Aprendendo CTGV:

Estratégia proposta de Aprendizado Acelerado para iniciantes:

Fase 1 (1-2 semanas): Descobrir as Formas

•Use `gui.py` (se disponível) para montar redes visuais.

•Teste cada **`Shape`** individualmente:

•**`ORIGIN`** é uma fonte.

•**`FLOW`** deixa passar sinal.

•**`DECISOR`** toma decisões.

•**`RESONATOR`** amplifica padrões.

•**`INHIBITOR`** bloqueia.

Fase 2 (2-4 semanas): Projetos 1 e 2

•Implemente o **semáforo inteligente**.

•Depois o **detector de anomalias**.

•**Regra de ouro:** Desenhe a topologia (mapa da lógica) no papel ou aplicativo antes de codificar.

Fase 3 (1-2 meses): Projetos 3 e 4

•Avance para **otimização de rotas**.

•Depois **sistema de recomendação**.

•Documente cada decisão topológica.

Fase 4 (contínua): Projeto 5 e além

•Domine a **simulação distribuída**.

•Crie suas próprias **`Shapes`.**

•Contribua para a educação com o código aberto.

Checklist de Domínio

•Sei diferenciar cada **`Shape`** e seu comportamento.

•Consigo modelar um problema simples em topologia CTGV.

•Entendo como a coerência mede a "saúde" da rede.

•Sei usar o `TemporalBindingArbiter` para resolver ambiguidades.

•Consigo visualizar e interpretar a propagação de campos.

•Sei quando usar CTGV vs. abordagens tradicionais.

Aplicação Prática em 6 Níveis

A partir deste material didático que organiza **5 projetos práticos** em um caminho estruturado, com foco na **aplicação progressiva** e no **desenvolvimento intuitivo** da modelagem topológica espero repassar de maneira objetiva e fácil de entender os conceitos fundamentais, do pensamento topológico / geométrico e sua aplicabilidade.

Nível 1 – Fundamentação Conceitual: Por que Topologia?

Conceito-chave:
No CTGV, você não programa instruções, mas **estrutura relações**.
Pense em uma cidade: ruas (conexões), semáforos (decisores), praças (resonadores) e bloqueios (inibidores).
O sistema **propaga sinais** como uma mente que “pensa” em geometria.

Analogia Inicial:

ORIGIN → FLOW → DECISOR → (AMPLIFIER ou INHIBITOR) (Fonte) (Canal) (Escolha) (Amplifica ou Bloqueia)

Exercício Mental (sem código):

Imagine modelar um **restaurante** como uma rede CTGV:

•ORIGIN = cozinha
•FLOW = garçons
•DECISOR = chefe de sala
•MEMORY = estoque
•INHIBITOR = mesa ocupada
Como o sinal (pedido) se propaga?

Nível 2 – Projeto 1: Semáforo Inteligente

Objetivo: Controlar cruzamentos com base em sensores de tráfego.

Topologia proposta:

```
[SENSOR_RUA_A] → [DECISOR_CENTRAL] → [FLOW_SEMAFORO_A] (verde)
                        ↓
                [INHIBITOR_SEMAFORO_B] (vermelho)
```

Código guiado:

Pyton:

```
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 = {}
# Inicializa os estados com a intensidade dos sensores
for sensor in sensors:
final_states[sensor.label] = sensor.intensity

# Calcula a intensidade do decisor baseado nos sensores e pesos
decisor_intensity = 0
total_weight = 0
for sensor in sensors:
for (conn_geb, weight) in sensor.connections:
if conn_geb.shape == Shape.DECISOR:
decisor_intensity += sensor.intensity * weight
total_weight += weight

if total_weight > 0:
decisor_intensity /= total_weight
else:
decisor_intensity = 0
final_states['Controlador'] = decisor_intensity

# Propaga para os semáforos considerando pesos dos decisor para semáforos
semaforo_a = None
semaforo_b = None

for (conn_geb, weight) in decisor.connections:
if conn_geb.label == "Semaforo_A":
semaforo_a = (conn_geb, weight)
elif conn_geb.label == "Semaforo_B":
semaforo_b = (conn_geb, weight)

# Semaforo A ativado (verde)
if semaforo_a is not None:
intensity_a = decisor_intensity * semaforo_a[1]
final_states['Semaforo_A'] = intensity_a * 0.75

# Semaforo B inibido (vermelho)
if semaforo_b is not None:
intensity_b = decisor_intensity * semaforo_b[1]
final_states['Semaforo_B'] = intensity_b * 0.10

return {'final_states': final_states}

# 1. Crie os gebits
sensor_a = Gebit(Shape.SENSOR, intensity=0.8, label="Sensor_A")
sensor_b = Gebit(Shape.SENSOR, intensity=0.3, label="Sensor_B")
decisor = Gebit(Shape.DECISOR, intensity=0.0, label="Controlador")
semaforo_a = Gebit(Shape.FLOW, intensity=0.0, label="Semaforo_A")
semaforo_b = Gebit(Shape.INHIBITOR, intensity=0.0, label="Semaforo_B")
```

```
# 2. Conecte a topologia
sensor_a.connect_to(decisor, 0.9)
sensor_b.connect_to(decisor, 0.9)
decisor.connect_to(semaforo_a, 1.0)
decisor.connect_to(semaforo_b, 1.0)

# 3. Execute a simulação
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"Controlador: {result['final_states']['Controlador']:.2f}")
print(f"Semaforo_A: {result['final_states']['Semaforo_A']:.2f} # Ativado (verde)")
print(f"Semaforo_B: {result['final_states']['Semaforo_B']:.2f} # Inibido (vermelho)")
```

Visualização esperada no console:

```
text
Sensor_A: 0.80
Sensor_B: 0.30
Controlador: 0.55
Semaforo_A: 0.41   # Ativado (verde)
Semaforo_B: 0.06   # Inibido (vermelho)
```

Aprendizado esperado:

- `SENSOR` captura intensidade (tráfego).
- `DECISOR` escolhe com base na entrada maior.
- `FLOW` deixa passar; `INHIBITOR` bloqueia.

Nível 3 – Projeto 2: Detector de Anomalias em Transações

Objetivo: Identificar padrões suspeitos em uma rede financeira.

Topologia avançada:

```
[CONTA_1] → [TRANSACAO_NORMAL] → [MEMORY_PADRAO]
[CONTA_2] → [TRANSACAO_ANOMALA] → [RESONATOR_ALERTA] → [AMPLIFIER_SINAL]
```

Passo a passo:

1. **Crie contas como** `MEMORY` (guardam histórico).
2. **Transações são conexões com peso = valor**.
3. **Valores altos ativam** `RESONATOR`.
4. **Use** `ClarificationEngine` **para detectar desvios**.

Este código modela as transações, calcula a entropia, ativa alarmes se anomalias são detectadas com base na entropia da rede:

Código essencial:

Pyton

```
from enum import Enum
import math

# Extende enum Shape com 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:
# Ressonador amplifica sua intensidade
self.intensity = min(1.0, self.intensity + 0.7)

class CTGVEngine:
def propagate(self, gebits):
for geb in gebits:
# Propaga intensidade pelas conexões
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):
# Entropia baseada na distribuição de intensidades normalizadas
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

# Definições iniciais
LIMITE = 0.7
THRESHOLD = 0.5

# Criando Gebits de Contas (MEMORY)
conta_1 = Gebit(Shape.MEMORY, intensity=0.5, label="Conta_1")
conta_2 = Gebit(Shape.MEMORY, intensity=0.5, label="Conta_2")

# Outros gebits da rede
transacao_normal = Gebit(Shape.FLOW, intensity=0.0, label="Transacao_Normal")
transacao_anomala = Gebit(Shape.FLOW, intensity=0.0, label="Transacao_Anomala")
memory_padrao = Gebit(Shape.MEMORY, intensity=0.0, label="Memory_Padrao")
resonator_alerta = Gebit(Shape.RESONATOR, intensity=0.0, label="Resonator_Alerta")
amplifier_sinal = Gebit(Shape.AMPLIFIER, intensity=0.0, label="Amplifier_Sinal")

# Conectando a topologia
conta_1.connect_to(transacao_normal, weight=0.4)
transacao_normal.connect_to(memory_padrao, weight=0.8)

conta_2.connect_to(transacao_anomala, weight=0.9) # transação suspeita
transacao_anomala.connect_to(resonator_alerta, weight=1.0)
resonator_alerta.connect_to(amplifier_sinal, weight=1.2)

# Ativando ressonador para valor alto
valor = 0.9
```

```
if valor > LIMITE:
conta_2.connect_to(resonator_alerta, weight=0.9)
resonator_alerta.activate()

# Simular propagação
engine = CTGVEngine()
gebits = [conta_1, conta_2, transacao_normal, transacao_anomala, memory_padrao,
resonator_alerta, amplifier_sinal]
result = engine.propagate(gebits)

# Calcular entropia da rede
entropy = calculate_network_entropy(gebits)

# Resultado
print(f"Entropia da Rede: {entropy:.2f}")
if entropy > THRESHOLD:
print("     Possível fraude detectada!")
else:
print("     Rede estável")

# Mostrar intensidades atuais
for geb in gebits:
print(f"{geb.label}: {geb.intensity:.2f}")
```

Visualização esperada no console:

```
Entropia da Rede: 1.78
     Possível fraude detectada!
Conta_1: 0.50
Conta_2: 0.50
Transacao_Normal: 0.20
Transacao_Anomala: 0.45
Memory_Padrao: 0.16
Resonator_Alerta: 1.00
Amplifier_Sinal: 1.00
```

Aprendizado esperado:

- Redes estáveis têm **baixa entropia topológica**.
- Anomalias **ressonam** e amplificam sinais.

Nível 4 – Projeto 3: Otimizador de Rotas Logísticas

Objetivo: Encontrar o melhor caminho em um mapa com obstáculos.

Conceito de campo de coerência:

ORIGIN (armazém) → propaga por todos os FLOWs (estradas)

→ encontra RESONATOR (destino)

→ INHIBITOR (bloqueio) desvia o sinal

Implementação:

Pyton

```
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) # evita modificação durante iteração
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("\nMapa com intensidades após propagação:")
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)
```

Visualize o caminho ótimo com `visualize_ctgv_processing()`.

Aprendizado:

•`ORIGIN` emite sinal que **flui por `FLOW`s**.

•`INHIBITOR`s bloqueiam propagação.

•O caminho com **maior coerência** é o mais eficiente.

Visualização esperada no console:

Mapa com intensidades após propagação:

++ .. -- --

++ .. -- -- --

.. -- XX -- --

-- -- -- XX --

-- -- -- -- --

O mapa apresentado mostra uma grade 5x5 simulando uma malha de rotas logísticas com propagação de sinal a partir de um ponto origem:

●Os símbolos indicam a intensidade do sinal em cada posição da grade:
●"##" indica intensidade alta (mais coerente e provável rota principal).
●"++" intensidade média-alta.
●".." intensidade média-baixa.
●"--" intensidade muito baixa ou ausência de sinal.
●As células marcadas com "XX" representam obstáculos (**INHIBITOR**), locais bloqueados que impedem a passagem do sinal e forçam desvios.

- A propagação começa no canto superior esquerdo (origem/armazém) com intensidade máxima que diminui conforme se distancia e encontra obstáculos.
- O mapa reflete caminhos possíveis para chegar a diferentes partes da grade, mostrando os desvios causados pelos bloqueios.

Essa representação auxilia a visualizar as rotas mais eficientes e como o sistema ressoa a coerência das conexões, simulando otimização logística em um ambiente com obstáculos.

Nível 5 – Projeto 4:

Sistema de Recomendação Explicável

Objetivo: Recomendar com base em similaridade topológica, não apenas histórico.

Arquitetura:

```
[USUARIO] → [NARRATIVA_PRECO] → [TBA] → [RECOMENDACAO]
        → [NARRATIVA_CATEGORIA]
```

Como funciona:

1.**Cada usuário e produto é um `Gebit`**.

2.**Conexões = interações** (visualização, compra).

3.**Múltiplas narrativas competem** no `TemporalBindingArbiter`.

4.**A narrativa mais coerente vence** e gera explicação.

Código simplificado:

Pyton

```
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):
# Simular coerência topológica, baseada em intensidade e conexões
base = narrative.intensity
conn_weight = sum(weight for _, weight in narrative.connections)
# Simples ponderação para coerência do exemplo
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
}

# Criar narrativas
narrativa_preco = Gebit(Shape.ORIGIN, intensity=0.7, label="Narrativa_Preço")
narrativa_categoria = Gebit(Shape.ORIGIN, intensity=0.6, label="Narrativa_Categoria")

# Conectar exemplos de estrutura comum que fortalece narrativa preço
decisor = Gebit(Shape.DECISOR, label="Decisor")
amplificador = Gebit(Shape.AMPLIFIER, label="Amplificador")

# Exemplo conexões que refletem topologia e reforço
narrativa_preco.connect_to(decisor, 0.9)
decisor.connect_to(amplificador, 0.8)

narrativa_categoria.connect_to(decisor, 0.5)
# narrativa_categoria menos reforçada

# Resolver competição com TBA
```

```
tba = TemporalBindingArbiter()
resultado = tba.resolve_ambiguity([narrativa_preco, narrativa_categoria])

# Saída explicável
print("Recomendamos 'Produto X' porque:")
print(f"• {int(narrativa_preco.intensity*100)}% dos usuários com perfil topológico similar compraram")
print(f"• Coerência estrutural: {resultado['dominant_score']:.2f}")
print(f"• Narrativa dominante: {resultado['dominant_narrative']}")
```

Visualização esperada no console; Saída explicável:

```
Recomendamos 'Produto X' porque:
• 70% dos usuários com perfil topológico similar compraram
• Coerência estrutural: 0.74
• Narrativa dominante: Narrativa_Preço
```

Aprendizado:

•**TBA resolve competição entre interpretações**.

•**Explicabilidade vem da estrutura da rede**, não de estatísticas.

Esse código cria estruturas Gebit para duas narrativas, define suas conexões topológicas simulando interações, usa o arbiter para determinar a narrativa dominante e apresenta uma explicação textual com pontuação de coerência.

Nível 6: Simulador de Epidemias

Objetivo: Modelar propagação de doenças com intervenções.

Topologia complexa:

```
[POPULAÇÃO] → [CONTATOS] → [INFECÇÃO] → [INTERVENÇÃO]
      ↓          ↓           ↓
  [SAUDÁVEL]   [INFECTADO]  [VACINADO]
```

A IMPLEMENTAÇÃO DISTRIBUÍDA:

Pyton

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

class Shape(Enum):
"""Formas epidemiológicas especializadas (CTGV)"""
SUSCEPTIBLE = 1 # Pessoa saudável (memória fraca)
INFECTED = 2 # Pessoa infectada (amplificador)
RECOVERED = 3 # Pessoa recuperada (resonador estável)
QUARANTINE = 4 # Quarentena (inibidor)
VACCINATED = 5 # Vacina (transformador protetor)
HOSPITAL = 6 # Hospital (decisor de recursos) - não usado ainda
SUPERSREADER = 7 # Superdisseminador (amplificador forte)

class EpidemiologicalGebit:
"""Gebit especializado em modelagem epidemiológica"""

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

# Definir nível de infecção inicial por forma
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):
"""Conexão bidirecional com peso ajustado"""
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:
"""Propagar infecção apenas se for amplificador"""
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):
"""Transições de estado"""
if self.shape in (Shape.INFECTED, Shape.SUPERSREADER) and self.days_infected >=
self.recovery_days:
if random.random() < 0.95: # 95% recuperam
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:
"""Simulador baseado em topologia CTGV"""

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):
# Paciente 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)
# Clusters familiares
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

# Conexões sociais aleatórias
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))

# Superdisseminadores
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:
"""Arbitrador de políticas com explicabilidade topológica"""

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:
# Backup completo do estado
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))

# Restauração completa
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": "Restrições de mobilidade para reduzir conexões sociais",
"vaccination_campaign": "Imunização em massa para criar barreiras topológicas",
"mass_testing": "Identificação e isolamento de nós infectados",
"social_distancing": "Redução direta do peso das conexões sociais"
}
base = texts.get(policy, "Intervenção topológica")
efficacy = "altamente eficaz" if score > 0.8 else "moderadamente eficaz" if score > 0.6 else "pouco
eficaz"
return f"{base} - {efficacy} (score: {score:.2f})"


# ==========================================
# DEMONSTRAÇÃO COMPLETA
# ==========================================

print("=" * 60)
print("     SIMULADOR DE EPIDEMIAS - MODELAGEM TOPOLÓGICA CTGV")
print("=" * 60)

simulator = EpidemicSimulator(population_size=100)

print(f"\n     População criada: {len(simulator.population)} indivíduos")
stats0 = simulator._calculate_statistics()
print(f" • Infectados iniciais: {stats0['infected']}")
print(f" • Vacinados: {stats0['vaccinated']}")
print(f" • Superdisseminadores: {stats0['superspreader']}")

print("\n     FASE 1: Propagação natural (40 dias)")
print("-" * 50)
for day in range(40):
stats = simulator.simulate_day()
if day % 8 == 0 or day == 39:
print(f"Dia {stats['day']:3d}: {stats['infected']:3d} infectados | "
f"R = {stats['r_effective']:.2f} | Coerência = {stats['coherence']:.3f}")

print("\n     FASE 2: Avaliação de políticas")
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      Política recomendada: {recommendation['recommended_policy']}")
print(f" Score: {recommendation['policy_score']:.3f}")
print(f" Justificativa: {recommendation['interpretation']}")

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

print("\n      FASE 3: Aplicando melhor política + 40 dias")
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"Dia {stats['day']:3d}: {stats['infected']:3d} infectados | "
f"R = {stats['r_effective']:.2f} | Coerência = {stats['coherence']:.3f}")

final_stats = simulator._calculate_statistics()
print("\n" + "=" * 60)
print("      RELATÓRIO FINAL")
print("=" * 60)
print(f" Infectados: {final_stats['infected']}")
print(f" Recuperados: {final_stats['recovered']}")
print(f" Vacinados: {final_stats['vaccinated']}")
print(f" R efetivo: {final_stats['r_effective']:.2f}")
print(f" Coerência: {final_stats['coherence']:.3f}")

diag = "      CONTROLADA" if final_stats['coherence'] > 0.85 else \
"      MODERADA" if final_stats['coherence'] > 0.6 else \
"      ATIVA" if final_stats['coherence'] > 0.4 else "      DESCONTROLADA"
print(f"\n      DIAGNÓSTICO: {diag}")

print("\n      INSIGHTS TOPOLÓGICOS PRESERVADOS:")
print(" • Forma define função (amplificador, inibidor, transformador)")
print(" • Coerência mede resiliência estrutural da rede")
print(" • Intervenções alteram topologia (pesos, nós, arestas)")
print(" • Superdisseminadores = alta centralidade")

print("\n" + "=" * 60)
print("Modelo aprimorado: propagação realista,")
print("=" * 60)
```

Visualização esperada no console:

```
============================================================
    SIMULADOR DE EPIDEMIAS - MODELAGEM TOPOLÓGICA CTGV
============================================================

    População criada: 100 indivíduos
  • Infectados iniciais: 2
  • Vacinados: 4
  • Superdisseminadores: 1

    FASE 1: Propagação natural (40 dias)
----------------------------------------------------
Dia   1:   4 infectados | R = 3.56 | Coerência = 0.960
Dia   9:   4 infectados | R = 3.56 | Coerência = 0.960
Dia  17:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  25:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  33:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  40:   0 infectados | R = 0.00 | Coerência = 1.000




    FASE 2: Avaliação de políticas
----------------------------------------------------

    Política recomendada: lockdown
  Score: 1.000
  Justificativa: Restrições de mobilidade para reduzir conexões sociais - altamente eficaz (score:
1.00)

    Scores das políticas:
  • lockdown            : 1.000
  • vaccination_campaign  : 1.000
  • mass_testing        : 1.000
  • social_distancing     : 1.000

    FASE 3: Aplicando melhor política + 40 dias
----------------------------------------------------
Dia  41:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  51:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  61:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  71:   0 infectados | R = 0.00 | Coerência = 1.000
Dia  80:   0 infectados | R = 0.00 | Coerência = 1.000

============================================================
    RELATÓRIO FINAL
============================================================
  Infectados: 0
  Recuperados: 4
```

```
Vacinados: 4
R efetivo: 0.00
Coerência: 1.000

   DIAGNÓSTICO:      CONTROLADA

   INSIGHTS TOPOLÓGICOS PRESERVADOS:
• Forma define função (amplificador, inibidor, transformador)
• Coerência mede resiliência estrutural da rede
• Intervenções alteram topologia (pesos, nós, arestas)
• Superdisseminadores = alta centralidade

Modelo aprimorado: propagação realista
============================================================
```

Aprendizado:

•**Propagação como dinâmica de rede**.

•**Intervenções são `INHIBITOR`s ou `RESONATOR`s**.

•**Coerência mede organização do sistema**.

Dica Final

"No CTGV, você não programa comportamentos; você **projeta ecossistemas relacionais** onde a inteligência emerge da estrutura."

Comece com **redes pequenas** (5-10 gebits),**teste iterativamente**, e **documente cada descoberta**. A curva é íngreme, mas cada degrau revela uma nova forma de pensar sobre problemas complexos.

Tudo isso pode ser aprimorado, otimizado e desenvolvido com assistência das Inteligências Artificiais treinada / implementada com esta a própria ferramenta.

Repositório: https://github.com/Bear-urso/CTGV-System-V-1.5

Publicação: https://doi.org/10.5281/zenodo.18360864

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

Registro de anterioridade:

No **Bloco 933881 do Bitcoin**, **Minerado em 26,de janeiro de 2026 às 03:31:34 UTC.** Que atesta a existência do registro Timestamp of 6fb6036d3294 do **Arquivo (Aprendendo CTGV 26.01.26.pdf 485.7 kB) HASH SHA256**: **6fb6036d32943b5579fac44b40d8b0707c9e2587a97efea22452fcc2e1cfd5d1**

Data 26 de janeiro de 2026.