Learning CTGV:

Proposed Accelerated Learning Strategy for beginners:

Phase 1 (1-2 weeks): Discover the Shapes
• Use gui.py (if available) to assemble visual networks.
• Test each Shape individually:
◦ ORIGIN is a source.
◦ FLOW lets signals pass.
◦ DECISOR makes decisions.
◦ RESONATOR amplifies patterns.
◦ INHIBITOR blocks.

Phase 2 (2-4 weeks): Projects 1 and 2
• Implement the intelligent traffic light.
• Then the anomaly detector.
• Golden rule: Draw the topology (logic map) on paper or in an app before coding.

Phase 3 (1-2 months): Projects 3 and 4
• Advance to route optimization.
• Then recommendation system.
• Document each topological decision.

Phase 4 (ongoing): Project 5 and beyond
• Master distributed simulation.
• Create your own Shapes.
• Contribute to education with open-source code.

Mastery Checklist
• I can differentiate each Shape and its behavior.
• I can model a simple problem into CTGV topology.
• I understand how coherence measures the network's "health".
• I know how to use the TemporalBindingArbiter to resolve ambiguities.
• I can visualize and interpret field propagation.
• I know when to use CTGV vs. traditional approaches.

Practical Application in 6 Levels
Based on this didactic material that organizes 5 practical projects in a structured path, focusing on progressive application and the intuitive development of topological modeling, I hope to convey the fundamental concepts of topological/geometric thinking and their applicability in an objective and easy-to-understand manner.

Level 1 – Conceptual Foundation: Why Topology?

Key Concept:

In CTGV, you don't program instructions, you **structure relations**.

Think of a city: streets (connections), traffic lights (deciders), squares (resonators), and blockages (inhibitors).

The system **propagates signals** like a mind that "thinks" in geometry.

Initial Analogy:

```
ORIGIN → FLOW → DECISOR → (AMPLIFIER or INHIBITOR)
(Source) (Channel) (Choice) (Amplifies or Blocks)
```

Mental Exercise (no code):

Imagine modeling a restaurant as a CTGV network:

- ORIGIN = kitchen
- FLOW = waiters
- DECISOR = head waiter
- MEMORY = inventory
- INHIBITOR = occupied table

How does the signal (order) propagate?

Level 2 – Project 1: Intelligent Traffic Light

Objective: Control intersections based on traffic sensors.

Proposed topology:

```
[SENSOR_STREET_A] → [DECISOR_CENTRAL] → [FLOW_TRAFFIC_LIGHT_A] (green)
↓
[INHIBITOR_TRAFFIC_LIGHT_B] (red)
```

Guided code:

Python:

```python
from enum import Enum

class Shape(Enum):
SENSOR = 1
DECISOR = 2
FLOW = 3
INHIBITOR = 4

class Gebit:
def __init__(self, shape, intensity, label):
```

```
self.shape = shape
self.intensity = intensity
self.label = label
self.connections = []

def connect_to(self, other_geb, weight):
self.connections.append((other_geb, weight))

class CTGVEngine:
def propagate(self, sensors):
final_states = {}
# Initialize states with sensor intensities
for sensor in sensors:
final_states[sensor.label] = sensor.intensity

# Calculate decider intensity based on sensors and weights
decider_intensity = 0
total_weight = 0
for sensor in sensors:
for (conn_geb, weight) in sensor.connections:
if conn_geb.shape == Shape.DECISOR:
decider_intensity += sensor.intensity * weight
total_weight += weight

if total_weight > 0:
decider_intensity /= total_weight
else:
decider_intensity = 0
final_states['Controller'] = decider_intensity

# Propagate to traffic lights considering weights from decider to lights
traffic_light_a = None
traffic_light_b = None

for (conn_geb, weight) in decider.connections:
if conn_geb.label == "Traffic_Light_A":
traffic_light_a = (conn_geb, weight)
elif conn_geb.label == "Traffic_Light_B":
traffic_light_b = (conn_geb, weight)

# Traffic Light A activated (green)
if traffic_light_a is not None:
intensity_a = decider_intensity * traffic_light_a[1]
final_states['Traffic_Light_A'] = intensity_a * 0.75

# Traffic Light B inhibited (red)
if traffic_light_b is not None:
intensity_b = decider_intensity * traffic_light_b[1]
final_states['Traffic_Light_B'] = intensity_b * 0.10
```

```
return {'final_states': final_states}

# 1. Create the gebits
sensor_a = Gebit(Shape.SENSOR, intensity=0.8, label="Sensor_A")
sensor_b = Gebit(Shape.SENSOR, intensity=0.3, label="Sensor_B")
decider = Gebit(Shape.DECISOR, intensity=0.0, label="Controller")
traffic_light_a = Gebit(Shape.FLOW, intensity=0.0, label="Traffic_Light_A")
traffic_light_b = Gebit(Shape.INHIBITOR, intensity=0.0, label="Traffic_Light_B")

# 2. Connect the topology
sensor_a.connect_to(decider, 0.9)
sensor_b.connect_to(decider, 0.9)
decider.connect_to(traffic_light_a, 1.0)
decider.connect_to(traffic_light_b, 1.0)

# 3. Run the simulation
engine = CTGVEngine()
result = engine.propagate([sensor_a, sensor_b])

print("text")
print(f"Sensor_A: {sensor_a.intensity:.2f}")
print(f"Sensor_B: {sensor_b.intensity:.2f}")
print(f"Controller: {result['final_states']['Controller']:.2f}")
print(f"Traffic_Light_A: {result['final_states']['Traffic_Light_A']:.2f} # Activated (green)")
print(f"Traffic_Light_B: {result['final_states']['Traffic_Light_B']:.2f} # Inhibited (red)")
```

Expected console visualization:

```
text
Sensor_A: 0.80
Sensor_B: 0.30
Controller: 0.55
Traffic_Light_A: 0.41 # Activated (green)
Traffic_Light_B: 0.06 # Inhibited (red)
```

Expected learning:

• SENSOR captures intensity (traffic).
• DECISOR chooses based on higher input.
• FLOW lets pass; INHIBITOR blocks.

Level 3 – Project 2: Transaction Anomaly Detector

Objective: Identify suspicious patterns in a financial network.
Advanced topology:

```
[ACCOUNT_1] → [NORMAL_TRANSACTION] → [MEMORY_PATTERN]
[ACCOUNT_2] → [ANOMALOUS_TRANSACTION] → [RESONATOR_ALERT] → [AMPLIFIER_SIGNAL]
```

Step by step:

1. Create accounts as MEMORY (store history).
2. Transactions are connections with weight = value.
3. High values activate RESONATOR.
4. Use ClarificationEngine to detect deviations.

This code models transactions, calculates entropy, activates alarms if anomalies are detected based on network entropy:

Essential code:

python

```
from enum import Enum
import math

# Extend Shape enum with MEMORY, RESONATOR, AMPLIFIER
class Shape(Enum):
SENSOR = 1
DECISOR = 2
FLOW = 3
INHIBITOR = 4
MEMORY = 5
RESONATOR = 6
AMPLIFIER = 7

class Gebit:
def __init__(self, shape, intensity, label):
self.shape = shape
self.intensity = intensity
self.label = label
self.connections = []

def connect_to(self, other_geb, weight):
self.connections.append((other_geb, weight))

def activate(self):
```

```
if self.shape == Shape.RESONATOR:
# Resonator amplifies its intensity
self.intensity = min(1.0, self.intensity + 0.7)

class CTGVEngine:
def propagate(self, gebits):
for geb in gebits:
# Propagate intensity through connections
for (conn_geb, weight) in geb.connections:
conn_geb.intensity = min(1.0, conn_geb.intensity + geb.intensity * weight)
states = {g.label: g.intensity for g in gebits}
return {'final_states': states}

def calculate_network_entropy(gebits):
# Entropy based on normalized intensity distribution
intensities = [g.intensity for g in gebits]
total = sum(intensities) + 1e-9
probs = [i / total for i in intensities if i > 0]
entropy = -sum(p * math.log(p) for p in probs)
return entropy

# Initial definitions
LIMIT = 0.7
THRESHOLD = 0.5

# Creating Account Gebits (MEMORY)
account_1 = Gebit(Shape.MEMORY, intensity=0.5, label="Account_1")
account_2 = Gebit(Shape.MEMORY, intensity=0.5, label="Account_2")

# Other network gebits
normal_transaction = Gebit(Shape.FLOW, intensity=0.0, label="Normal_Transaction")
anomalous_transaction = Gebit(Shape.FLOW, intensity=0.0, label="Anomalous_Transaction")
memory_pattern = Gebit(Shape.MEMORY, intensity=0.0, label="Memory_Pattern")
resonator_alert = Gebit(Shape.RESONATOR, intensity=0.0, label="Resonator_Alert")
amplifier_signal = Gebit(Shape.AMPLIFIER, intensity=0.0, label="Amplifier_Signal")

# Connecting the topology
account_1.connect_to(normal_transaction, weight=0.4)
normal_transaction.connect_to(memory_pattern, weight=0.8)

account_2.connect_to(anomalous_transaction, weight=0.9) # suspicious transaction
anomalous_transaction.connect_to(resonator_alert, weight=1.0)
resonator_alert.connect_to(amplifier_signal, weight=1.2)

# Activating resonator for high value
value = 0.9
if value > LIMIT:
account_2.connect_to(resonator_alert, weight=0.9)
resonator_alert.activate()
```

```
# Simulate propagation
engine = CTGVEngine()
gebits = [account_1, account_2, normal_transaction, anomalous_transaction, memory_pattern,
resonator_alert, amplifier_signal]
result = engine.propagate(gebits)

# Calculate network entropy
entropy = calculate_network_entropy(gebits)

# Result
print(f"Network Entropy: {entropy:.2f}")
if entropy > THRESHOLD:
print("    Possible fraud detected!")
else:
print("    Network stable")

# Show current intensities
for geb in gebits:
print(f"{geb.label}: {geb.intensity:.2f}")
```

Expected console visualization:

```
Network Entropy: 1.78
   Possible fraud detected!
Account_1: 0.50
Account_2: 0.50
Normal_Transaction: 0.20
Anomalous_Transaction: 0.45
Memory_Pattern: 0.16
Resonator_Alert: 1.00
Amplifier_Signal: 1.00
```

Expected learning:

• Stable networks have low topological entropy.
• Anomalies resonate and amplify signals.

Level 4 – Project 3: Logistics Route Optimizer

Objective: Find the best path on a map with obstacles.

Coherence field concept:

ORIGIN (warehouse) → propagates through all FLOWs (roads)
→ finds RESONATOR (destination)
→ INHIBITOR (blockage) diverts the signal

Implementation:
Python

```
import math
from enum import Enum

class Shape(Enum):
SENSOR = 1
DECISOR = 2
FLOW = 3
INHIBITOR = 4
MEMORY = 5
RESONATOR = 6
AMPLIFIER = 7
ORIGIN = 8

class Gebit:
def __init__(self, shape, intensity, label):
self.shape = shape
self.intensity = intensity
self.label = label
self.connections = []

def connect_to(self, other_geb, weight):
self.connections.append((other_geb, weight))

class CTGVEngine:
def propagate(self, starting_gebits):
all_gebits = set(starting_gebits)
for _ in range(10):
next_intensities = {}
current_gebits = list(all_gebits) # avoid modification during iteration
for geb in current_gebits:
for (conn_geb, weight) in geb.connections:
if conn_geb.shape == Shape.INHIBITOR:
continue
increment = geb.intensity * weight * 0.6
if conn_geb in next_intensities:
next_intensities[conn_geb] = max(next_intensities[conn_geb], increment)
else:
```

```
next_intensities[conn_geb] = increment
all_gebits.add(conn_geb)
for geb in all_gebits:
if geb in next_intensities:
geb.intensity = max(geb.intensity, next_intensities[geb])
return {g.label: g.intensity for g in all_gebits}

def create_map_grid(rows, cols, obstacles):
grid = []
for r in range(rows):
row = []
for c in range(cols):
if (r, c) in obstacles:
gebit = Gebit(Shape.INHIBITOR, intensity=0.0, label=f"Block_{r}_{c}")
else:
gebit = Gebit(Shape.FLOW, intensity=0.0, label=f"Road_{r}_{c}")
row.append(gebit)
grid.append(row)
for r in range(rows):
for c in range(cols):
if c+1 < cols: grid[r][c].connect_to(grid[r][c+1], 1.0)
if c-1 >= 0: grid[r][c].connect_to(grid[r][c-1], 1.0)
if r+1 < rows: grid[r][c].connect_to(grid[r+1][c], 1.0)
if r-1 >= 0: grid[r][c].connect_to(grid[r-1][c], 1.0)
return grid

def visualize_ctgv_processing(grid, intensities):
print("\nMap with intensities after propagation:")
for r, row in enumerate(grid):
line = ""
for c, geb in enumerate(row):
intensity = intensities.get(geb.label, 0)
if geb.shape == Shape.INHIBITOR:
line += " XX "
else:
if intensity > 0.75:
line += " ## "
elif intensity > 0.5:
line += " ++ "
elif intensity > 0.25:
line += " .. "
else:
line += " -- "
print(line)

if __name__ == "__main__":
rows, cols = 5, 5
obstacles = [(2, 2), (3, 3)]

grid = create_map_grid(rows, cols, obstacles)
```

```
origin = grid[0][0]
origin.shape = Shape.ORIGIN
origin.intensity = 1.0

engine = CTGVEngine()
intensities = engine.propagate([origin])

visualize_ctgv_processing(grid, intensities)
```

Expected learning:

• ORIGIN emits a signal that flows through FLOWs.
• INHIBITORs block propagation.
• The path with the highest coherence is the most efficient.

Expected console visualization:

```
Map with intensities after propagation:
 ##  ++  ..  --  --
 ++  ..  --  --  --
 ..  --  XX  --  --
 --  --  --  XX  --
 --  --  --  --  --
```

The presented map shows a 5x5 grid simulating a logistics route network with signal propagation from an origin point:

• The symbols indicate signal intensity at each grid position:
◦ "##" indicates high intensity (more coherent and likely main route).
◦ "++" medium-high intensity.
◦ ".." medium-low intensity.
◦ "--" very low intensity or absence of signal.
• Cells marked with "XX" represent obstacles (INHIBITOR), blocked locations that prevent signal passage and force detours.
• Propagation starts in the top-left corner (origin/warehouse) with maximum intensity, which decreases with distance and upon encountering obstacles.
• The map reflects possible paths to reach different parts of the grid, showing detours caused by blockages.
This representation helps visualize the most efficient routes and how the system resonates the coherence of connections, simulating logistics optimization in an environment with obstacles.

Level 5 – Project 4: Explainable Recommendation System

Objective: Recommend based on topological similarity, not just history.

Architecture:

```
[USER] → [PRICE_NARRATIVE] → [TBA] → [RECOMMENDATION]
→ [CATEGORY_NARRATIVE]
```

How it works:

1. Each user and product is a **Gebit**.
2. Connections = interactions (viewing, purchase).
3. Multiple narratives compete in the **TemporalBindingArbiter**.
4. The most coherent narrative wins and generates an explanation.

Simplified code:

Python

```
from enum import Enum

class Shape(Enum):
ORIGIN = 1
DECISOR = 2
AMPLIFIER = 3

class Gebit:
def __init__(self, shape, intensity=0.0, label=""):
self.shape = shape
self.intensity = intensity
self.label = label
self.connections = []
def connect_to(self, other_gebit, weight=1.0):
self.connections.append((other_gebit, weight))

class TemporalBindingArbiter:
def __init__(self, engine=None):
self.engine = engine

def coherence_score(self, narrative):
# Simulate topological coherence, based on intensity and connections
base = narrative.intensity
conn_weight = sum(weight for _, weight in narrative.connections)
# Simple weighting for example coherence
coherence = min(1.0, base * 0.8 + conn_weight * 0.2)
return coherence

def resolve_ambiguity(self, narratives):
scores = {}
for narrative in narratives:
```

```
score = self.coherence_score(narrative)
scores[narrative.label] = score

dominant_label = max(scores, key=scores.get)
dominant_score = scores[dominant_label]
return {
'dominant_narrative': dominant_label,
'dominant_score': dominant_score,
'scores': scores
}

# Create narratives
price_narrative = Gebit(Shape.ORIGIN, intensity=0.7, label="Price_Narrative")
category_narrative = Gebit(Shape.ORIGIN, intensity=0.6, label="Category_Narrative")

# Connect examples of common structure that strengthens price narrative
decider = Gebit(Shape.DECISOR, label="Decider")
amplifier = Gebit(Shape.AMPLIFIER, label="Amplifier")

# Example connections that reflect topology and reinforcement
price_narrative.connect_to(decider, 0.9)
decider.connect_to(amplifier, 0.8)

category_narrative.connect_to(decider, 0.5)
# category_narrative less reinforced

# Resolve competition with TBA
tba = TemporalBindingArbiter()
result = tba.resolve_ambiguity([price_narrative, category_narrative])

# Explainable output
print("We recommend 'Product X' because:")
print(f"• {int(price_narrative.intensity*100)}% of users with similar topological profile purchased
it")
print(f"• Structural coherence: {result['dominant_score']:.2f}")
print(f"• Dominant narrative: {result['dominant_narrative']}")
```

Expected console visualization; Explainable output:

```
We recommend 'Product X' because:
• 70% of users with similar topological profile purchased it
• Structural coherence: 0.74
• Dominant narrative: Price_Narrative
```

Expected learning:

• TBA resolves competition between interpretations.

• Explainability comes from the network structure, not from statistics.

This code creates Gebit structures for two narratives, defines their topological connections simulating interactions, uses the arbiter to determine the dominant narrative, and presents a textual explanation with a coherence score.

Level 6: Epidemic Simulator

Objective: Model disease spread with interventions.

Complex topology:

```
[POPULATION] → [CONTACTS] → [INFECTION] → [INTERVENTION]
↓ ↓ ↓
[HEALTHY] [INFECTED] [VACCINATED]
```

DISTRIBUTED IMPLEMENTATION:

Python

```python
from enum import Enum
import random
from typing import List, Dict, Tuple

class Shape(Enum):
    """Specialized epidemiological shapes (CTGV)"""
    SUSCEPTIBLE = 1 # Healthy person (weak memory)
    INFECTED = 2 # Infected person (amplifier)
    RECOVERED = 3 # Recovered person (stable resonator)
    QUARANTINE = 4 # Quarantine (inhibitor)
    VACCINATED = 5 # Vaccine (protective transformer)
    HOSPITAL = 6 # Hospital (resource decider) - not used yet
    SUPERSREADER = 7 # Super-spreader (strong amplifier)

class EpidemiologicalGebit:
    """Gebit specialized in epidemiological modeling"""

    def __init__(self, shape: Shape, infection_prob: float = 0.0, label: str = ""):
        self.shape = shape
        self.infection_prob = infection_prob
        self.label = label
        self.connections: List[Tuple['EpidemiologicalGebit', float]] = []
        self.days_infected = 0
        self.recovery_days = 14
        self.isolation_factor = 1.0

        # Define initial infection level by shape
        if shape in (Shape.INFECTED, Shape.SUPERSREADER):
            self.infection_level = 0.8 if shape == Shape.INFECTED else 1.2
        elif shape == Shape.VACCINATED:
            self.infection_level = 0.1
        elif shape == Shape.QUARANTINE:
            self.infection_level = 0.05
            self.isolation_factor = 0.1
        else:
            self.infection_level = 0.0

    def connect_to(self, other: 'EpidemiologicalGebit', contact_frequency: float = 1.0):
```

```
        """Bidirectional connection with adjusted weight"""
        if other == self:
            return
        weight = contact_frequency * (1.0 - abs(self.infection_level - other.infection_level))
        if not any(o == other for o, _ in self.connections):
            self.connections.append((other, weight))
            other.connections.append((self, weight))

    def propagate_infection(self) -> float:
        """Propagate infection only if an amplifier"""
        if self.shape not in (Shape.INFECTED, Shape.SUPERSREADER):
            return 0.0

        total_spread = 0.0
        base_power = self.infection_level
        decay = max(0.0, 1.0 - self.days_infected / self.recovery_days)
        infection_power = base_power * decay

        for neighbor, weight in self.connections:
            if neighbor.shape == Shape.SUSCEPTIBLE:
                transmission_prob = infection_power * weight * neighbor.isolation_factor
                transmission_prob *= random.uniform(0.8, 1.2)
                transmission_prob = min(1.0, transmission_prob)

                if transmission_prob > neighbor.infection_prob:
                    neighbor.infection_prob = transmission_prob
                    total_spread += transmission_prob

        self.days_infected += 1
        return total_spread

    def update_state(self):
        """State transitions"""
        if self.shape in (Shape.INFECTED, Shape.SUPERSREADER) and self.days_infected >=
self.recovery_days:
            if random.random() < 0.95: # 95% recover
                self.shape = Shape.RECOVERED
                self.infection_level = 0.01

        elif self.shape == Shape.SUSCEPTIBLE and self.infection_prob > 0.7:
            self.shape = Shape.INFECTED
            self.infection_level = 0.8
            self.days_infected = 0

class EpidemicSimulator:
    """Simulator based on CTGV topology"""

    def __init__(self, population_size: int = 100):
        self.population: List[EpidemiologicalGebit] = []
        self.interventions: Dict[str, float] = {}
        self.day = 0
        self._create_population(population_size)
```

```
        self._create_social_network()

    def _create_population(self, size: int):
        # Patient zero
        self.population.append(EpidemiologicalGebit(Shape.INFECTED, 1.0, "Patient_Zero"))

        for i in range(1, size):
            if i < size * 0.05:
                self.population.append(EpidemiologicalGebit(Shape.VACCINATED, 0.0,
f"Vaccinated_{i}"))
            elif i < size * 0.10:
                self.population.append(EpidemiologicalGebit(Shape.QUARANTINE, 0.0,
f"Quarantine_{i}"))
            else:
                self.population.append(EpidemiologicalGebit(Shape.SUSCEPTIBLE, 0.0,
f"Person_{i}"))

    def _create_social_network(self):
        random.shuffle(self.population)
        # Family clusters
        i = 0
        while i < len(self.population):
            cluster_size = random.randint(3, 5)
            cluster = self.population[i:i+cluster_size]
            for p1 in cluster:
                for p2 in cluster:
                    if p1 != p2:
                        p1.connect_to(p2, 0.9)
            i += cluster_size

        # Random social connections
        for person in self.population:
            for _ in range(random.randint(4, 12)):
                other = random.choice(self.population)
                if other != person and not any(o == other for o, _ in person.connections):
                    person.connect_to(other, random.uniform(0.2, 0.6))

        # Super-spreaders
        num = max(2, int(len(self.population) * 0.02))
        for s in random.sample(self.population, num):
            if s.shape == Shape.SUSCEPTIBLE:
                s.shape = Shape.SUPERSREADER
                s.infection_level = 1.2
                for _ in range(20):
                    other = random.choice(self.population)
                    if other != s:
                        s.connect_to(other, 0.8)

    def apply_intervention(self, intervention_type: str, intensity: float):
        self.interventions[intervention_type] = intensity

        if intervention_type == "lockdown":
```

```
            for p in self.population:
                p.isolation_factor *= (1.0 - intensity)

        elif intervention_type == "vaccination_campaign":
            targets = [p for p in self.population if p.shape == Shape.SUSCEPTIBLE]
            num = int(len(targets) * intensity)
            for p in random.sample(targets, min(num, len(targets))):
                p.shape = Shape.VACCINATED
                p.infection_level = 0.1

        elif intervention_type == "mass_testing":
            infected = [p for p in self.population if p.shape in (Shape.INFECTED,
Shape.SUPERSREADER)]
            for p in infected:
                if random.random() < intensity:
                    p.shape = Shape.QUARANTINE
                    p.isolation_factor = 0.05

        elif intervention_type == "social_distancing":
            for p in self.population:
                p.connections = [(n, w * (1.0 - intensity)) for n, w in p.connections]

    def simulate_day(self) -> Dict[str, float]:
        self.day += 1
        total_spread = sum(p.propagate_infection() for p in self.population)
        for p in self.population:
            p.update_state()
        stats = self._calculate_statistics()
        stats["day"] = self.day
        stats["total_spread"] = total_spread
        return stats

    def _calculate_statistics(self) -> Dict[str, float]:
        counts = {s: 0 for s in Shape}
        for p in self.population:
            counts[p.shape] += 1

        infected = counts[Shape.INFECTED] + counts[Shape.SUPERSREADER]
        total_conn_infected = sum(len(p.connections) for p in self.population if p.shape in
(Shape.INFECTED, Shape.SUPERSREADER))
        r_effective = (total_conn_infected / infected * 0.15) if infected > 0 else 0.0
        coherence = 1.0 - (infected / len(self.population))

        return {
            "infected": infected,
            "susceptible": counts[Shape.SUSCEPTIBLE],
            "recovered": counts[Shape.RECOVERED],
            "vaccinated": counts[Shape.VACCINATED],
            "quarantined": counts[Shape.QUARANTINE],
            "superspreader": counts[Shape.SUPERSREADER],
            "r_effective": r_effective,
            "coherence": coherence,
```

```
            "total_population": len(self.population)
        }

class EpidemicPolicyArbiter:
    """Policy arbiter with topological explainability"""

    def __init__(self, simulator: EpidemicSimulator):
        self.simulator = simulator
        self.policy_scores: Dict[str, float] = {}

    def evaluate_policy(self, policy_name: str, intensity: float, duration_days: int = 30) -> float:
        # Complete state backup
        backup = [
            (p.shape, p.infection_prob, p.days_infected, p.isolation_factor, p.infection_level,
p.connections[:])
            for p in self.simulator.population
        ]
        day_backup = self.simulator.day

        self.simulator.apply_intervention(policy_name, intensity)
        total_infected = sum(self.simulator.simulate_day()["infected"] for _ in range(duration_days))
        avg_infected = total_infected / duration_days
        score = 1.0 - (avg_infected / len(self.simulator.population))

        # Complete restoration
        self.simulator.day = day_backup
        for i, person in enumerate(self.simulator.population):
            shape, prob, days, iso, level, conns = backup[i]
            person.shape = shape
            person.infection_prob = prob
            person.days_infected = days
            person.isolation_factor = iso
            person.infection_level = level
            person.connections = conns[:]

        self.policy_scores[policy_name] = score
        return score

    def recommend_best_policy(self, policies: List[Tuple[str, float]]) -> Dict:
        self.policy_scores.clear()
        for name, intensity in policies:
            self.evaluate_policy(name, intensity)

        best = max(self.policy_scores.items(), key=lambda x: x[1])
        return {
            "recommended_policy": best[0],
            "policy_score": best[1],
            "all_scores": self.policy_scores.copy(),
            "interpretation": self._generate_interpretation(best)
        }

    def _generate_interpretation(self, best_policy: Tuple[str, float]) -> str:
```

```
        policy, score = best_policy
        texts = {
            "lockdown": "Mobility restrictions to reduce social connections",
            "vaccination_campaign": "Mass immunization to create topological barriers",
            "mass_testing": "Identification and isolation of infected nodes",
            "social_distancing": "Direct reduction of social connection weights"
        }
        base = texts.get(policy, "Topological intervention")
        efficacy = "highly effective" if score > 0.8 else "moderately effective" if score > 0.6 else
"slightly effective"
        return f"{base} - {efficacy} (score: {score:.2f})"

# ============================================
# COMPLETE DEMONSTRATION
# ============================================

print("=" * 60)
print("      EPIDEMIC SIMULATOR - CTGV TOPOLOGICAL MODELING")
print("=" * 60)

simulator = EpidemicSimulator(population_size=100)

print(f"\n      Population created: {len(simulator.population)} individuals")
stats0 = simulator._calculate_statistics()
print(f" • Initial infected: {stats0['infected']}")
print(f" • Vaccinated: {stats0['vaccinated']}")
print(f" • Super-spreaders: {stats0['superspreader']}")

print("\n      PHASE 1: Natural propagation (40 days)")
print("-" * 50)
for day in range(40):
    stats = simulator.simulate_day()
    if day % 8 == 0 or day == 39:
        print(f"Day {stats['day']:3d}: {stats['infected']:3d} infected | "
              f"R = {stats['r_effective']:.2f} | Coherence = {stats['coherence']:.3f}")

print("\n      PHASE 2: Policy evaluation")
print("-" * 50)
arbiter = EpidemicPolicyArbiter(simulator)

policies_to_test = [
    ("lockdown", 0.8),
    ("vaccination_campaign", 0.6),
    ("mass_testing", 0.75),
    ("social_distancing", 0.7)
]

recommendation = arbiter.recommend_best_policy(policies_to_test)

print(f"\n      Recommended policy: {recommendation['recommended_policy']}")
print(f" Score: {recommendation['policy_score']:.3f}")
print(f" Justification: {recommendation['interpretation']}")
```

```
print("\n      Policy scores:")
for p, s in recommendation['all_scores'].items():
    print(f" • {p:22s}: {s:.3f}")

print("\n      PHASE 3: Applying best policy + 40 days")
print("-" * 50)
policy_name = recommendation['recommended_policy']
intensity = next((i for n, i in policies_to_test if n == policy_name), 0.5)
simulator.apply_intervention(policy_name, intensity)

for day in range(40):
    stats = simulator.simulate_day()
    if day % 10 == 0 or day == 39:
        print(f"Day {stats['day']:3d}: {stats['infected']:3d} infected | "
              f"R = {stats['r_effective']:.2f} | Coherence = {stats['coherence']:.3f}")

final_stats = simulator._calculate_statistics()
print("\n" + "=" * 60)
print("      FINAL REPORT")
print("=" * 60)
print(f" Infected: {final_stats['infected']}")
print(f" Recovered: {final_stats['recovered']}")
print(f" Vaccinated: {final_stats['vaccinated']}")
print(f" Effective R: {final_stats['r_effective']:.2f}")
print(f" Coherence: {final_stats['coherence']:.3f}")

diag = "      CONTROLLED" if final_stats['coherence'] > 0.85 else \
       "      MODERATE" if final_stats['coherence'] > 0.6 else \
       "      ACTIVE" if final_stats['coherence'] > 0.4 else "      UNCONTROLLED"
print(f"\n      DIAGNOSIS: {diag}")

print("\n      PRESERVED TOPOLOGICAL INSIGHTS:")
print(" • Shape defines function (amplifier, inhibitor, transformer)")
print(" • Coherence measures structural resilience of the network")
print(" • Interventions alter topology (weights, nodes, edges)")
print(" • Super-spreaders = high centrality")

print("\n" + "=" * 60)
print("Enhanced model: realistic propagation,")
print("=" * 60)
```

Expected console visualization:

text

```
============================================================
    EPIDEMIC SIMULATOR - CTGV TOPOLOGICAL MODELING
============================================================

    Population created: 100 individuals
• Initial infected: 3
• Vaccinated: 4
• Super-spreaders: 2

    PHASE 1: Natural propagation (40 days)
------------------------------------------------
Day   1:   9 infected | R = 2.95 | Coherence = 0.910
Day   9:  12 infected | R = 2.71 | Coherence = 0.880
Day  17:   1 infected | R = 2.55 | Coherence = 0.990
Day  25:   0 infected | R = 0.00 | Coherence = 1.000
Day  33:   0 infected | R = 0.00 | Coherence = 1.000
Day  40:   0 infected | R = 0.00 | Coherence = 1.000

    PHASE 2: Policy evaluation
------------------------------------------------

    Recommended policy: lockdown
 Score: 1.000
 Justification: Mobility restrictions to reduce social connections - highly effective (score: 1.00)

    Policy scores:
• lockdown             : 1.000
• vaccination_campaign  : 1.000
• mass_testing          : 1.000
• social_distancing     : 1.000

    PHASE 3: Applying best policy + 40 days
------------------------------------------------
Day  41:   0 infected | R = 0.00 | Coherence = 1.000
Day  51:   0 infected | R = 0.00 | Coherence = 1.000
Day  61:   0 infected | R = 0.00 | Coherence = 1.000
Day  71:   0 infected | R = 0.00 | Coherence = 1.000
Day  80:   0 infected | R = 0.00 | Coherence = 1.000

============================================================
    FINAL REPORT
============================================================
 Infected: 0
 Recovered: 12
 Vaccinated: 4
 Effective R: 0.00
 Coherence: 1.000
```

DIAGNOSIS: CONTROLLED

PRESERVED TOPOLOGICAL INSIGHTS:
• Shape defines function (amplifier, inhibitor, transformer)
• Coherence measures structural resilience of the network
• Interventions alter topology (weights, nodes, edges)
• Super-spreaders = high centrality

==========
Enhanced model: realistic propagation,
==========

Expected learning:

• Propagation as network dynamics.
• Interventions are **INHIBITORs** or **RESONATORs**.
• Coherence measures system organization.

Final Tip

"In CTGV, you don't program behaviors; you design relational ecosystems where intelligence emerges from structure."
Start with small networks (5-10 gebits), test iteratively, and document each discovery. The curve is steep, but each step reveals a new way of thinking about complex problems.
All of this can be improved, optimized, and developed with assistance from Artificial Intelligences trained/implemented with this very tool.

Repository: https://github.com/Bear-urso/CTGV-System-V-1.5
Publication: https://doi.org/10.5281/zenodo.18360864

Author: Begnomar dos Santos Porto (0009-0002-6109-7443) – ORCID

Prior Artifact Record:

Bitcoin Block 933881, mined on January 26, 2026 at 03:31:34 UTC. Which attests to the existence of the Timestamp record of 6fb6036d3294 from the file (Learning CTGV 26.01.26.pdf 485.7 kB)
HASH SHA256: 6fb6036d32943b5579fac44b40d8b0707c9e2587a97efea22452fcc2e1cfd5d1