Uniwersytet Przyrodniczy we Wrocławiu
Wydział Biologii i Hodowli Zwierząt
Kierunek: Bioinformatyka
Studia stacjonarne pierwszego stopnia

Damian Tułacz
112156

Konfrontacja modeli równowagi środowiskowej typu automat komórkowy z modelami typu Lotki-Volterry

Comparison of ecological balance cellular automaton models with Lotka-Volterra models

Praca wykonana pod kierunkiem
Dr. Jana Jełowickiego
Katedra Matematyki

Wrocław 2020

Oświadczenie opiekuna pracy

Oświadczam, że niniejsza praca została przygotowana pod moim kierownictwem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data ..... Podpis opiekuna pracy .....

Oświadczenie autora pracy

Oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami ani też nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Data ..... Podpis autora pracy .....

Streszczenie

Konfrontacja modeli równowagi środowiskowej typu automat komórkowy z modelami typu Lotki-Volterry

Tematem niniejszej pracy jest zbadanie, czy model równowagi środowiskowej typu drapieżnik-ofiara, inspirowany modelem Lotki-Volterry, działający na zasadzie automatu komórkowego, będzie zachowywał się inaczej, jeśli zostanie zastosowany terytorializm u drapieżników, w porównaniu do modelu bez terytorializmu. Pierwsza część stanowi wprowadzenie, w którym przedstawiono cel pracy. W drugiej części opisano zagadnienia teoretyczne związane z poruszanym tematem. Zdefiniowano pojęcia automatu komórkowego, modelu Lotki-Volterry oraz terytorializmu. Trzecia część została poświęcona modelowi typu automat komórkowy, który został stworzony na potrzeby problemu. Opisano zasadę jego działania, sposób jego implementacji w języku Python, skonfrontowano go z modelem Lotki-Volterry oraz zwrócono uwagę na jego wady i ograniczenia. W czwartej części przeprowadzono analizę danych, dotyczących liczebności osobników w poszczególnych momentach trwania symulacji. Wywnioskowano, że mechanizm terytorializmu istotnie wpłynął na liczebności osobników.

Słowa kluczowe: drapieżnik, ofiara, model, automat komórkowy, terytorializm

Abstract

Comparison of ecological balance cellular automaton models with Lotka-Volterra models

The subject of this project is to examine whether the ecological balance predator-prey model, inspired by the Lotka-Volterra model, operating on the principle of cellular automaton, will behave differently, if territorialism is applied to predators, compared to the model without territorialism. The first part is an introduction, which presents the aim of the project. The second part describes the theoretical issues related to the discussed topic, such as cellular automaton, Lotka Volterra model and territorialsm. The third part contains information about cellular automaton model, which was created for this project. The principles of its operation was described as well as the way, it was implemented in Python language. It was confrontated with the Lotka-Volterra model and attention was paid to its drawbacks and limitations. In the fourth part, the analysis of data containing the number of individuals in particular moments of the simulation was conducted. It was concluded that the mechanism of territorialism had significant impact on the number of individuals.

Keywords: predator, prey, model, cellular automaton, territorialism

Spis treści

1 Wprowadzenie

Celem pracy było stworzenie symulacji współistnienia drapieżników i ofiar w środowisku, inspirowanej modelem Lotki-Volterry (opisany w [2] i [8], oraz w podsekcji 2.2), oraz zbadanie, czy mechanizm terytorializmu (opisany w [4] oraz w podsekcji 2.3.2) wpływa na zachowanie stworzonego modelu. Symulacja ma formę automatu komórkowego.

Teza: Mechanizm terytorializmu opisany w [4] ma wpływ na zachowanie stworzonego modelu.

2 Wstęp teoretyczny

2.1 Automat komórkowy

Automaty komórkowe to dyskretne, abstrakcyjne systemy obliczeniowe, które okazały się przydatne w różnych dziedzinach nauki. Można w nich zaobserwować złożone zachowania, dotyczące prostych komórek zachowujących się zgodnie z prostymi zasadami.

Automaty komórkowe zazwyczaj są dyskretne czasowo i przestrzennie. Składają się ze skończonego lub policzalnego zbioru homogenicznych, prostych jednostek, atomów lub komórek. W każdej jednostce czasu, każda komórka przyjmuje jeden ze stanów. Liczba możliwych do przyjęcia stanów jest skończona. Komórki zmieniają swój stan równolegle w dyskretnych jednostkach czasu, zgodnie z zasadami panującymi w danym automacie. Aktualizacja stanów komórek bierze pod uwagę stany komórek znajdujących się w sąsiedztwie. Gdy komórki są oddalone, nie wpływają na siebie.

Automaty komórkowe są abstrakcyjne. Oznacza to, że mogą być określone w sposób czysto matematyczny, a struktury fizyczne mogą je realizować.

Automaty komórkowe są systemami obliczeniowymi. Mogą obliczać funkcje oraz rozwiązywać problemy algorytmiczne. Automat komórkowy z odpowiednimi regułami może naśladować klasyczną maszynę Turinga, a zatem może obliczać, biorąc pod uwagę hipotezę Churcha-Turinga, wszystko, co można obliczyć.

Struktura automatu komórkowego zawiera 4 parametry:

- Dyskretna n-wymiarowa siatka składająca się z komórek. Komórki mogą mieć dowolny kształt. Homogeniczność jest definiowana jako jakościowa identyczność wszystkich komórek.
- Dyskretne stany. W każdym dyskretnym kroku czasowym, każda komórka posiada jeden określony stan. Liczba możliwych stanów jest skończona.

- Lokalne interakcje komórek. Zachowanie każdej komórki zależy od stanów komórek znajdujących się w jej sąsiedztwie. Sąsiedztwo może być zdefiniowane w różny sposób dla różnych automatów komórkowych.

- Dyskretna dynamika. W każdym kroku czasowym, każda komórka aktualizuje swój aktualny stan zgodnie z deterministyczną funkcją przejścia. Zazwyczaj, choć nie zawsze, zakłada się również, że aktualizacja jest synchroniczna oraz przyjmuje jako informacja wejściowa w kroku czasowym t stany komórek w sąsiedztwie w bezpośrednio poprzedzającym kroku czasowym $t - 1$

Przykładami znanych automatów komórkowych są:

- Zestaw elementarnych automatów komórkowych opisany przez Stephena Wolframa [5]. Charakteryzują się tym, że są jednowymiarowe, a komórka może zawierać jeden z dwóch stanów.

- „Gra w życie" [6] - Dwuwymiarowy automat komórkowy, w którym każda komórka może zawierać jeden z dwóch stanów. Reguły są następujące:

  – Komórka ze stanem '0', która ma dokładnie 3 sąsiadów ze stanem '1', w następnej jednostce czasu przyjmuje stan '1'

  – Komórka ze stanem '1', która ma 2, lub 3 komórki ze stanem '1' w sąsiedztwie, nie zmienia stanu. Przy innej liczbie sąsiadów ze stanem '1' przyjmie stan '0'.

2.2 Model Lotki-Volterry

Model Lotki-Volterry [2] [8] powstał w latach 20, XX wieku i jest najstarszym modelem opisującym współzależność dwóch gatunków w przyrodzie. Model ten zakłada, że osobniki jednego gatunku (drapieżniki) polują na osobniki drugiego gatunku (ofiary). Powstał on w celu wyjaśnienia obserwacji, wedle których po zakończeniu I wojny światowej nastąpiła dysproporcja w zmianach zagęszczeń ryb drapieżnych oraz ryb stanowiących ich pożywienie. Dane empiryczne nie zgadzały się z założeniem, że ograniczenie połowów wywołane działaniami wojennymi powinno skutkować wzrostem liczebności obu gatunków ryb.

Cechą charakterystyczną modelu jest występowanie w nim cykliczności. Klasycznym, empirycznym przykładem są dane pochodzące z firmy Hudson's Bay Company, zebrane w drugiej połowie XIX wieku. Dane przedstawiają liczebność rysi i zajęcy na podstawie liczby skupowanych skór tych zwierząt. Można w nich zaobserwować powtarzalność w liczebności tych zwierząt co pewien okres.

Założenia potrzebne do skonstruowania modelu:

1. W otoczeniu występują tylko 2 gatunki (drapieżniki i ofiary), lub dopuszczalne jest, aby pominąć występowanie innych gatunków ze względu na ich znikomą liczebność.

2. Gdy w środowisku występują tylko ofiary, to pod nieobecność drapieżników nic ich nie ogranicza, przez co obserwuje się ich wzrost wykładniczy.

3. Gdy w środowisku występują jedynie drapieżniki, wówczas następuje ich spadek wykładniczy, gdyż kiedy nie ma ofiar, drapieżniki nie mają co jeść, co skutkuje brakiem energii do reprodukcji.

4. Gdy oba gatunki występują w środowisku, wówczas liczebność ofiar spada z uwagi na to, że są zjadane przez drapieżniki, które dzięki pożywieniu mają energię do reprodukcji, więc ich liczebność wzrasta. W miarę zmniejszania się liczby ofiar, liczba drapieżników również spada, gdyż nie mają wystarczającej ilości pożywienia. Gdy liczebność drapieżników spada, liczba ofiar rośnie, gdyż zagrożenie ze strony drapieżników pomniejsza się.

5. Model uwzględnia średnie zagęszczenia drapieżników i ofiar w otoczeniu. Nie bierze pod uwagę ich rozmieszczenia przestrzennego.

Przy tych założeniach dostajemy następujący układ równań różniczkowych:

$$\begin{cases} \frac{dV}{dt} = rV - aVP \\ \frac{dP}{dt} = baVP - sP, \end{cases} \tag{1}$$

gdzie:

- $V(t)$ - liczebność ofiar w chwili t
- $P(t)$ - liczebność drapieżników w chwili t
- r - współczynnik przyrostu ofiar
- a - współczynnik efektywności polowań
- b - współczynnik przyrostu drapieżników
- s - współczynnik umieralności drapieżników

Wszystkie rozwiązania powyższego układu równań są okresowe z wyjątkiem jednowymiarowych funkcji wykładniczych.

2.3 Terytorializm

2.3.1 Terytorializm w przyrodzie

Terytorializm jest charakterystycznym rodzajem zachowania zwierząt obejmującym większość gatunków ssaków z rzędu drapieżne (*Carnivora*) [3]. Tylko u nielicznych gatunków tego rzędu nie stwierdzono terytorializmu. Zjawisko to występuje również m.in. u wielu drapieżnych ptaków, ryb czy jaszczurek. Cechuje się tym, że zwierzęta są związane ze swoim terytorium, którego bronią przed innymi osobnikami, najczęściej tego samego gatunku. Wynikiem tego jest zajmowanie przez osobniki, bądź grupy osobników niezależnych obszarów. Najważniejsze funkcje terytorializmu u ssaków drapieżnych to:

- stały dostęp do pokarmu dla osobnika, bądź grupy,
- zapewnienie schronień dla właścicieli terytorium oraz ich młodych,
- utworzenie więzi pomiędzy osobnikami.

Obrona terytorium może mieć miejsce na wiele sposobów. Jednym z nich może być czynne przepędzanie intruza lub przyjmowanie postawy sygnalizującej prawo do zajmowanego obszaru. Do prawdziwych walk dochodzi stosunkowo rzadko. Najistotniejszym rodzajem ochrony wydaje się być znakowanie terenu wydzielinami. Ssaki drapieżne mają dobre warunki do odbierania takich znaków ze względu na dobrze rozwinięty węch. Tego typu znaki niosą dla innych osobników informację o wielu cechach zwierzęcia takich jak np. wiek czy płeć. Wywołuje to odpowiednią reakcję u osobnika odczytującego taki znak. Osobniki do znakowania terenu wybierają miejsca, na których zapach będzie możliwie najlepiej rozprzestrzeniony, a ślad będzie odporny na zniszczenie i trwały przez jak najdłuższy czas.

2.3.2 Modelowanie terytorializmu

Artykuł [4] przedstawia dwa podejścia do modelowania terytorializmu. Pierwsze zakłada, że nie jest znane dokładne położenie osobnika i wykorzystuje funkcję gęstości prawdopodobieństwa. Drugie podejście dotyczy pojedynczych osobników. Model opisany w rozdziale 3 korzysta z podejścia indywidualnego.

Modele z tzw. „terytorialnymi losowymi wędrówkami" działają w taki sposób, że zwierzęta poruszają się po dyskretnej siatce, a każde z nich pozostawia po sobie ślad po przemieszczeniu. Ślad pozostaje trwały przez określoną ilość czasu, po czym wygasa i inne osobniki przestają na niego reagować. Zwierzęta mogą przemieszczać się na każde sąsiednie miejsce siatki, za wyjątkiem miejsc, które zawierają ślady innych osobników. Terytorium jest wtedy zdefiniowane jako zbiór wszystkich aktywnych śladów osobnika. Zaletą tego podejścia jest

to, że mamy w nim do czynienia z terytorium tworzącym się naturalnie w każdym momencie. Terytoria nie są wtedy statyczne, ale zmieniają się powoli w czasie.

Zaletą modelu z podejściem indywidualnym jest to, że wyjaśnia zjawisko przesuwania się terytorium, które zostało zaobserwowane u gatunków pochodzących z różnych taksonów. Jego wada polega na tym, że jest wymagający obliczeniowo.

Przykładem gatunku zachowującego się w podobny sposób jest lis rudy (*Vulpes vulpes*) [11].

3 Model

3.1 Opis modelu

Model ma za zadanie symulować współistnienie dwóch gatunków (drapieżniki i ofiary) w środowisku. Działa na zasadzie automatu komórkowego, z dyskretną siatką dwuwymiarową oraz dyskretnym czasem. W każdej komórce może znajdować się jeden drapieżnik, jedna ofiara, lub komórka może być pusta. Drapieżniki poruszają się, zmieniając swoje położenie. Ich zasięg wzroku, jest kołem o stałym promieniu. Jeśli w zasięgu wzroku drapieżnika nie ma żadnej ofiary, lub drapieżnik nie osiągnął jeszcze wystarczającej wartości głodu, aby szukać pokarmu, wtedy osobnik przemieści się losowo lub pozostanie na swoim miejscu. Jeśli poziom głodu osiągnął odpowiednią wartość, a ofiara znajduje się w zasięgu, drapieżnik przemieści się w jej stronę. Jeśli ofiara znajdzie się wtedy dostatecznie blisko, zostaje zjedzona przez drapieżnika i znika. Stan głodu drapieżnika zostaje wtedy wyzerowany. Gdy stan głodu drapieżnika osiągnie wartość krytyczną (czyli przez określoną liczbę kroków od ostatniego zjedzenia ofiary lub od urodzenia, osobnik nie natrafi na ofiarę), drapieżnik ginie. Ofiary poruszają się w sposób losowy. Dodatkowo, gdy wiek osobnika jest podzielny przez współczynnik określający częstość rodzenia nowych zwierząt, na jego poprzednim miejscu pojawi się nowy osobnik (ale tylko, jeśli w danym kroku czasowym osobnik się przemieści, inaczej nowy osobnik nie może powstać, gdyż musiały by okupować to samo miejsce). Współczynnik ten może być inny dla drapieżników i ofiar. Wiek osobników generowanych w stanie początkowym jest losowy. Osobnik nie przemieści się, jeśli miejsce, na które chce przejść, jest już zajęte przez innego osobnika. Warunki brzegowe są przenikające, co oznacza, że skrajne miejsca siatki graniczą ze sobą, a osobnik po przejściu przez krawędź pojawi się z drugiej strony.

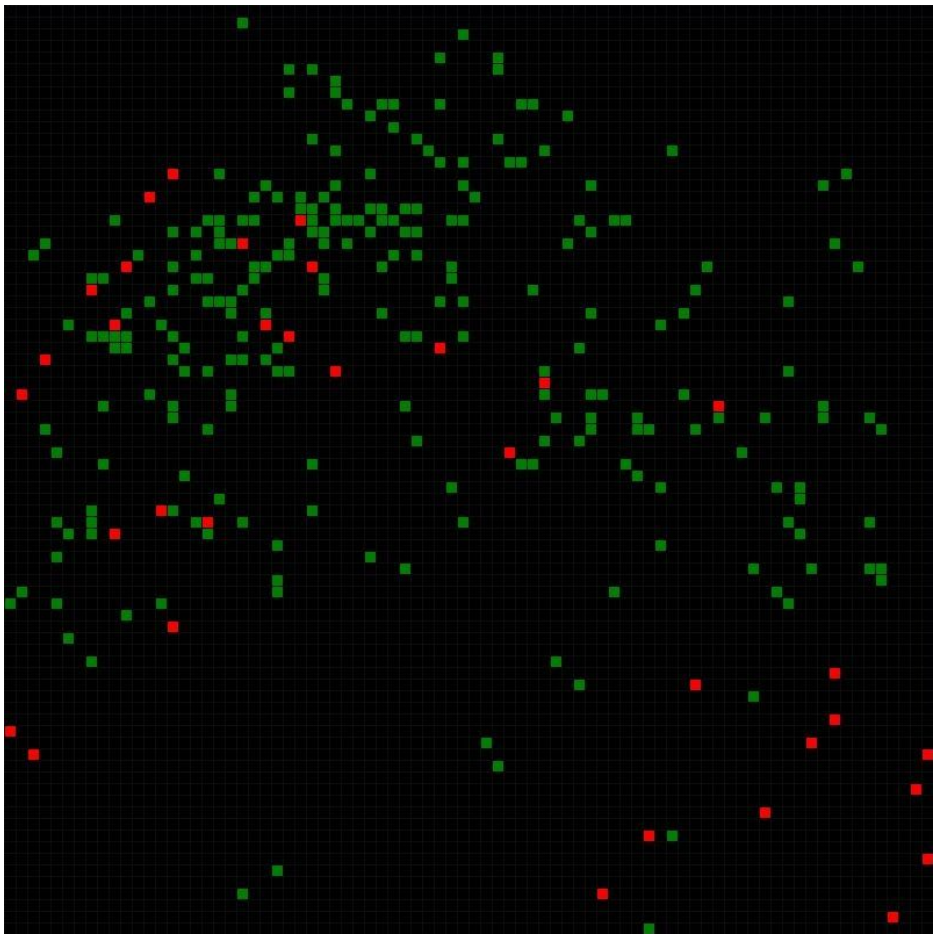
Po wywołaniu symulacji na określony czas, są 3 możliwe scenariusze:

- Drapieżniki i ofiary żyją równocześnie w równowadze dynamicznej, z quasi-cyklicznymi wahaniami liczebności obu populacji,
- Całkowite wyginięcie drapieżników, wtedy ofiary rozmnażają się bez ograniczeń ze strony drapieżników,
- Całkowite wyginięcie ofiar, co powoduje również wyginięciem drapieżników.

3.2 Opis programu

Program stanowi implementację modelu opisanego w podrozdziale 3.1 w języku programowania *Python* [9]. Oblicza on położenie każdego osobnika w aktualnym kroku czasowym, a następnie wyświetla informację o ich położeniu na siatce dwuwymiarowej w postaci kwadratów o odpowiednich kolorach. Pozwala też na zapisanie do pliku informacji o liczebności zwierząt w poszczególnych momentach. Plik z programem został załączony do pracy.

Rysunek 1: Przykładowy widok interfejsu programu



Rysunek 2: Przykładowy widok pliku z liczebnościami osobników, wygenerowanego przez program, z początkowymi 10 wierszami.

```
index;predators;preys
1;40;368
2;47;384
3;48;393
4;51;401
5;52;396
6;58;392
7;61;419
8;63;427
9;69;442
```

3.2.1 Używane moduły

W programie użyto zewnętrznych modułów i funkcji dla języka *Python*:

- **tkinter** - moduł graficznego interfejsu,
- **numpy** - moduł ułatwiający operacje na wektorach i macierzach,
- **time.sleep** - funkcja pozwalająca wprowadzać kontrolowaną przerwę w wykonywaniu instrukcji kodu,
- **random.shuffle** - funkcja pozwalająca ustawiać ciąg w sposób losowy,
- **random.randint** - funkcja pozwalająca generować całkowite liczby pseudolosowe na określonym, przedziale,
- **random.choice** - funkcja pozwalająca wybrać losowo element z listy.

3.2.2 Parametry

Lista parametrów, za pomocą których można dostosować działanie modelu (kod 1).

no_of_generations - liczba kroków czasowych, jakie ma trwać symulacja
report_time - liczba kroków czasowych, na które przypada zapisanie liczby zwierząt do pliku
speed - tyle razy na sekundę jest wyświetlane nowe położenie osobników
square_size - wielkość pojedynczego kwadratu (w pikselach)
field_size - długość i szerokość siatki (jako liczba komórek)
predator_vision - zasięg widzenia drapieżników (promień koła)

hunger_hunt - liczba kroków czasowych, po których drapieżnik zaczyna zwracać uwagę na ofiary
hunger_death - liczba kroków czasowych jaką drapieżnik może przeżyć bez jedzenia
preys_born - odstęp czasowy, z jakim rodzą się nowe ofiary
predators_born - odstęp czasowy, z jakim rodzą się nowe drapieżniki
max_traces - tyle kroków czasowych ostatnich położeń drapieżników jest zapamiętywane
preys_percent - określenie, ile procent dostępnych miejsc mają zajmować ofiary na początku
predators_percent - określenie, ile procent dostępnych miejsc mają zajmować drapieżniki na początku

Kod 1: Przykładowe parametry

```
no_of_generations = 2000
report_time = 5
speed = 1000
square_size = 10
field_size = 80
predator_vision = 4
hunger_hunt = 4
hunger_death = 8
preys_born = 31
predators_born = 13
max_traces = 20
preys_percent = 5
predators_percent = 0.5
```

3.2.3 Obliczenie początkowej liczby drapieżników i ofiar

Następuje sprawdzenie, czy osobniki w stanie początkowym nie zajmują więcej, niż 100% komórek automatu. Następnie ma miejsce obliczenie liczby osobników, jakie mają być stworzone na początku symulacji (kod 2).

Kod 2: Obliczenie początkowej liczby drapieżników i ofiar

```
assert preys_percent + predators_percent <= 100,
  'Początkowy procent drapieżników i ofiar nie może przekraczać 100'
area = field_size ** 2
preys_no = int(area * preys_percent / 100)
predators_no = int(area * predators_percent / 100)
```

3.2.4 Utworzenie tła dla siatki

W tym fragmencie kodu (kod 3) zostaje utworzony obiekt klasy 'Tk', ustalający główne okno programu. Następnie w tym oknie wprowadza się tło dla siatki. Szerokość i wysokość tła (w pikselach) zależy od parametrów: 'field_size' oraz 'square_size'. Szerokość zawsze równa się wysokości, więc siatka jest kwadratowa. Zostają również stworzone 3 macierze. Macierz 'canvas_squares' będzie zawierać informację o tym, jaki kwadrat jest narysowany na siatce w danym położeniu i w danym kroku czasowym. Macierz 'preys_cells' będzie zawierać informację o tym, czy jakaś ofiara znajduje się w danym kroku czasowym i w danym położeniu na siatce. Analogiczną informację o drapieżnikach będzie zawierać macierz 'predators_cells'. Na początku macierze przyjmują na wszystkich położeniach wartość '0'.

Kod 3: Utworzenie tła dla siatki, utworzenie macierzy zawierającej rysowane na siatce kwadraty, utworzenie macierzy zawierających informacje o osobnikach na siatce

```
master = Tk()
background = Canvas(master, background='grey',
                    width=square_size*field_size,
                    height=square_size*field_size,
                    highlightthickness=0)
background.pack()
canvas_squares = np.zeros((field_size, field_size), dtype=int)
preys_cells = np.zeros((field_size, field_size), dtype=bool)
predators_cells = np.zeros((field_size, field_size), dtype=bool)
```

3.2.5 Utworzenie klas ’drapieżnik’ i ’ofiara’

Zostają utworzone dwie klasy: klasa Predator (drapieżnik) i klasa Prey (ofiara) (kod 4). Klasa ofiary przyjmuje atrybut ’index’, oznaczający aktualne położenie na siatce, atrybut ’prev_index’ oznaczający położenie na siatce w poprzednim kroku czasowym oraz atrybut ’age’, który powiększa się o 1 po każdym przeżytym kroku czasowym. Klasa drapieżnika, prócz atrybutów ofiary, posiada dwa inne atrybuty. Pierwszym z nich jest atrybut ’hunger’, oznaczający głód drapieżnika. Gdy drapieżnik w danym kroku czasowym nie zje ofiary, wtedy ten atrybut wzrośnie o 1. Gdy ten atrybut u danego osobnika osiągnie wartość graniczną, osobnik umiera. Drugim atrybutem jest atrybut ’traces’ oznaczający ślady, czyli położenia drapieżnika w określonej liczbie poprzednich kroków czasowych.

Kod 4: Utworzenie klas ’drapieżnik’ i ’ofiara’

```
class Predator:
    def __init__(self, predator_index, age=1):
        self.index = predator_index
        self.prev_index = ()
        self.hunger = 0
        self.traces = []
        self.age = age

class Prey:
    def __init__(self, prey_index, age=1):
        self.index = prey_index
        self.prev_index = ()
        self.age = age
```

3.2.6 Funkcja tworząca początkowe obiekty

Funkcja (kod 5) tworzy początkowe obiekty klas drapieżników i ofiar. Najpierw zostają utworzone puste listy, które będą zawierać obiekty. Powstaje zbiór wszystkich możliwych komórek. Za każdym razem, kiedy wytwarzany jest nowy osobnik, losowana jest jedna z nich, w jej miejscu umieszczany jest osobnik, a następnie pozycja jest usuwana ze zbioru możliwych komórek. Funkcje tworzą zadaną liczbę osobników, o losowych położeniach, a ich obiekty zostają umieszczone na odpowiednich listach. Funkcja zwraca listę ofiar oraz liczbę drapieżników.

Kod 5: Utworzenie początkowych obiektów klasy 'drapieżnik' i klasy 'ofiara'

```
def init_orgs():
    init_preys = []
    init_predators = []
    possible_cells = set()
    for a in range(field_size):
        for b in range(field_size):
            possible_cells.add((a, b))
    for i in range(preys_no):
        xy = choice(tuple(possible_cells))
        prey = Prey(xy, randint(1, preys_born))
        init_preys.append(prey)
        preys_cells[xy] = True
        possible_cells.remove(xy)
    for i in range(predators_no):
        xy = choice(tuple(possible_cells))
        predator = Predator(xy, randint(1, predators_born))
        init_predators.append(predator)
        predators_cells[xy] = True
        possible_cells.remove(xy)
    return init_preys, init_predators
```

3.2.7 Funkcja wyświetlająca kwadrat w danym miejscu siatki

Funkcja (kod 6) najpierw usuwa z danego miejsca siatki aktualnie narysowany element, a następnie rysuje w tym miejscu kwadrat o odpowiednim kolorze oraz wymiarach (w pikselach).

Kod 6: Funkcja wyświetlająca kwadrat w danym miejscu siatki

```
def draw_square(xy):
    background.delete(canvas_squares[xy])
    if preys_cells[xy]:
        color = 'green'
    elif predators_cells[xy]:
        color = 'red'
```

```
    else:
        color = 'black'
    y_square = xy[1] * square_size
    x_square = xy[0] * square_size
    canvas_squares[xy] = background.create_rectangle(
        y_square, x_square,
        y_square + square_size,
        x_square + square_size,
        fill=color, outline='gray5')
```

3.2.8 Funkcja wyświetlająca początkowy stan na siatce

Funkcja (kod 7) rysuje na siatce stan początkowy symulacji.

Kod 7: Funkcja wyświetlająca początkowy stan na siatce

```
def draw_init():
    for x in range(field_size):
        for y in range(field_size):
            if preys_cells[x, y]:
                color = 'green'
            elif predators_cells[x, y]:
                color = 'red'
            else:
                color = 'black'
            y_square = y * square_size
            x_square = x * square_size
            canvas_squares[x, y] = background.create_rectangle(
                y_square, x_square,
                y_square + square_size,
                x_square + square_size,
                fill=color, outline='gray5')
```

3.2.9 Funkcja ustalająca nowe położenie drapieżników

Funkcja (kod 8) odpowiada za ustalenie nowych położeń drapieżników. Pętla iteruje po wszystkich drapieżnikach. Jeśli atrybut drapieżnika 'hunger' osiągnął wartość parametru 'hunger_hunt', wtedy dla danego drapieżnika zostaje sprawdzone, która ofiara spośród tych w zasięgu wzroku znajduje się najbliżej niego, po czym ustala jego nowe położenie, tak aby przemieścił się w kierunku ofiary. Jeśli ofiara znajduje się odpowiednio blisko, jej obiekt zostaje usunięty z listy ofiar, a atrybut obiektu drapieżnika 'hunger' przyjmuje wartość '0'. Jeśli atrybut drapieżnika 'hunger' nie osiągnął wartości parametru 'hunger_hunt', lub w zasięgu wzroku nie znajduje się żadna ofiara, wtedy drapieżnik przemieści się losowo. Przemieszczenie może być przeprowadzone na sąsiadującą komórkę (po prostej lub po ukosie) lub osobnik może pozostać na obecnej. Jeśli nowe położenie drapieżnika znajduje się w komórce, która była wcześniej (określony czas temu) zajęta przez innego drapieżnika, wtedy przemieści się on na inną losowo wybraną komórkę, która nie była wcześniej zajmowana przez innego osobnika tego gatunku. Jeśli nowe położenie znajdzie się poza granicą, osobnik pojawi się po drugiej stronie siatki (periodyczne warunki brzegowe). Następnie miejsce siatki, które wcześniej było zajmowane przez osobnika oraz miejsce zgodne z nowym położeniem, zostają zaktualizowane. Drapieżnik nie przemieści się, jeśli jego nowa pozycja jest już zajęta przez innego osobnika. Funkcja zwraca listę pozostałych przy życiu drapieżników.

Kod 8: Funkcja ustalająca nowe położenie drapieżników

```
def move_predators():
    shuffle(predators)
    # iteracja po drapieżnikach:
    for predator in predators:
        nearest = predator_vision
        nearest_pos = None
        if predator.hunger >= hunger_hunt:

            # wśród wszystkich ofiar sprawdza, która z nich znajduje
            #   się w zasięgu wzroku.
            for checked_prey in preys:
                distance_squared = (predator.index[0] -
                  checked_prey.index[0]) ** 2 + (
                                    predator.index[1] -
                  checked_prey.index[1]) ** 2
                if distance_squared <= predator_vision_squared:
```

```
                distance = distance_squared ** 0.5

                # wśród tych które znajdują się w zasięgu
                #   sprawdza, która jest najbliżej.
                if distance <= nearest:
                    nearest = distance
                    nearest_pos = checked_prey

        # tworzy wektor ruchu w kierunku najbliższej ofiary.
        if nearest_pos:
            vector = [0, 0]
            vector[0] += (predator.index[0] - nearest_pos.index[0])
            vector[1] += (predator.index[1] - nearest_pos.index[1])
            vabs = (vector[0] ** 2 + vector[1] ** 2) ** 0.5
            if vabs:
                vector[0] = vector[0] / vabs
                vector[1] = vector[1] / vabs

                # określenie nowego położenia (new_x, new_y)
                # x
                if vector[0] > 0.5:
                    x_add = -1
                elif vector[0] < -0.5:
                    x_add = 1
                else:
                    x_add = 0
                # y
                if vector[1] > 0.5:
                    y_add = -1
                elif vector[1] < -0.5:
                    y_add = 1
                else:
                    y_add = 0
            else:
                x_add = 0
                y_add = 0
```

```
            new_xy = [predator.index[0] + x_add,
                      predator.index[1] + y_add]
        else:
            rand1 = randint(-1, 1)
            if rand1 == 0:
                rand2 = choice((-1, 1))
            else:
                rand2 = randint(-1, 1)
            new_xy = [predator.index[0] + rand1,
                      predator.index[1] + rand2]
    else:
        rand1 = randint(-1, 1)
        if rand1 == 0:
            rand2 = choice((-1, 1))
        else:
            rand2 = randint(-1, 1)
        new_xy = [predator.index[0] + rand1,
                  predator.index[1] + rand2]

    # upewnienie się, że nowe położenie nie leży poza siatką:
    new_xy[0] = new_xy[0] % field_size
    new_xy[1] = new_xy[1] % field_size
    new_xy = tuple(new_xy)

    # stworzenie listy, która zawiera wszystkie ślady innych
    #   drapieżników:
    all_traces = set()
    for checked_predator in predators:
        if predator != checked_predator:
            for trace in checked_predator.traces:
                all_traces.add(trace)

    # jeśli nowe położenie znajduje się w miejscu śladu innego
    #   drapieżnika, to zmiana położenia nastąpi w innym
    #   losowym kierunku:
    if new_xy in all_traces:
```

```
            possible_cell = set()
            for i in (-1, 0, 1):
                for j in (-1, 0, 1):
                    if i != 0 and j != 0:
                        check = ((predator.index[0] + i) % field_size,
                                 (predator.index[1] + j) % field_size)
                        if check not in all_traces and
                          not preys_cells[check] and
                          not predators_cells[check]:
                            possible_cell.add(check)
            if possible_cell:
                new_xy = choice(tuple(possible_cell))
            else:
                new_xy = predator.index
        else:
            # sprawdzenie, czy miejsce jest zajęte
            if predators_cells[new_xy] or preys_cells[new_xy]:
                new_xy = predator.index

        # zmiany atrybutów drapieżników:
        predator.prev_index = predator.index
        predator.index = new_xy
        predator.traces.append(predator.prev_index)
        predators_cells[predator.prev_index] = False
        draw_square(predator.prev_index)
        predators_cells[predator.index] = True
        draw_square(predator.index)
        if len(predator.traces) >= max_traces:
            predator.traces = predator.traces[1:]

        # zjedzenie ofiary, jeśli znajduje się dostatecznie blisko:
        if nearest_pos and nearest < 2:
            preys.remove(nearest_pos)
            preys_cells[nearest_pos.index] = False
            draw_square(nearest_pos.index)
            # jeśli nastąpiło zjedzenie, głód wraca do wartości 0,
```

```
            #   jeśli nie, zwiększa się o 1:
            predator.hunger = 0
        else:
            predator.hunger += 1

        if predator.age % predators_born == 0:
            if not predators_cells[predator.prev_index] and
              not preys_cells[predator.prev_index]:
                predators.append(Predator(predator.prev_index))
                predators_cells[predator.prev_index] = True
                draw_square(predator.prev_index)

        predator.age += 1

    # drapieżniki giną, jeśli ich głód osiągnie określony poziom
    #   (hunger_death)
    predators_new = []
    for predator in predators:
        if predator.hunger == hunger_death:
            predators_cells[predator.index] = False
            draw_square(predator.index)
        else:
            predators_new.append(predator)
    return predators_new
```

3.2.10 Funkcja ustalająca nowe położenie ofiar

Funkcja (kod 9) odpowiada za ustalenie nowych położeń ofiar. Nowa pozycja zostaje wybrane losowo. Przemieszczenie może być przeprowadzone na sąsiadującą komórkę (po prostej lub po ukosie). Jeśli nowe położenie znajdzie się poza granicą, osobnik pojawi się po drugiej stronie siatki (periodyczne warunki brzegowe). Jeśli nowe położenie jest już zajęte przez innego osobnika, przemieszczenie nie nastąpi. Następnie miejsce siatki, które wcześniej było zajmowane przez osobnika oraz miejsce zgodne z nowym położeniem, zostają zaktualizowane.

Kod 9: Funkcja ustalająca nowe położenie ofiar

```
def move_preys():
    shuffle(preys)
    # iteracja po ofiarach
    for prey in preys:
        rand1 = randint(-1, 1)
        if rand1 == 0:
            rand2 = choice((-1, 1))
        else:
            rand2 = randint(-1, 1)

        new_xy = [prey.index[0] + rand1,
                  prey.index[1] + rand2]

        # upewnienie się, że nowe położenie nie leży poza siatką:
        new_xy[0] = new_xy[0] % field_size
        new_xy[1] = new_xy[1] % field_size
        new_xy = tuple(new_xy)

        # sprawdzenie, czy miejsce jest zajęte
        if preys_cells[new_xy] or preys_cells[new_xy]:
            new_xy = prey.index

        # zmiany atrybutów ofiar
        prey.prev_index = prey.index
        prey.index = new_xy

        # usuwanie kwadratów, które odpowiadają poprzednim położeniom
        #   ofiar
        preys_cells[prey.prev_index] = False
        draw_square(prey.prev_index)
        # rysowanie kwadratów, które odpowiadają aktualnym położeniom
        #   ofiar
        preys_cells[prey.index] = True
        draw_square(prey.index)
```

```
        # rodzenie się nowych ofiar
        if prey.age % preys_born == 0:
            if not predators_cells[prey.prev_index] and
              not preys_cells[prey.prev_index]:
                preys.append(Prey(prey.prev_index))
                preys_cells[prey.prev_index] = True
                draw_square(prey.prev_index)

        prey.age += 1
```

3.2.11 Pętla tworząca kroki czasowe

W tym miejscu (kod 10) najpierw są wywołane funkcje 'init_orgs' i 'draw_init'. Pierwsza tworzy listy z obiektami osobników, a druga rysuje ich początkowe położenie. Następnie zostaje otwarty plik, do którego będą zapisywane liczebności osobników i następuje zapisanie nazw kolumn. Dalej zostaje utworzona pętla, w której każda iteracja będzie osobnym krokiem czasowym. Zmienna 'loops_done' określa, ile razy wykonała się główna pętla odpowiedzialna za kroki czasowe. Zmienna 'loops_index' oznacza, który raz dane o liczebnościach osobników zostają dodawane do pliku. Następnie wywołane są funkcje 'move_predators' oraz 'move_preys'. Na koniec, we wcześniej otwartym pliku zapisane zostają aktualne liczebności osobników.

Kod 10: Pętla tworząca kroki czasowe

```
preys, predators = init_orgs()
draw_init()
file = open("predators_preys.txt", "a")
file.write("index;predators;preys\n")
index = 1
loops_done = 0
while loops_done < no_of_generations:
    predators = move_predators()
    move_preys()

    background.update()
    sleep(1 / speed)
```

```
    # zliczanie liczebności zwierząt do pliku
    if loops_done % report_time == report_time - 1:
        file.write("{};{};{}\n".format(index, len(predators),
          len(preys)))
        index += 1

    loops_done += 1
file.close()
mainloop()
```

3.3 Konfrontacja symulacji z modelem Lotki-Volterry

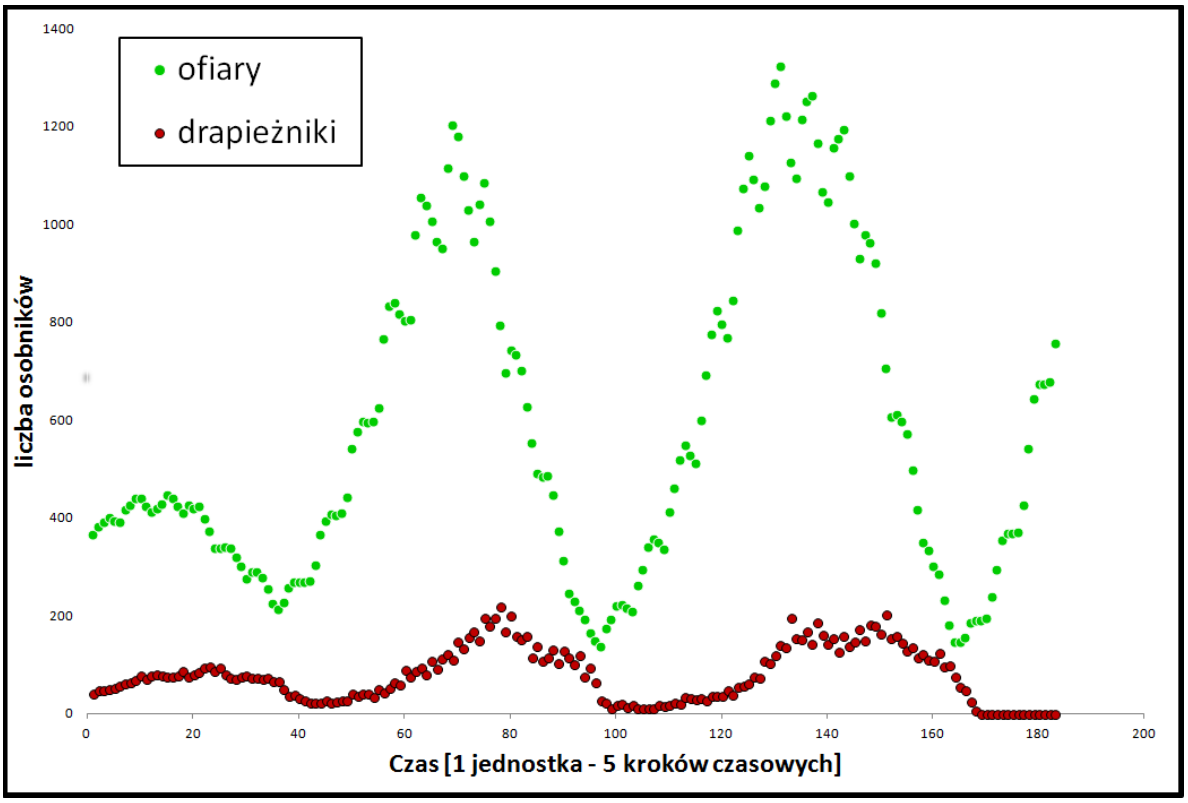
Symulacja była inspirowana klasycznym modelem Lotki-Volterry [2] [8] (dalej oznaczanym jako L-V), jednak nie wyraża go w idealny sposób. Model L-V ma naturę ciągłą i nie uwzględnia położenia poszczególnych osobników. Symulacja ma charakter dyskretny, a poruszające się osobniki zawsze znajdują się w konkretnych miejscach.

Analizując równanie L-V (wzór 1, w podrozdziale 2.2), oraz konfrontując z nim parametry symulacji (opisane w podrozdziale 3.2.2), można znaleźć pewne podobieństwa:

- Parametr 'preys_born' można utożsamiać ze współczynnikiem przyrostu ofiar (r). Zwiększając parametr, ofiary namnażają się w szybszym tempie.
- Parametr 'predators_born' można utożsamiać ze współczynnikiem przyrostu drapieżników (b). Zwiększając parametr, drapieżniki namnażają się w szybszym tempie.
- Parametr 'hunger_death' działa na innej zasadzie, niż współczynnik umieralności drapieżników (s). Parametr powoduje śmierć po ustalonej liczbie kroków, a współczynnik powoduje wykładnicze zanikanie populacji.

W symulacji, w miejscach dużego zagęszczenia ofiar, rośnie zagęszczenie drapieżników. Gdy rośnie zagęszczenie drapieżników w danym miejscu, spada tam liczebność ofiar. Gdy spada zagęszczenie ofiar, zagęszczenie drapieżników również spada w tym miejscu. Mniejsze zagęszczenie drapieżników pozwala na przyrost liczby ofiar. Występuje quasi-okresowość, która może się utrzymywać, lub ulec zaburzeniu.

Rysunek 3: Przykładowy wykres obrazujący liczbę drapieżników i ofiar w poszczególnych krokach.



3.4 Wady i ograniczenia modelu

Warto wspomnieć o wadach oraz ograniczeniach, które zaobserwowano w modelu oraz w jego implementacji:

- Model jest stosunkowo złożony, gdyż wymaga wyznaczenia interakcji między osobnikami. Potrzeba do tego dużej wydajności obliczeniowej, szczególnie przy wielu osobnikach na siatce. Optymalne czasowo zaprogramowanie tego procesu może być przedmiotem osobnych analiz.

- Każdy drapieżnik, który poluje, sprawdza w jakiej odległości znajduje się każda z obecnych ofiar, niezależnie, jak jest daleko. Lepszym rozwiązaniem był by mechanizm, w którym drapieżnik sprawdzałby tylko pola, które ma w zasięgu i na tej podstawie badał, czy ofiara się tam znajduje. Wymagałoby to takiej reprezentacji danych odnośnie położeń ofiar, aby po wskazaniu danego pola mieć dostęp do informacji, jaka ofiara się na nim znajduje.

- Osobniki poruszają się na boki (o 1 jednostkę), lub na ukos (o $\sqrt{2}$ jednostki). Oznacza to, że nie zawsze zwierzę porusza się na jednakową odległość.
- Gdy drapieżnik ginie, wszystkie jego ślady znikają. W realnym środowisku po śmierci osobnika jego ślad dalej może oddziaływać na inne osobniki.
- Drapieżniki mogą przechodzić przez krawędź siatki i znaleźć się po drugiej stronie tylko, gdy poruszają się losowo. Nie widzą one ofiar, które znajdują się po drugiej stronie siatki.

4 Analiza wpływu terytorializmu na działanie modelu

4.1 Przebieg analizy

W analizie porównano zachowanie modelu bez mechanizmu symulującego terytorializm u drapieżników, oraz z obecnym mechanizmem (mechanizm opisany w podsekcji 2.3.2). Analiza została oparta na liczebnościach drapieżników i ofiar, wygenerowanych przez program opisany w podsekcji 3.2. Pliki z danymi zostały załączone do pracy. Symulacja została przeprowadzona 60 razy. Każda z nich trwała maksymalnie 3000 kroków czasowych (tyle trwała, jeśli przynajmniej jeden drapieżnik wciąż był przy życiu). Gdy wszystkie drapieżniki wyginęły, symulacja kończyła działanie. W 30 przypadkach parametr 'max_traces' ustawiony został na 0 (brak terytorializmu), a w pozostałych 30 parametr ten był ustawiony na 20 (terytorializm obecny, każdy ślad drapieżnika był trwały przez 20 kroków czasowych).

Pozostałe parametry zostały przypisane następująco:

- no_of_generations = 3000,
  maksymalny czas symulacji to 3000 kroków czasowych,
- report_time = 5,
  liczebność osobników zliczana jest co 5 kroków czasowych,
- field_size = 80,
  wielkość siatki: 80 x 80 komórek, co daje 6400 komórek,
- predator_vision = 4,
  promień widzenia drapieżnika: 4 komórki,
- hunger_hunt = 4,
  drapieżnik po urodzeniu, lub po zjedzeniu ofiary, nie poluje przez 4 kroki czasowe,

- hunger_death = 8,
  drapieżnik po 8 krokach czasowych bez upolowania ofiary, umiera,
- preys_born = 34,
  ofiara pozostawia potomka raz na 34 kroki czasowe,
- predators_born = 12,
  drapieżnik pozostawia potomka raz na 12 kroków czasowych,
- preys_percent = 5,
  5% dostępnych komórek zostaje wstępnie zapełnionych ofiarami, czyli 320,
- predators_percent = 0.5,
  0.5% dostępnych komórek zostaje wstępnie zapełnionych drapieżnikami, czyli 32.

Przeanalizowano wielkości takie jak:

- średnia liczba drapieżników w pojedynczym kroku czasowym
- średnia liczba ofiar w pojedynczym kroku czasowym
- średnia maksymalna liczba ofiar w pojedynczej symulacji
- średnia maksymalna liczba drapieżników w pojedynczej symulacji

Do sprawdzenia istotności różnic pomiędzy wielkościami posłużono się testem nieparametrycznym Manna-Whitneya-Wilcoxona [7]. Obliczenia przeprowadzono w programie R [10] (funkcja 'wilcox.test()'). Dla każdego testu hipoteza zerowa brzmi: wielkości nie są istotnie różne, a hipoteza alternatywna: wielkości różnią się istotnie.

4.2 Wyniki

W tym podrozdziale przedstawiono wyniki testów Manna-Whitneya-Wilcoxona.

4.2.1 Średnia liczba drapieżników w pojedynczym kroku czasowym

Rodzaj modelu	Średnia	Wartość p	Wynik
bez terytorializmu	91.94	0.000	hipoteza alternatywna
z terytorializmem	79.66		

Tabela 1: Wynik testu Manna-Whitneya-Wilcoxona dla porównania średnich liczb drapieżników w pojedynczym kroku czasowym

Wynik: przyjmujemy hipotezę alternatywną, wielkości różnią się istotnie.

4.2.2 Średnia liczba ofiar w pojedynczym kroku czasowym

Rodzaj modelu	Średnia	Wartość p	Wynik
bez terytorializmu	900.20	0.000	hipoteza alternatywna
z terytorializmem	653.31		

Tabela 2: Wynik testu Manna-Whitneya-Wilcoxona dla porównania średnich liczb ofiar w pojedynczym kroku czasowym

Wynik: przyjmujemy hipotezę alternatywną, wielkości różnią się istotnie.

4.2.3 Średnia maksymalna liczba drapieżników w pojedynczej symulacji

Rodzaj modelu	Średnia	Wartość p	Wynik
bez terytorializmu	260.67	0.096	hipoteza zerowa
z terytorializmem	242.30		

Tabela 3: Wynik testu Manna-Whitneya-Wilcoxona dla porównania średnich maksymalnych liczb drapieżników w pojedynczej symulacji

Wynik: brak podstaw do odrzucenia hipotezy zerowej, wielkości nie różnią się istotnie.

4.2.4 Średnia maksymalna liczba ofiar w pojedynczej symulacji

Rodzaj modelu	Średnia	Wartość p	Wynik
bez terytorializmu	2513.10	0.000	hipoteza alternatywna
z terytorializmem	1846.57		

Tabela 4: Wynik testu Manna-Whitneya-Wilcoxona dla porównania średnich maksymalnych liczb ofiar w pojedynczej symulacji

Wynik: przyjmujemy hipotezę alternatywną, wielkości różnią się istotnie.

4.3 Wnioski

Można stwierdzić, że mechanizm terytorializmu istotnie wpływa na zachowanie modelu. Terytorializm sprawił, że mniej osobników żyło w tym samym czasie, a więcej komórek pozostawało pustych. W symulacji bez terytorializmu, populacja ofiar rozrastała się do większej liczby. Być może działo się tak dlatego, że drapieżniki, dzieląc się przestrzenią, bardziej efektywnie pomniejszały populację ofiar. Wnioski zostały dokonane na podstawie danych z programu komputerowego. Aby stwierdzić różnicę w realnym świecie, należałoby przeprowadzić analizę danych zebranych w środowisku.

Literatura

[1] Berto, Francesco and Tagliabue, Jacopo, *„Cellular Automata”*, The Stanford Encyclopedia of Philosophy (Fall 2017 Edition), Edward N. Zalta (ed.),
URL: https://plato.stanford.edu/archives/fall2017/entries/cellular-automata/

[2] U. Foryś, 2005, *„Matematyka w biologii”*, 66-72

[3] P. Sumiński, J. Goszczyński, J. Romanowski, 1993, *„Ssaki drapieżne Europy”*, 166-171

[4] Jonathan R. Potts and Mark A. Lewis, 2014, *„How do animal territories form and change? Lessons from 20 years of mechanistic modelling”* Proc. R. Soc. B.28120140231,
URL: https://doi.org/10.1098/rspb.2014.0231

[5] Stephen Wolfram, 2002, *„A New Kind of Science”*
URL: https://www.wolframscience.com

[6] Games, M., 1970, *„The fantastic combinations of John Conway's new solitaire game „life” by Martin Gardner”* Scientific American, 223, 120–123

[7] Mann, H. B., & Whitney, D. R., 1947, *„On a test of whether one of two random variables is stochastically larger than the other”*. Annals of Mathematical Statistics, 18, 50–60

[8] V. Volterra, 1931, *„Variations and fluctuations of the number of indviduals in animal species living together”* Animal Ecology. McGraw-Hill. Translated from 1928 edition by R. N. Chapman.

[9] Python programming language,
URL = https://www.python.org

[10] The R Project for Statistical Computing,
URL: https://www.r-project.org

[11] Potts JR, Harris S, Giuggioli L, 2013, „Quantifying behavioural changes in territorial animals caused by sudden population declines” Am. Nat. 182