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

Based on OMNIMOIRE architecture, simplified to 7 core layers.
Designed to discover and evolve SAT solving algorithms.
"""

import random
import time
from dataclasses import dataclass, field
from typing import List, Dict, Tuple, Optional, Callable, Any
from collections import defaultdict
import json

#
============================================================================
====
# SPELL DEFINITIONS (Core Computational Primitives)
#
============================================================================
====

@dataclass
class Spell:
    """Base spell - represents a computational operation"""
    name: str
    function: Callable
    description: str

    def cast(self, *args, **kwargs):
        """Execute the spell's function"""
        return self.function(*args, **kwargs)

#
============================================================================
====
# LAYER 0: FOUNDATION - Memory & Identity
#
============================================================================
====

class MemorySubstrate:
    """Preserva + Odyssea - State preservation and tracking"""
```

```python
    def __init__(self):
        self.state_history = []
        self.current_state = {}
        self.journey_log = []

    def checkpoint(self, state: Dict):
        """Preserva: Save current state"""
        self.state_history.append(state.copy())
        self.current_state = state.copy()
        return state

    def track_journey(self, event: str):
        """Odyssea: Track long-running process"""
        self.journey_log.append({
            'timestamp': time.time(),
            'event': event,
            'state_id': len(self.state_history)
        })

    def get_history(self):
        """Retrieve full history"""
        return self.state_history

#
============================================================================
====
# LAYER 1: ALGORITHM GENERATION ENGINE
#
============================================================================
====

@dataclass
class AlgorithmTemplate:
    """Represents a SAT solving algorithm structure"""
    name: str
    heuristics: List[str]
    strategies: List[str]
    parameters: Dict[str, Any]
    fitness: float = 0.0

class GenerationEngine:
    """Musara + Dreamara + Alchemara - Creative algorithm generation"""

    def __init__(self, memory: MemorySubstrate):
```

```python
        self.memory = memory
        self.templates_created = 0

        # Available algorithmic components
        self.heuristic_pool = [
            "random_variable_selection",
            "most_constrained_first",
            "least_constrained_first",
            "degree_heuristic",
            "activity_based"
        ]

        self.strategy_pool = [
            "pure_backtracking",
            "backtracking_with_learning",
            "unit_propagation",
            "pure_literal_elimination",
            "clause_learning"
        ]

    def generate_algorithm(self) -> AlgorithmTemplate:
        """Musara: Generate new algorithm through creative inspiration"""
        num_heuristics = random.randint(1, 3)
        num_strategies = random.randint(1, 3)

        algorithm = AlgorithmTemplate(
            name=f"Algorithm_{self.templates_created}",
            heuristics=random.sample(self.heuristic_pool, num_heuristics),
            strategies=random.sample(self.strategy_pool, num_strategies),
            parameters={
                'restart_threshold': random.randint(10, 100),
                'learning_rate': random.uniform(0.1, 0.9),
                'decay_factor': random.uniform(0.8, 0.99)
            }
        )

        self.templates_created += 1
        self.memory.track_journey(f"Generated {algorithm.name}")
        return algorithm

    def mutate_algorithm(self, base: AlgorithmTemplate) -> AlgorithmTemplate:
        """Alchemara: Transform existing algorithm"""
        mutated = AlgorithmTemplate(
            name=f"{base.name}_mutated_{random.randint(0,999)}",
```

```
            heuristics=base.heuristics.copy(),
            strategies=base.strategies.copy(),
            parameters=base.parameters.copy()
        )

        # Randomly mutate one component
        mutation_type = random.choice(['heuristic', 'strategy', 'parameter'])

        if mutation_type == 'heuristic' and self.heuristic_pool:
            if random.random() < 0.5 and len(mutated.heuristics) < len(self.heuristic_pool):
                # Add new heuristic
                available = [h for h in self.heuristic_pool if h not in mutated.heuristics]
                if available:
                    mutated.heuristics.append(random.choice(available))
            elif mutated.heuristics:
                # Replace existing
                idx = random.randint(0, len(mutated.heuristics) - 1)
                mutated.heuristics[idx] = random.choice(self.heuristic_pool)

        elif mutation_type == 'strategy' and self.strategy_pool:
            if random.random() < 0.5 and len(mutated.strategies) < len(self.strategy_pool):
                available = [s for s in self.strategy_pool if s not in mutated.strategies]
                if available:
                    mutated.strategies.append(random.choice(available))
            elif mutated.strategies:
                idx = random.randint(0, len(mutated.strategies) - 1)
                mutated.strategies[idx] = random.choice(self.strategy_pool)

        else:  # parameter
            param = random.choice(list(mutated.parameters.keys()))
            if isinstance(mutated.parameters[param], int):
                mutated.parameters[param] += random.randint(-10, 10)
            else:
                mutated.parameters[param] *= random.uniform(0.8, 1.2)

        self.memory.track_journey(f"Mutated {base.name} → {mutated.name}")
        return mutated

#
============================================================================
====
# LAYER 2: EXECUTION & TESTING ENGINE
```

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

@dataclass
class SATInstance:
    """Represents a 3-SAT problem instance"""
    num_variables: int
    clauses: List[Tuple[int, int, int]]  # Each clause is 3 literals

    @staticmethod
    def generate_random(num_vars: int, num_clauses: int) -> 'SATInstance':
        """Generate random 3-SAT instance"""
        clauses = []
        for _ in range(num_clauses):
            # Each literal is a variable (1 to num_vars) with random sign
            clause = tuple(
                random.choice([i, -i])
                for i in random.sample(range(1, num_vars + 1), 3)
            )
            clauses.append(clause)
        return SATInstance(num_vars, clauses)

class ExecutionEngine:
    """Solva + Titanis - Execute and measure algorithms"""

    def __init__(self, memory: MemorySubstrate):
        self.memory = memory
        self.execution_count = 0

    def execute_algorithm(self, algorithm: AlgorithmTemplate,
                          instance: SATInstance,
                          max_steps: int = 1000) -> Dict:
        """Execute algorithm on SAT instance and measure performance"""
        start_time = time.time()

        # Simplified SAT solver simulation
        # In real implementation, this would use the algorithm's actual heuristics
        steps = 0
        assignment = {}
        solved = False  # Initialize solved flag

        # Simulate solving with random walk (placeholder for real algorithm)
        for step in range(max_steps):
```

```
            steps += 1

            # Check if we should give up based on algorithm parameters
            if steps > algorithm.parameters.get('restart_threshold', 50):
                break

            # Simulate some progress
            if random.random() < 0.01:  # Small chance of "solving"
                solved = True
                break

        execution_time = time.time() - start_time

        result = {
            'algorithm': algorithm.name,
            'solved': solved,
            'steps': steps,
            'time': execution_time,
            'instance_size': instance.num_variables
        }

        self.execution_count += 1
        self.memory.track_journey(
            f"Executed {algorithm.name}: {'✓' if solved else '✗'} in {steps} steps"
        )

        return result

#
============================================================================
====
# LAYER 3: PATTERN ANALYSIS ENGINE
#
============================================================================
====

class AnalysisEngine:
    """Insighta + Clarivis + Fractala - Pattern recognition"""

    def __init__(self, memory: MemorySubstrate):
        self.memory = memory
        self.patterns = defaultdict(list)

    def analyze_results(self, results: List[Dict]) -> Dict[str, Any]:
```

```
        """Analyze execution results to find patterns"""
        if not results:
            return {}

        # Calculate success rate
        total = len(results)
        solved = sum(1 for r in results if r['solved'])
        success_rate = solved / total if total > 0 else 0

        # Average steps for solved instances
        solved_results = [r for r in results if r['solved']]
        avg_steps = (sum(r['steps'] for r in solved_results) / len(solved_results)
                     if solved_results else 0)

        # Identify best performing configurations
        analysis = {
            'total_runs': total,
            'success_rate': success_rate,
            'average_steps': avg_steps,
            'best_result': min(solved_results, key=lambda x: x['steps']) if solved_results else None
        }

        self.memory.track_journey(f"Analysis: {success_rate:.1%} success rate")
        return analysis

#
==========================================================================
====
# LAYER 4: META-LEARNING ENGINE
#
==========================================================================
====

class MetaLearningEngine:
    """Metalearnara + Evolvia + Spirala - Learn to improve"""

    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]:
        """Assign fitness scores based on performance"""
```

```python
    # Group results by algorithm
    algo_results = defaultdict(list)
    for result in results:
        algo_results[result['algorithm']].append(result)

    # Calculate fitness for each algorithm
    for algo in algorithms:
        algo_results_list = algo_results[algo.name]
        if algo_results_list:
            # Fitness = success_rate - (normalized_steps / 1000)
            success_rate = sum(1 for r in algo_results_list if r['solved']) / len(algo_results_list)
            avg_steps = sum(r['steps'] for r in algo_results_list) / len(algo_results_list)
            algo.fitness = success_rate - (avg_steps / 1000)
        else:
            algo.fitness = 0.0

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

def evolve_generation(self, algorithms: List[AlgorithmTemplate],
                      generator: GenerationEngine,
                      keep_top: int = 5) -> List[AlgorithmTemplate]:
    """Spirala: Evolve next generation through selection and mutation"""
    self.generation += 1

    # Keep top performers
    next_gen = algorithms[:keep_top].copy()

    # Generate new algorithms through mutation of top performers
    while len(next_gen) < len(algorithms):
        parent = random.choice(algorithms[:keep_top])
        child = generator.mutate_algorithm(parent)
        next_gen.append(child)

    avg_fitness = sum(a.fitness for a in algorithms) / len(algorithms) if algorithms else 0
    self.fitness_history.append(avg_fitness)

    self.memory.track_journey(
        f"Generation {self.generation}: avg_fitness={avg_fitness:.4f}"
    )

    return next_gen
```

```
#
================================================================================
====
# LAYER 5: KNOWLEDGE SYNTHESIS ENGINE
#
================================================================================
====

class KnowledgeEngine:
    """Sophira + Athena + Pyros - Wisdom accumulation"""

    def __init__(self, memory: MemorySubstrate):
        self.memory = memory
        self.insights = []

    def synthesize_insights(self, algorithms: List[AlgorithmTemplate],
                           analysis: Dict) -> List[str]:
        """Extract strategic insights from results"""
        insights = []

        # Find best performing algorithms
        top_algos = sorted(algorithms, key=lambda a: a.fitness, reverse=True)[:3]

        if top_algos:
            best = top_algos[0]
            insights.append(f"Best algorithm: {best.name} (fitness: {best.fitness:.4f})")
            insights.append(f"  Heuristics: {', '.join(best.heuristics)}")
            insights.append(f"  Strategies: {', '.join(best.strategies)}")

            # Find common patterns in top performers
            common_heuristics = defaultdict(int)
            common_strategies = defaultdict(int)

            for algo in top_algos:
                for h in algo.heuristics:
                    common_heuristics[h] += 1
                for s in algo.strategies:
                    common_strategies[s] += 1

            if common_heuristics:
                most_common_h = max(common_heuristics.items(), key=lambda x: x[1])
                insights.append(f"  Effective heuristic: {most_common_h[0]} (used in
{most_common_h[1]}/3 top)")
```

```
        if common_strategies:
            most_common_s = max(common_strategies.items(), key=lambda x: x[1])
            insights.append(f"  Effective strategy: {most_common_s[0]} (used in
{most_common_s[1]}/3 top)")

        self.insights.extend(insights)
        self.memory.track_journey(f"Synthesized {len(insights)} insights")

        return insights

#
============================================================================
====
# LAYER 6: ORCHESTRATION & CONTROL
#
============================================================================
====

class SystemOrchestrator:
    """Athena + Leviathan - Coordinate all subsystems"""

    def __init__(self):
        # Initialize all layers
        self.memory = MemorySubstrate()
        self.generator = GenerationEngine(self.memory)
        self.executor = ExecutionEngine(self.memory)
        self.analyzer = AnalysisEngine(self.memory)
        self.meta_learner = MetaLearningEngine(self.memory)
        self.knowledge = KnowledgeEngine(self.memory)

        # System state
        self.population_size = 10
        self.test_instances_per_gen = 5
        self.current_population = []

    def initialize(self):
        """Initialize first generation of algorithms"""
        print("🌟 INITIALIZING SAT ALGORITHM DISCOVERY SYSTEM")
        print("=" * 60)

        self.current_population = [
            self.generator.generate_algorithm()
            for _ in range(self.population_size)
        ]
```

```
        self.memory.checkpoint({
            'generation': 0,
            'population_size': self.population_size
        })

        print(f"✓ Generated initial population of {self.population_size} algorithms")

    def run_generation(self, generation_num: int) -> Dict:
        """Execute one complete generation cycle"""
        print(f"\n{'='*60}")
        print(f"GENERATION {generation_num}")
        print(f"{'='*60}")

        # Generate test instances
        test_instances = [
            SATInstance.generate_random(
                num_vars=random.randint(10, 20),
                num_clauses=random.randint(20, 40)
            )
            for _ in range(self.test_instances_per_gen)
        ]

        # Execute all algorithms on all instances
        all_results = []
        for algo in self.current_population:
            for instance in test_instances:
                result = self.executor.execute_algorithm(algo, instance)
                all_results.append(result)

        # Analyze results
        analysis = self.analyzer.analyze_results(all_results)
        print(f"\n📊 ANALYSIS:")
        print(f"  Success Rate: {analysis['success_rate']:.1%}")
        print(f"  Avg Steps: {analysis['average_steps']:.1f}")

        # Evaluate and evolve
        self.current_population = self.meta_learner.evaluate_population(
            self.current_population,
            all_results
        )

        # Synthesize insights
        insights = self.knowledge.synthesize_insights(
```

```
            self.current_population,
            analysis
        )

        print(f"\n💡 INSIGHTS:")
        for insight in insights:
            print(f"  {insight}")

        # Evolve to next generation
        self.current_population = self.meta_learner.evolve_generation(
            self.current_population,
            self.generator
        )

        # Checkpoint state
        state = {
            'generation': generation_num,
            'best_fitness': self.current_population[0].fitness,
            'avg_fitness': sum(a.fitness for a in self.current_population) / len(self.current_population)
        }
        self.memory.checkpoint(state)

        return state

    def run_experiment(self, num_generations: int = 10):
        """Run complete experiment"""
        self.initialize()

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

        # Final report
        print(f"\n{'='*60}")
        print("EXPERIMENT COMPLETE")
        print(f"{'='*60}")
        print(f"\nFitness Evolution:")
        for i, fitness in enumerate(self.meta_learner.fitness_history):
            print(f"  Generation {i+1}: {fitness:.4f}")

        print(f"\n📈 Journey Log ({len(self.memory.journey_log)} events):")
        for event in self.memory.journey_log[-10:]:
            print(f"  {event['event']}")

        print(f"\n🏆 BEST ALGORITHM FOUND:")
```

```
        best = self.current_population[0]
        print(f"  Name: {best.name}")
        print(f"  Fitness: {best.fitness:.4f}")
        print(f"  Heuristics: {best.heuristics}")
        print(f"  Strategies: {best.strategies}")
        print(f"  Parameters: {best.parameters}")

#
============================================================================
====
# MAIN EXECUTION
#
============================================================================
====

if __name__ == "__main__":
    # Create and run the system
    system = SystemOrchestrator()

    # Run experiment with 10 generations
    system.run_experiment(num_generations=10)

    print("\n✨ System demonstration complete!")
    print("This is a simplified proof-of-concept.")
    print("Full OMNIMOIRE would include:")
    print("  - Formal verification (SMT solvers)")
    print("  - Complexity analysis (symbolic execution)")
    print("  - Advanced meta-learning (neural architecture search)")
    print("  - Distributed execution (multi-node parallelism)")
```

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

==
GENERATION 1
==

📊 RESULTS:
  Verified: 40/40 (100.0%)
  Avg Operations: 27

🏆 BEST: Algo_5

Fitness: 0.9985
Verified: 5/5
Heuristics: ['random_variable_selection']
Strategies: ['pure_literal_elimination', 'unit_propagation']
Complexity: O(~1.38^n) EXPONENTIAL
==
GENERATION 2
==

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 22

🏆 BEST: Algo_3_m74
Fitness: 0.9983
Verified: 5/5
Heuristics: ['random_variable_selection']
Strategies: ['pure_literal_elimination', 'unit_propagation']
Complexity: O(~1.27^n) EXPONENTIAL
==
GENERATION 3
==

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 18

🏆 BEST: Algo_3_m74_m17
Fitness: 0.9987
Verified: 5/5
Heuristics: ['random_variable_selection']
Strategies: ['pure_literal_elimination', 'unit_propagation']
Complexity: O(~1.29^n) EXPONENTIAL
==
GENERATION 4
==

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 20

🏆 BEST: Algo_2
Fitness: 0.9987
Verified: 5/5

Heuristics: ['most_constrained_first', 'random_variable_selection']
Strategies: ['unit_propagation']
Complexity: O(~1.28^n) EXPONENTIAL
==
GENERATION 5
==

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 14

🏆 BEST: Algo_2
Fitness: 0.9986
Verified: 5/5
Heuristics: ['most_constrained_first', 'random_variable_selection']
Strategies: ['unit_propagation']
Complexity: O(~1.25^n) EXPONENTIAL
==
GENERATION 6
==

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 18

🏆 BEST: Algo_2
Fitness: 0.9985
Verified: 5/5
Heuristics: ['most_constrained_first', 'random_variable_selection']
Strategies: ['unit_propagation']
Complexity: O(~1.22^n) EXPONENTIAL
==
GENERATION 7
==

📊 RESULTS:
Verified: 40/40 (100.0%)
Avg Operations: 15

🏆 BEST: Algo_2_m32_m77
Fitness: 0.9986
Verified: 5/5
Heuristics: ['activity_based', 'random_variable_selection']
Strategies: ['unit_propagation']

Complexity: O(~1.24^n) EXPONENTIAL
==
GENERATION 8
==

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

🏆 BEST: Algo_2_m32_m77
Fitness: 0.9987
Verified: 5/5
Heuristics: ['activity_based', 'random_variable_selection']
Strategies: ['unit_propagation']
Complexity: O(~1.23^n) EXPONENTIAL

==
EXPERIMENT COMPLETE
==

📈 Fitness Evolution:
Gen 1: 0.9973
Gen 2: 0.9978
Gen 3: 0.9982
Gen 4: 0.9980
Gen 5: 0.9986
Gen 6: 0.9982
Gen 7: 0.9985
Gen 8: 0.9987

🏆 FINAL BEST ALGORITHM:
Algo_2_m32_m77
Fitness: 0.9987
Success: 5/5 verified
Heuristics: ['activity_based', 'random_variable_selection']
Strategies: ['unit_propagation']

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

```
=============================================================================
>>>
```