Jerzy Boksa, Bartosz Gacek

# Epidemia Strategii Społecznych (Zaraźliwe Zachowania)

## Proste reguły

- Sieć agentów połączonych relacjami w przestrzeni 2D
- Każdy agent ma strategię: altruista (płaci koszt dla sąsiadów) lub egoista (korzysta bez płacenia)
- Agenci kopiują strategię najbogatszego sąsiada z pewnym prawdopodobieństwem (deterministycznie lub proporcjonalnie)
- Niewielka losowa mutacja strategii zapobiega zamrożeniu systemu

## Opis problemu

Model bada, jak proste reguły interakcji społecznych prowadzą do złożonych zjawisk ewolucyjnych. Kluczowe pytanie: **czy kooperacja może się utrzymać w populacji, gdy egoizm jest strategią dominującą krótkoterminowo?**

Model symuluje dylemat współpracy: altruiści ponoszą koszt (C_A), aby przynosić korzyści sąsiadom (B_A), podczas gdy egoiści korzystają bez płacenia. Mechanizmy społecznego uczenia się, struktura przestrzenna i stochastyczność mogą zmieniać klasyczną dominację egoizmu.

## Model

**Struktura:**

- Siatka 100×100 (lub większa) z periodycznymi warunkami brzegowymi (≥10 000 agentów)
- Strategie binarne: altruista (A) lub egoista (E)
- Sąsiedztwo Moore'a (8 sąsiadów)

**Reguły:**

*Faza 1 - Kalkulacja bogactwa:*

```
W_i = -C_A · s_i + B_A · Σ(s_j)
```

gdzie s_i ∈ {0,1}, Σ(s_j) = suma strategii 8 sąsiadów

*Faza 2 - Uczenie się:* Z prawdopodobieństwem p_copy agent aktualizuje strategię:

- **Tryb deterministyczny** (use_proportional=False): kopiuje strategię najbogatszego sąsiada
- **Tryb proporcjonalny** (use_proportional=True): wybiera strategię proporcjonalnie do bogactwa sąsiadów (softmax-like)

*Faza 3 - Mutacja:* Z prawdopodobieństwem μ (mutation_rate) agent losowo zmienia strategię, co zapobiega zamrożeniu systemu w lokalnych optimach

**Parametry kluczowe:**

- C_A (koszt altruizmu): 0.25-0.60
- B_A (korzyść od altruisty): 0.25-0.50
- p_copy (prawdopodobieństwo kopiowania): 0.01-0.30
- μ (mutation_rate): 0.001-0.010
- use_proportional (tryb wyboru): True/False
- Stosunek B_A/C_A: 0.5-2.0 (>1 → altruizm opłacalny, ≈0.9-1.05 → oscylacje)

## Zjawiska emergentne

1. **Fale strategii** - przestrzenne fale przemieszczające się przez siatkę (niskie p_copy + proporcjonalny wybór)
2. **Klastry kooperacji** - stabilne homogeniczne obszary altruistów chroniących się wzajemnie
3. **Punkty krytyczne** - przy B_A/C_A ≈ 0.875 wrażliwość na warunki początkowe (bifurkacja)
4. **Oscylacje populacyjne** - cykliczne zmiany 40-60% altruistów bez równowagi (wymagają: B_A/C_A ≈ 0.92-1.05, μ > 0.001, use_proportional=True)
5. **Złożone trajektorie** - spiralne orbity i chaotyczne wędrówki środków ciężkości grup

# Mechanizmy wspierające oscylacje

Dla uzyskania cyklicznych fal niezbędne są:

- **Bliska równowaga** (B_A/C_A ≈ 0.92-1.05): żadna strategia nie dominuje absolutnie
- **Stochastyczność** (use_proportional=True lub wyższe μ): zapobiega zamrożeniu w stanie stabilnym
- **Niskie tempo kopiowania** (p_copy = 0.02-0.05): pozwala na rozwinięcie fal przestrzennych
- **Mutacja** (μ = 0.002-0.005): utrzymuje dynamikę nawet gdy system zbliża się do absorpcji
- **Duża przestrzeń** (grid ≥ 120×120, n_steps ≥ 1500): daje czas i miejsce na rozwój cykli

## Wnioski

- **Przestrzeń ratuje kooperację**: altruizm przetrwa nawet gdy B_A < C_A (niemożliwe w modelu dobrze wymieszanym)

- **Emergencja lokalna**: złożone wzorce powstają spontanicznie bez centralnej koordynacji
- **Tempo uczenia się**: p_copy kontroluje przejście wolne fale → chaos
- **Stochastyczność vs. determinizm**: wybór proporcjonalny umożliwia oscylacje, deterministyczny prowadzi do szybkiej stabilizacji
- **Rola mutacji**: niewielki szum (μ ≈ 0.002) kluczowy dla trwałej dynamiki
- **Progi nieliniowe**: małe zmiany parametrów → jakościowe zmiany zachowania

**Zastosowania**: ewolucja kooperacji biologicznej, dyfuzja norm społecznych, ekonomia behawioralna, teoria gier ewolucyjnych.

**Kluczowy wniosek**: lokalność interakcji + społeczne uczenie się + niewielki szum stochastyczny wystarczają do wyjaśnienia trwałości kooperacji i emergencji złożonych dynamik bez zakładania wrodzonego altruizmu.

## Ograniczenia

- Binarna reprezentacja strategii (brak kontinuum)
- Stałe parametry (brak adaptacji C_A, B_A w czasie)
- Regularna siatka (rzeczywiste sieci są skalowo-wolne)
- Brak kar, reputacji, wielopoziomowej selekcji
- Mutacja losowa (w rzeczywistości błędy uczenia się mogą mieć inne rozkłady)
- Model zakłada pełną informację o bogactwie sąsiadów

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from typing import Dict, Tuple, Optional


class SocialStrategyGrid:
    def __init__(
        self,
        size: int,
        initial_altruist_ratio: float,
        altruist_cost: float,
        altruist_benefit: float,
    ):
        self.size = size
        self.altruist_cost = altruist_cost
        self.altruist_benefit = altruist_benefit

        # 0 = egoista, 1 = altruista
        self.strategies = np.random.choice(
            [0, 1],
            size=(size, size),
            p=[1 - initial_altruist_ratio, initial_altruist_ratio],
        )
        self.wealth = np.zeros((size, size))

    def get_neighbors(self, i: int, j: int) -> list:
        neighbors = []
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if di == 0 and dj == 0:
                    continue
                ni, nj = (i + di) % self.size, (j + dj) % self.size
                neighbors.append((ni, nj))
        return neighbors

    def calculate_wealth(self):
        new_wealth = np.zeros((self.size, self.size))

        for i in range(self.size):
            for j in range(self.size):
                neighbors = self.get_neighbors(i, j)

                # Koszt bycia altruistą
                if self.strategies[i, j] == 1:
                    new_wealth[i, j] -= self.altruist_cost

                # Korzyści od altruistycznych sąsiadów
                for ni, nj in neighbors:
                    if self.strategies[ni, nj] == 1:
                        new_wealth[i, j] += self.altruist_benefit

        self.wealth = new_wealth

    def update_strategies(
        self,
        copy_probability: float,
        mutation_rate: float = 0.001,
        use_proportional: bool = False,
    ):
        new_strategies = self.strategies.copy()

        for i in range(self.size):
            for j in range(self.size):
                # Mutacja losowa (zapobiega zamrożeniu)
                if np.random.random() < mutation_rate:
                    new_strategies[i, j] = 1 - self.strategies[i, j]
```

```
                continue

            if np.random.random() < copy_probability:
                neighbors = self.get_neighbors(i, j)

                if use_proportional:
                    # Wybór proporcjonalny do bogactwa (bardziej stochastyczny)
                    wealths = [self.wealth[ni, nj] for ni, nj in neighbors]
                    wealths.append(self.wealth[i, j])

                    # Przeskaluj do prawdopodobieństw (softmax-like)
                    min_w = min(wealths)
                    wealths = [
                        w - min_w + 0.1 for w in wealths
                    ]  # shift to positive
                    total = sum(wealths)
                    probs = [w / total for w in wealths]

                    # Wybierz strategię proporcjonalnie
                    all_neighbors = neighbors + [(i, j)]
                    chosen_idx = np.random.choice(len(all_neighbors), p=probs)
                    ni, nj = all_neighbors[chosen_idx]
                    new_strategies[i, j] = self.strategies[ni, nj]
                else:
                    # Oryginalna metoda - najlepszy sąsiad
                    max_wealth = self.wealth[i, j]
                    best_strategy = self.strategies[i, j]

                    for ni, nj in neighbors:
                        if self.wealth[ni, nj] > max_wealth:
                            max_wealth = self.wealth[ni, nj]
                            best_strategy = self.strategies[ni, nj]

                    new_strategies[i, j] = best_strategy

        self.strategies = new_strategies

    def get_statistics(self) -> Dict:
        n_altruists = np.sum(self.strategies == 1)
        n_egoists = np.sum(self.strategies == 0)

        # Środek ciężkości altruistów
        if n_altruists > 0:
            altruist_positions = np.where(self.strategies == 1)
            altruist_center = (
                np.mean(altruist_positions[0]),
                np.mean(altruist_positions[1]),
            )
        else:
            altruist_center = (np.nan, np.nan)

        # Środek ciężkości egoistów
        if n_egoists > 0:
            egoist_positions = np.where(self.strategies == 0)
            egoist_center = (np.mean(egoist_positions[0]), np.mean(egoist_positions[1]))
        else:
            egoist_center = (np.nan, np.nan)

        return {
            "n_altruists": n_altruists,
            "n_egoists": n_egoists,
            "altruist_center": altruist_center,
            "egoist_center": egoist_center,
        }


def run_simulation(
    grid_size: int = 100,
    altruist_cost: float = 0.4,
    altruist_benefit: float = 0.35,
    copy_probability: float = 0.05,
    initial_altruist_ratio: float = 0.5,
    n_steps: int = 800,
    random_seed: Optional[int] = None,
    verbose: bool = True,
    mutation_rate: float = 0.001,
    use_proportional: bool = False,
) -> Tuple[SocialStrategyGrid, Dict]:
    if random_seed is not None:
        np.random.seed(random_seed)

    if verbose:
        print(f"Inicjalizacja symulacji...")
        print(
            f"  Parametry: cost={altruist_cost}, benefit={altruist_benefit}, copy_prob={copy_probability}"
        )
        print(f"  Mutation rate: {mutation_rate}, Proportional: {use_proportional}")

    # Inicjalizacja
    grid = SocialStrategyGrid(
        grid_size, initial_altruist_ratio, altruist_cost, altruist_benefit
    )
```

```python
    # Przechowywanie historii
    history = {
        "n_altruists": [],
        "n_egoists": [],
        "altruist_center_x": [],
        "altruist_center_y": [],
        "egoist_center_x": [],
        "egoist_center_y": [],
        "params": {
            "grid_size": grid_size,
            "altruist_cost": altruist_cost,
            "altruist_benefit": altruist_benefit,
            "copy_probability": copy_probability,
            "initial_altruist_ratio": initial_altruist_ratio,
            "n_steps": n_steps,
            "mutation_rate": mutation_rate,
            "use_proportional": use_proportional,
        },
    }

    # Symulacja
    if verbose:
        print(f"Rozpoczęcie symulacji ({n_steps} kroków)...")

    for step in range(n_steps):
        if verbose and step % 100 == 0:
            print(f"  Krok {step}/{n_steps}")

        grid.calculate_wealth()
        grid.update_strategies(copy_probability, mutation_rate, use_proportional)

        stats = grid.get_statistics()
        history["n_altruists"].append(stats["n_altruists"])
        history["n_egoists"].append(stats["n_egoists"])
        history["altruist_center_x"].append(stats["altruist_center"][0])
        history["altruist_center_y"].append(stats["altruist_center"][1])
        history["egoist_center_x"].append(stats["egoist_center"][0])
        history["egoist_center_y"].append(stats["egoist_center"][1])

    if verbose:
        print("Symulacja zakończona!")

    # Podsumowanie
    total = grid_size * grid_size
    initial_altruists = history["n_altruists"][0]
    final_altruists = history["n_altruists"][-1]

    print(f"\nWyniki:")
    print(
        f"  Początkowy udział altruistów: {initial_altruists}/{total} ({initial_altruists/total*100:.1f}%)"
    )
    print(
        f"  Końcowy udział altruistów: {final_altruists}/{total} ({final_altruists/total*100:.1f}%)"
    )
    print(f"  Zmiana: {final_altruists - initial_altruists:+d} agentów")

    return grid, history


def plot_results(
    history: Dict,
    grid: Optional[SocialStrategyGrid] = None,
    title_suffix: str = "",
    figsize: Tuple[int, int] = (16, 12),
):
    params = history["params"]
    n_steps = params["n_steps"]
    grid_size = params["grid_size"]

    if grid is not None:
        fig = plt.figure(figsize=figsize)
        n_plots = 6
    else:
        fig = plt.figure(figsize=(figsize[0], figsize[1] * 5 / 6))
        n_plots = 5

    steps = range(n_steps)

    # 1. Liczebność strategii w czasie
    ax1 = plt.subplot(3, 2, 1) if n_plots == 6 else plt.subplot(3, 2, 1)
    ax1.plot(
        steps, history["n_altruists"], label="Altruiści", color="blue", linewidth=2
    )
    ax1.plot(steps, history["n_egoists"], label="Egoiści", color="red", linewidth=2)
    ax1.set_xlabel("Krok symulacji")
    ax1.set_ylabel("Liczba agentów")
    ax1.set_title(f"Dynamika populacji strategii{title_suffix}")
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # 2. Proporcje strategii
```

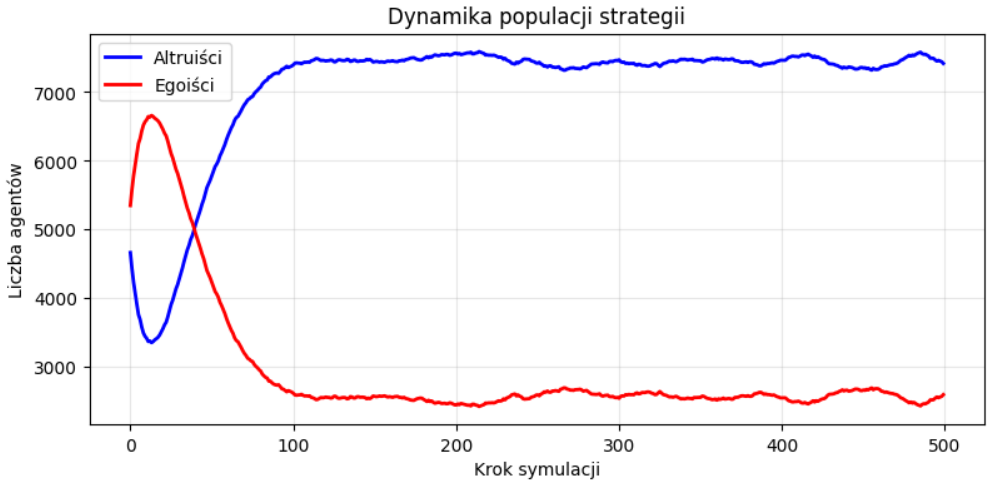
```
    ax2 = plt.subplot(3, 2, 2)
    total = grid_size * grid_size
    altruist_ratio = [n / total * 100 for n in history["n_altruists"]]
    ax2.plot(steps, altruist_ratio, color="purple", linewidth=2)
    ax2.set_xlabel("Krok symulacji")
    ax2.set_ylabel("Procent altruistów (%)")
    ax2.set_title(f"Udział altruistów w populacji{title_suffix}")
    ax2.axhline(y=50, color="gray", linestyle="--", alpha=0.5)
    ax2.grid(True, alpha=0.3)

    # 3. Środek ciężkości - współrzędna X
    ax3 = plt.subplot(3, 2, 3)
    ax3.plot(
        steps,
        history["altruist_center_x"],
        label="Altruiści",
        color="blue",
        linewidth=2,
        alpha=0.7,
    )
    ax3.plot(
        steps,
        history["egoist_center_x"],
        label="Egoiści",
        color="red",
        linewidth=2,
        alpha=0.7,
    )
    ax3.set_xlabel("Krok symulacji")
    ax3.set_ylabel("Współrzędna X")
    ax3.set_title(f"Środek ciężkości - pozycja X{title_suffix}")
    ax3.legend()
    ax3.grid(True, alpha=0.3)

    # 4. Środek ciężkości - współrzędna Y
    ax4 = plt.subplot(3, 2, 4)
    ax4.plot(
        steps,
        history["altruist_center_y"],
        label="Altruiści",
        color="blue",
        linewidth=2,
        alpha=0.7,
    )
    ax4.plot(
        steps,
        history["egoist_center_y"],
        label="Egoiści",
        color="red",
        linewidth=2,
        alpha=0.7,
    )
    ax4.set_xlabel("Krok symulacji")
    ax4.set_ylabel("Współrzędna Y")
    ax4.set_title(f"Środek ciężkości - pozycja Y{title_suffix}")
    ax4.legend()
    ax4.grid(True, alpha=0.3)

    # 5. Trajektoria środków ciężkości w przestrzeni 2D
    ax5 = plt.subplot(3, 2, 5)
    ax5.plot(
        history["altruist_center_x"],
        history["altruist_center_y"],
        label="Altruiści",
        color="blue",
        linewidth=1,
        alpha=0.6,
    )
    ax5.plot(
        history["egoist_center_x"],
        history["egoist_center_y"],
        label="Egoiści",
        color="red",
        linewidth=1,
        alpha=0.6,
    )
    ax5.scatter(
        history["altruist_center_x"][0],
        history["altruist_center_y"][0],
        color="blue",
        s=100,
        marker="o",
        label="Start altruistów",
        zorder=5,
    )
    ax5.scatter(
        history["egoist_center_x"][0],
        history["egoist_center_y"][0],
        color="red",
        s=100,
        marker="o",
        label="Start egoistów",
```

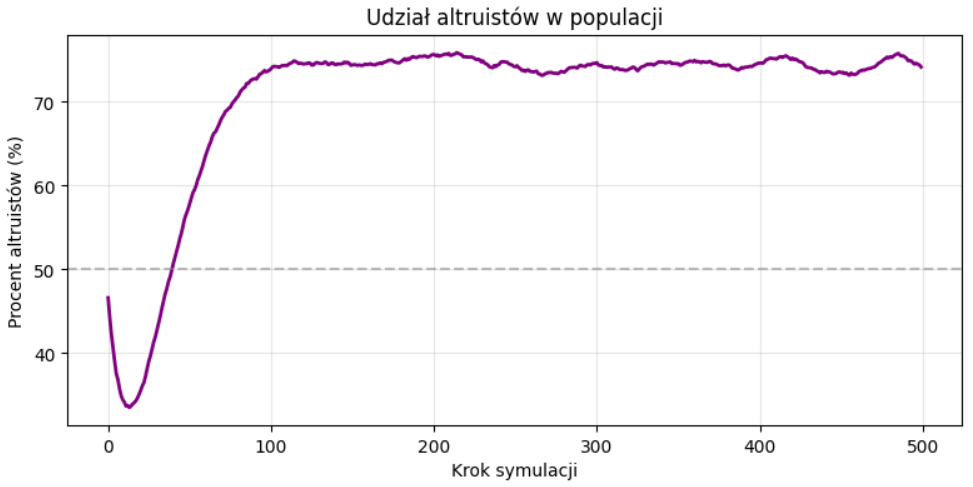
```
        zorder=5,
    )
    ax5.scatter(
        history["altruist_center_x"][-1],
        history["altruist_center_y"][-1],
        color="blue",
        s=100,
        marker="s",
        label="Koniec altruistów",
        zorder=5,
    )
    ax5.scatter(
        history["egoist_center_x"][-1],
        history["egoist_center_y"][-1],
        color="red",
        s=100,
        marker="s",
        label="Koniec egoistów",
        zorder=5,
    )
    ax5.set_xlabel("Pozycja X")
    ax5.set_ylabel("Pozycja Y")
    ax5.set_title(f"Trajektoria środków ciężkości{title_suffix}")
    ax5.legend(fontsize=8)
    ax5.grid(True, alpha=0.3)
    ax5.set_xlim(0, grid_size)
    ax5.set_ylim(0, grid_size)

    # 6. Końcowy stan siatki (jeśli dostępny)
    if grid is not None:
        ax6 = plt.subplot(3, 2, 6)
        from matplotlib.colors import ListedColormap

        cmap = ListedColormap(["red", "blue"])
        im = ax6.imshow(grid.strategies, cmap=cmap, interpolation="nearest")
        ax6.set_title(f"Końcowy stan siatki (krok {n_steps}){title_suffix}")
        ax6.set_xlabel("X")
        ax6.set_ylabel("Y")
        cbar = plt.colorbar(im, ax=ax6, ticks=[0, 1])
        cbar.set_ticklabels(["Egoista", "Altruista"])

    plt.tight_layout()
    return fig
```

## ALTRUISTYCZNY RAJ - Altruizm się opłaca i dominuje

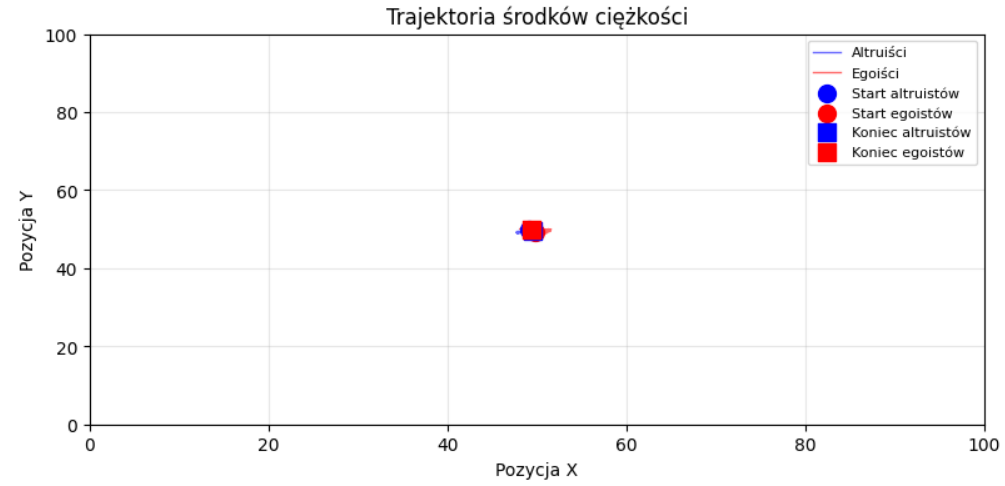
In [16]:
```
grid, history = run_simulation(
    grid_size=100,
    altruist_cost=0.25,
    altruist_benefit=0.40,
    copy_probability=0.08,
    mutation_rate=0.002,
    use_proportional=False,
    n_steps=500,
    random_seed=42,
)
plot_results(history, grid)
plt.show()
```

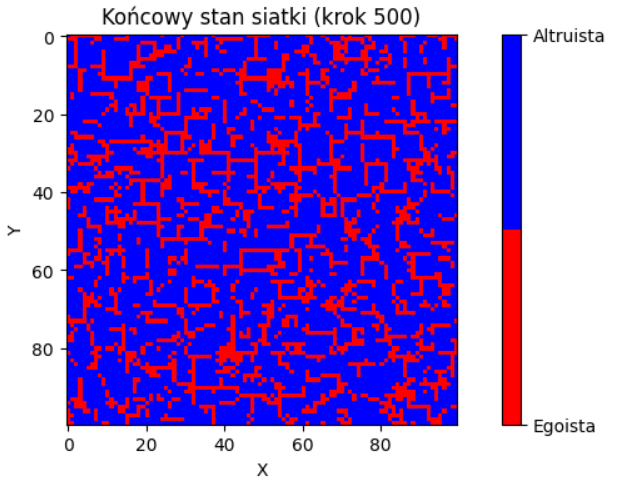
```
Inicjalizacja symulacji...
  Parametry: cost=0.25, benefit=0.4, copy_prob=0.08
  Mutation rate: 0.002, Proportional: False
Rozpoczęcie symulacji (500 kroków)...
  Krok 0/500
  Krok 100/500
  Krok 200/500
  Krok 300/500
  Krok 400/500
Symulacja zakończona!

Wyniki:
  Początkowy udział altruistów: 4657/10000 (46.6%)
  Końcowy udział altruistów: 7411/10000 (74.1%)
  Zmiana: +2754 agentów
```





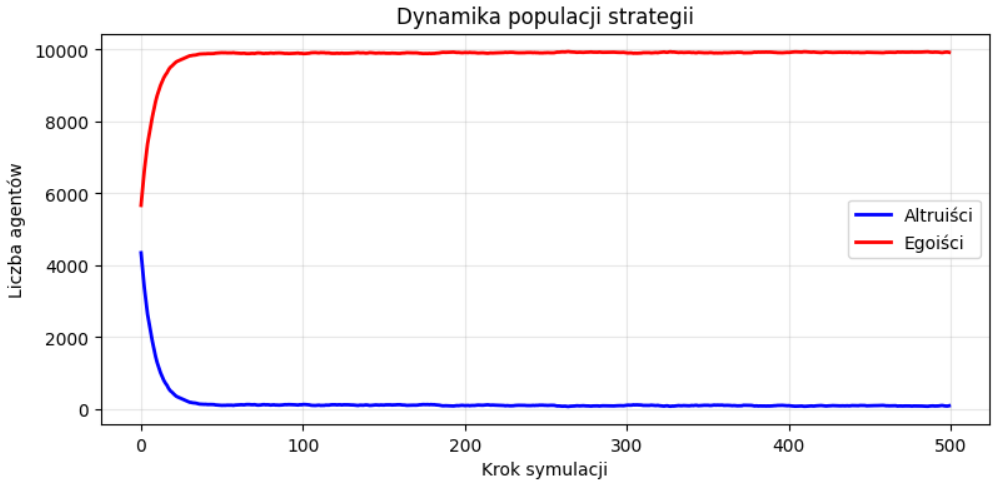



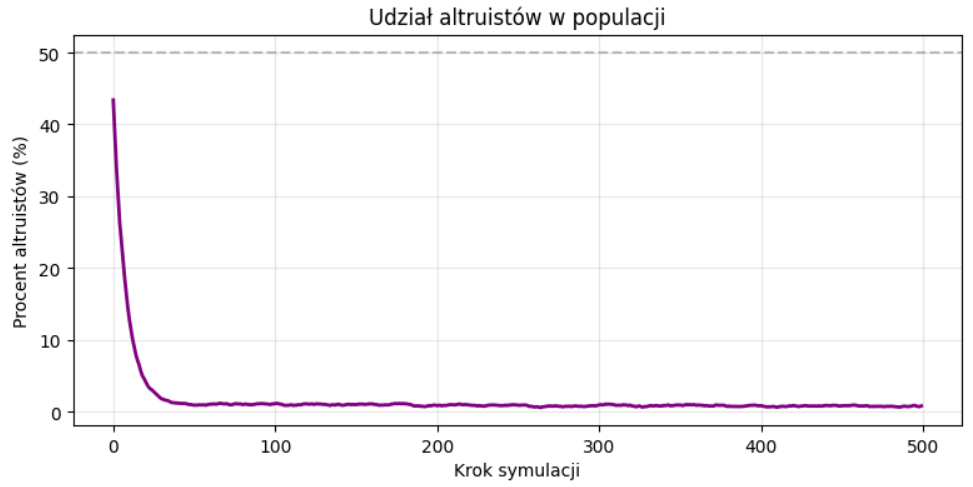




## EGOISTYCZNA PUSTKA - Egoizm triumfuje, współpraca zanika

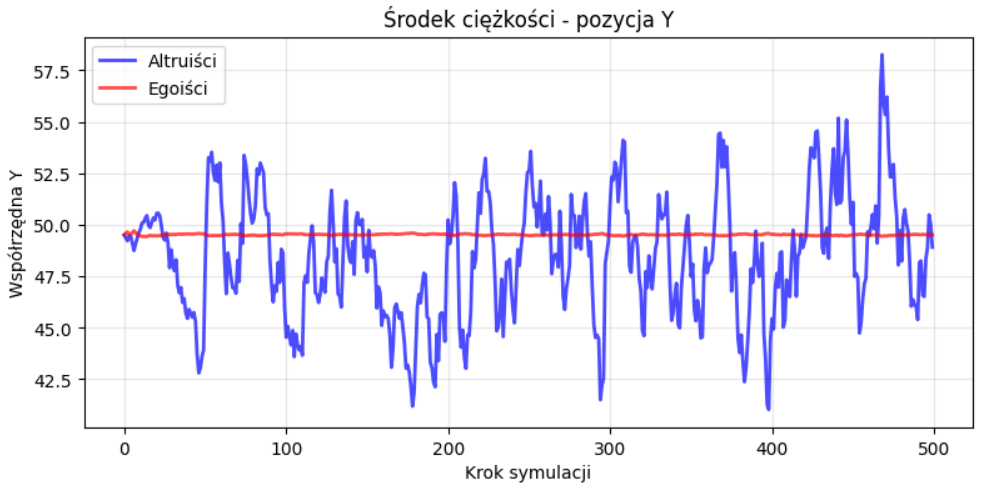
In [15]:
```
grid, history = run_simulation(
    grid_size=100,
    altruist_cost=0.6,
    altruist_benefit=0.25,
    copy_probability=0.12,
    mutation_rate=0.001,
    use_proportional=False,
    n_steps=500,
    random_seed=42,
)
plot_results(history, grid)
plt.show()
```

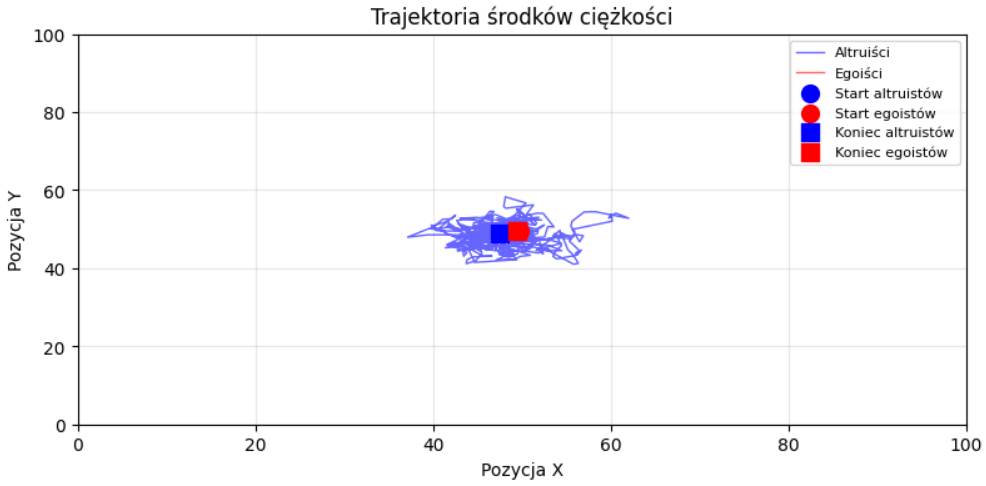
```
Inicjalizacja symulacji...
  Parametry: cost=0.6, benefit=0.25, copy_prob=0.12
  Mutation rate: 0.001, Proportional: False
Rozpoczęcie symulacji (500 kroków)...
  Krok 0/500
  Krok 100/500
  Krok 200/500
  Krok 300/500
  Krok 400/500
Symulacja zakończona!

Wyniki:
  Początkowy udział altruistów: 4340/10000 (43.4%)
  Końcowy udział altruistów: 81/10000 (0.8%)
  Zmiana: -4259 agentów
```

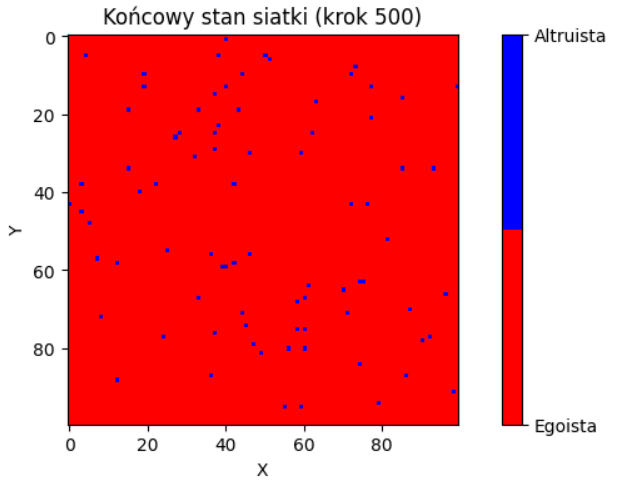






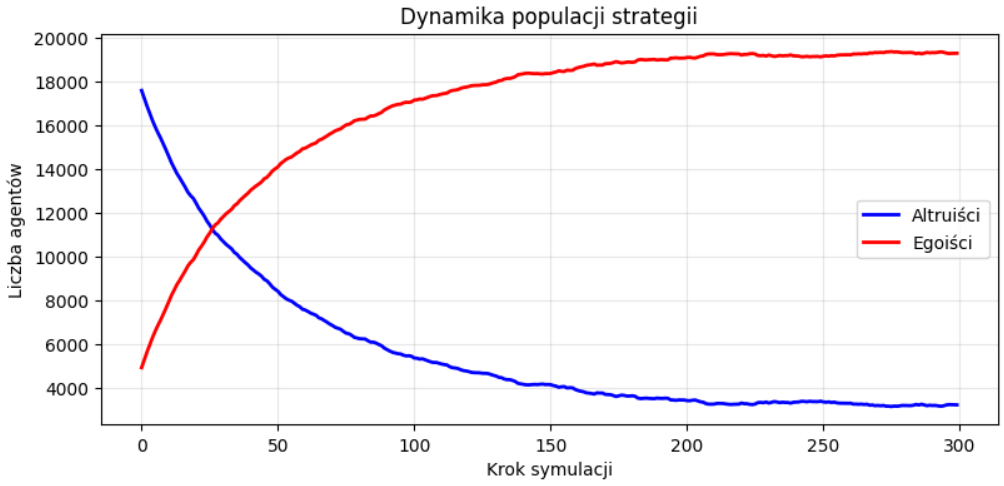



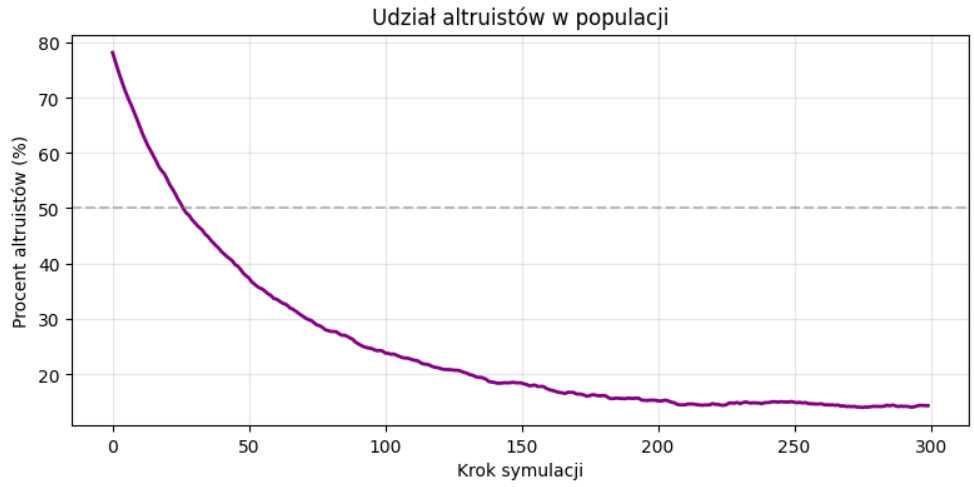




ZMIANA DOMINACJI - altruiści pomimo początkowej większej liczbności, tracą swoją przewage

In [25]:
```
grid, history = run_simulation(
    grid_size=150,
    altruist_cost=0.40,
    altruist_benefit=0.37,
    copy_probability=0.10,
    mutation_rate=0.008,
    use_proportional=True,
    initial_altruist_ratio=0.8,
    n_steps=300,
    random_seed=42,
)
plot_results(history, grid)
plt.show()
```

```
Inicjalizacja symulacji...
  Parametry: cost=0.4, benefit=0.37, copy_prob=0.1
  Mutation rate: 0.008, Proportional: True
Rozpoczęcie symulacji (300 kroków)...
  Krok 0/300
  Krok 100/300
  Krok 200/300
Symulacja zakończona!

Wyniki:
  Początkowy udział altruistów: 17587/22500 (78.2%)
  Końcowy udział altruistów: 3210/22500 (14.3%)
  Zmiana: -14377 agentów
```

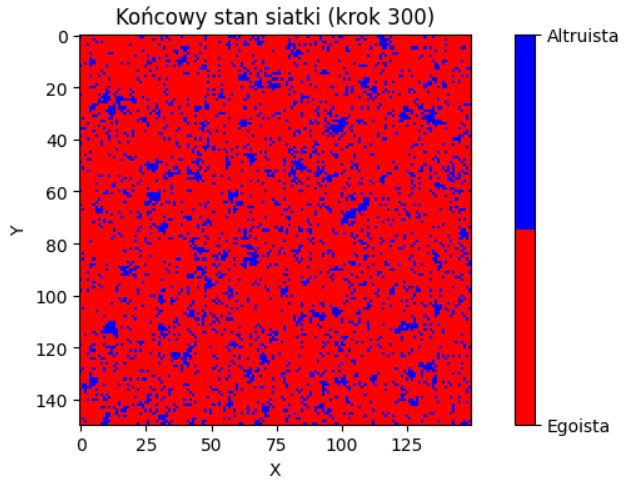

Dynamika populacji strategii
Liczba agentów
Krok symulacji
Altruiści
Egoiści


Udział altruistów w populacji
Procent altruistów (%)
Krok symulacji


Środek ciężkości - pozycja X
Współrzędna X
Krok symulacji
Altruiści
Egoiści


Środek ciężkości - pozycja Y
Współrzędna Y
Krok symulacji
Altruiści
Egoiści


Trajektoria środków ciężkości
Pozycja Y
Pozycja X
Altruiści
Egoiści
Start altruistów
Start egoistów
Koniec altruistów
Koniec egoistów


Końcowy stan siatki (krok 300)
Y
X
Altruista
Egoista