

```
        print(f"  Verified: {verified}/{total} ({verified/total:.1%})")
        print(f"  Avg Operations: {avg_ops:.0f}")

        self.current_population = self.meta_learner.evaluate_population(self.current_population,
all_results)

        best = self.current_population[0]
        print(f"\n🏆 BEST: {best.name}")
        print(f"  Fitness: {best.fitness:.4f}")
        print(f"  Verified: {best.success_count}/{best.total_runs}")
        print(f"  Heuristics: {best.heuristics}")
        print(f"  Strategies: {best.strategies}")

        complexity = self.executor.complexity_analyzer.estimate_complexity_class()
        print(f"  Complexity: {complexity}")

        self.current_population = self.meta_learner.evolve_generation(self.current_population,
self.generator)

    def run_experiment(self, num_generations: int = 8):
        self.initialize()

        for gen in range(1, num_generations + 1):
            self.run_generation(gen)

        print(f"\n{'='*70}")
        print("EXPERIMENT COMPLETE")
        print(f"{'='*70}")

        print(f"\n📈 Fitness Evolution:")
        for i, fitness in enumerate(self.meta_learner.fitness_history):
            print(f"  Gen {i+1}: {fitness:.4f}")

        best = self.current_population[0]
        print(f"\n🏆 FINAL BEST ALGORITHM:")
        print(f"  {best.name}")
        print(f"  Fitness: {best.fitness:.4f}")
        print(f"  Success: {best.success_count}/{best.total_runs} verified")
        print(f"  Heuristics: {best.heuristics}")
        print(f"  Strategies: {best.strategies}")

#
================================================================================
====
```

```
# MAIN
#
================================================================================
====

if __name__ == "__main__":
    system = SystemOrchestrator()
    system.run_experiment(num_generations=8)

    print("\n" + "="*70)
    print("✨ ENHANCED FEATURES:")
    print("  ✓ Real DPLL SAT solver")
    print("  ✓ Formal solution verification")
    print("  ✓ Symbolic operation counting")
    print("  ✓ Complexity estimation")
    print("  ✓ Evolutionary meta-learning")
    print("="*70)

🌟 ENHANCED SAT ALGORITHM DISCOVERY SYSTEM
   Real DPLL + Formal Verification + Complexity Analysis
================================================================================
✓ Generated 8 algorithms

================================================================================
GENERATION 1
================================================================================

📊 RESULTS:
  Verified: 40/40 (100.0%)
  Avg Operations: 23

🏆 BEST: Algo_1
  Fitness: 0.9984
  Verified: 5/5
  Heuristics: ['activity_based', 'most_constrained_first']
  Strategies: ['unit_propagation']
  Complexity: O(~1.28^n) EXPONENTIAL
================================================================================
GENERATION 2
================================================================================

📊 RESULTS:
  Verified: 40/40 (100.0%)
  Avg Operations: 24
```

🏆 BEST: Algo_1
  Fitness: 0.9984
  Verified: 5/5
  Heuristics: ['activity_based', 'most_constrained_first']
  Strategies: ['unit_propagation']
  Complexity: O(~1.28^n) EXPONENTIAL
==
GENERATION 3
==

📊 RESULTS:
  Verified: 40/40 (100.0%)
  Avg Operations: 24

🏆 BEST: Algo_1_m25_m85
  Fitness: 0.9980
  Verified: 5/5
  Heuristics: ['random_variable_selection', 'most_constrained_first']
  Strategies: ['unit_propagation']
  Complexity: O(~1.29^n) EXPONENTIAL
==
GENERATION 4
==

📊 RESULTS:
  Verified: 32/40 (80.0%)
  Avg Operations: 40

🏆 BEST: Algo_1
  Fitness: 0.7984
  Verified: 4/5
  Heuristics: ['activity_based', 'most_constrained_first']
  Strategies: ['unit_propagation']
  Complexity: O(~1.28^n) EXPONENTIAL
==
GENERATION 5
==

📊 RESULTS:
  Verified: 40/40 (100.0%)
  Avg Operations: 17

🏆 BEST: Algo_1

Fitness: 0.9984
Verified: 5/5
Heuristics: ['activity_based', 'most_constrained_first']
Strategies: ['unit_propagation']
Complexity: O(~1.30^n) EXPONENTIAL
==========
GENERATION 6
==========

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 16

🏆 BEST: Algo_1
Fitness: 0.9986
Verified: 5/5
Heuristics: ['activity_based', 'most_constrained_first']
Strategies: ['unit_propagation']
Complexity: O(~1.24^n) EXPONENTIAL
==========
GENERATION 7
==========

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 17

🏆 BEST: Algo_1
Fitness: 0.9985
Verified: 5/5
Heuristics: ['activity_based', 'most_constrained_first']
Strategies: ['unit_propagation']
Complexity: O(~1.23^n) EXPONENTIAL
==========
GENERATION 8
==========

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 13

🏆 BEST: Algo_1
Fitness: 0.9987
Verified: 5/5

Heuristics: ['activity_based', 'most_constrained_first']
Strategies: ['unit_propagation']
Complexity: O(~1.23^n) EXPONENTIAL

==
EXPERIMENT COMPLETE
==

📈 Fitness Evolution:
Gen 1: 0.9977
Gen 2: 0.9976
Gen 3: 0.9976
Gen 4: 0.7960
Gen 5: 0.9983
Gen 6: 0.9984
Gen 7: 0.9983
Gen 8: 0.9987

🏆 FINAL BEST ALGORITHM:
Algo_1
Fitness: 0.9987
Success: 5/5 verified
Heuristics: ['activity_based', 'most_constrained_first']
Strategies: ['unit_propagation']

==
✨ ENHANCED FEATURES:
✓ Real DPLL SAT solver
✓ Formal solution verification
✓ Symbolic operation counting
✓ Complexity estimation
✓ Evolutionary meta-learning
==
>>>