



UNIwersYTET GDAŃSKI

Gabriel Kaszewski
Nr albumu: 281113

Modelowanie procesów ewolucyjnych za pomocą automatów komórkowych.

Kierunek studiów: Bioinformatyka
Poziom studiów: I stopnia - licencjackie
Profil studiów: ogólnoakademicki
Forma studiów: studia stacjonarne

Praca dyplomowa licencjacka
wykonana pod kierunkiem
dr Adrian Kołodziejski

Gdańsk, 2025 r.

Spis treści

1. Streszczenie	3
2. Wprowadzenie	3
3. Entity Component System	4
3.1. Wstęp	4
3.2. Encje	4
3.3. Komponenty	4
3.4. Systemy	4
3.5. Świat	4
3.6. Zalety Entity Component System	4
3.7. Wielowątkowość w Entity Component System	5
3.8. Podsumowanie	5
4. Automaty komórkowe	5
4.1. Wstęp	5
4.2. Definicja	5
4.3. Przykłady	6
5. Mój model	6
5.1. Implementacja	6
5.2. Symulacja	7
5.3. Wizualizacja	30
5.4. Wyniki	33
5.4.1. Scenariusz 1 - Wpływ poziomu mutacji na adaptację populacji	33
5.4.2. Scenariusz 2 - Wpływ drapieżników na populację ofiar	45
5.4.3. Scenariusz 3 - Wpływ dostępności jedzenia na populację ofiar	53
5.5. Podsumowanie wyników i wnioski	71
5.5.1. Wpływ poziomu mutacji	71
5.5.2. Rola drapieżników	71
5.5.3. Dostępność jedzenia i środowisko	71
6. Podsumowanie	71
6.1. Kierunki dalszego rozwoju	72
Bibliografia	72
7. Kod źródłowy	72

Modelowanie procesów ewolucyjnych za pomocą automatów komórkowych

Gabriel Kaszewski

1. Streszczenie

Celem niniejszej pracy było opracowanie symulacji ewolucji populacji organizmów funkcjonujących w zróżnicowanym środowisku. Inspiracją do podjęcia tego tematu była koncepcja automatów komórkowych; prostych modeli obliczeniowych, w których lokalne reguły mogą prowadzić do powstawania złożonych, emergentnych zachowań. W ramach realizacji projektu wykorzystano język **Rust** wraz z biblioteką **Bevy** do implementacji symulacji, natomiast do analizy i wizualizacji wyników posłużono się językiem **Python** oraz popularnymi bibliotekami do przetwarzania danych. Efektem pracy jest program umożliwiający symulację procesów ewolucyjnych populacji organizmów, w którym Użytkownik może definiować różne parametry, takie jak początkowa ilość ofiar, drapieżników, stopień mutacji, rozmiar planszy, początkowy próg reprodukcji itp. wpływające na przebieg symulacji i obserwować zmiany zachodzące w populacjach w zależności od zadanych warunków.

W pracy rozważyłem trzy różne scenariusze. W pierwszym scenariuszu badałem jak poziom mutacji wpływa na adaptację populacji do środowiska. Rezultaty pokazały, że zbyt niski poziom mutacji ogranicza zdolności adaptacyjne organizmów. Z kolei zbyt wysoki poziom prowadził do niestabilności cech, co skutkowało chaotycznym rozkładem przystosowań i częstymi wahaniami populacji. Optymalne rezultaty uzyskano przy średnim poziomie mutacji. W drugim scenariuszu badałem rolę drapieżników w procesie ewolucji ofiar. Wyniki pokazały, że umiarkowana presja drapieżników stabilizuje populację i promuje rozwój korzystnych cech, takich jak większa szybkość czy niższy próg reprodukcji. Jednak zbyt silna presja prowadziła do wyginięcia ofiar, a w konsekwencji również drapieżników. Co interesujące, ofiary nie ewoluowały w kierunku mniejszych i szybszych jednostek, lecz przeciwnie, stawały się większe i miały wyższy próg reprodukcji. W trzecim scenariuszu analizowałem wpływ dostępności jedzenia i różnorodności środowiska na strategię przetrwania. W środowiskach bogatych populacje ofiar rozwijały się dynamicznie, lecz często prowadziło to do przeludnienia. W środowiskach jałowych przeżywały jedynie najlepiej przystosowane jednostki, a tempo wzrostu populacji było znacznie wolniejsze.

Symulacja wykazała, że nawet proste reguły mogą prowadzić do nieoczywistych i złożonych efektów ewolucyjnych. Przykładowo, w warunkach silnej presji drapieżników ofiary nie ewoluowały w kierunku mniejszych, energooszczędnych form, lecz przeciwnie stawały się większe i bardziej zasobochłonne. Jest to przykład emergentnego wyścigu zbrojeń pomiędzy drapieżnikiem a ofiarą. Uzyskane wyniki pokazują potencjał tego typu modeli do badania zjawisk biologicznych oraz ich dalszego rozwoju. Kierunkiem dalszego rozwoju może być m. in. wprowadzenie mechanizmów uczenia lub pamięci, symulacja większej ilości gatunków, wprowadzenie zmiennych klimatycznych, genomów czy genotypów.

2. Wprowadzenie

Celem pracy jest zbadanie możliwości modelowania procesów ewolucyjnych przy wykorzystaniu automatów komórkowych i **Entity Component System**, który w dalszej części pracy będzie przedstawiany jako **ECS**. Praca ma na celu nie tylko pogłębienie teoretycznych podstaw modelowania ewolucji przy użyciu dyskretnych metod obliczeniowych, ale także popularyzację nowoczesnych technik programistycznych.

Symulację napisałem w języku Rust z wykorzystaniem biblioteki **Bevy**, która jest oparta na **ECS** i dostarcza wszystkie potrzebne komponenty do stworzenia projektu multimedialnego. Do wizualizacji danych użyłem języka **Python** z całym pakietem bibliotek do analizy danych takich jak **NumPy**, **Matplotlib**, **Pandas**.

3. Entity Component System

3.1. Wstęp

ECS to współczesna architektura oprogramowania stosowana głównie w grach komputerowych oraz symulacjach. Umożliwia ona elastyczne i wydajne zarządzanie złożonymi systemami. Główną ideą ECS jest oddzielenie danych (komponentów) od logiki (systemów) oraz traktowanie encji (patrz punkt 3.2) jedynie jako identyfikatorów, co pozwala na łatwiejsze skalowanie, modyfikację oraz optymalizację (np. w kontekście wielowątkowości). W tym rozdziale wyjaśniam podstawowe pojęcia związane z ECS, jego strukturę oraz zalety.

3.2. Encje

Encje (ang. *entities*) stanowią podstawowy element ECS. Są one reprezentowane najczęściej jako unikalne identyfikatory (np. liczby całkowite) i same w sobie nie zawierają żadnych danych ani elementów logicznych. Encje są „nosicielami” komponentów, które definiują ich właściwości. Dzięki temu, encje zajmują mało miejsca w pamięci, a zarządzanie nimi - np. tworzenie, usuwanie czy modyfikacja - odbywa się w sposób efektywny, ponieważ nie wymaga to przeszukiwania złożonych struktur danych.

3.3. Komponenty

Komponenty (ang. *components*) to struktury danych, które zawierają właściwości lub stany encji. Każdy komponent jest dedykowany określonej właściwości obiektu, np. pozycji, prędkości czy „zdrowiu”. W ECS komponenty nie zawierają żadnej logiki, są one jedynie kontenerami na dane. Z racji tego, że ich główną rolą jest przechowywanie informacji, są one zazwyczaj proste i niezależne od siebie. Co więcej, komponenty powinny być rozdzielane na mniejsze, wyspecjalizowane jednostki. Dzięki temu systemy mogą operować na dokładnie tych danych, które są im potrzebne, co sprzyja modularności i łatwości wprowadzaniu zmian, a przez to wydajności.

3.4. Systemy

Systemy (ang. *systems*) to moduły odpowiedzialne za logikę działania symulacji albo gry. Operują one na zbiorach encji, które posiadają określone komponenty. Przykładowo, system odpowiedzialny za ruch będzie aktualizował pozycje encji, które posiadają komponent *Pozycja* oraz *Prędkość*. Systemy wykonują swoje operacje cyklicznie (np. w każdej klatce gry) i mogą być projektowane tak, aby działały niezależnie od siebie.

3.5. Świat

Świat w kontekście ECS to kontener, który przechowuje wszystkie encje, komponenty i systemy danego projektu. Stanowi centralny punkt, za pomocą którego systemy mogą uzyskać dostęp do danych i komunikować się między sobą.

3.6. Zalety Entity Component System

Architektura ECS posiada szereg zalet:

- **Modularność:** oddzielenie danych od logiki umożliwia łatwe modyfikowanie i rozszerzanie funkcjonalności bez wpływu na całą strukturę aplikacji.
- **Wydajność:** komponenty są przechowywane w pamięci w sposób ciągły, co sprzyja lokalności danych i pozwala na efektywne operacje na nich, ponieważ procesor o wiele szybciej uzyskuje dostęp do danych, które znajdują się w tzw. *cache'u* niż z pamięci RAM.
- **Elastyczność:** można łatwo dodawać lub usuwać funkcjonalności przez modyfikację lub dodanie nowych komponentów i systemów.
- **Skalowalność:** ECS doskonale nadaje się do obsługi dużej liczby encji, co jest szczególnie ważne w symulacjach oraz grach z wieloma interaktywnymi obiektami.
- **Łatwość debugowania i testowania:** dzięki modułowej budowie, testowanie i debugowanie poszczególnych komponentów i systemów jest znacznie prostsze.

3.7. Wielowątkowość w Entity Component System

Jednym z kluczowych atutów ECS jest możliwość łatwego wykorzystania wielowątkowości. Dzięki wyraźnemu oddzieleniu systemów, które operują na niepowiązanych zestawach komponentów, można równolegle przetwarzać dane w różnych wątkach. To podejście nie tylko zwiększa wydajność symulacji, ale także pozwala na lepsze wykorzystanie współczesnych procesorów wielordzeniowych. W praktyce oznacza to, że systemy nie muszą blokować siebie nawzajem, co znacznie poprawia skalowalność i responsywność aplikacji.

3.8. Podsumowanie

ECS to nowoczesne podejście do projektowania systemów, które wyróżnia się modularnością, wydajnością i elastycznością. Dzięki oddzieleniu encji, komponentów i systemów możliwe jest tworzenie skomplikowanych symulacji oraz gier w sposób przejrzysty i łatwy do skalowania. Dodatkową zaletą jest możliwość równoległego przetwarzania, co jest kluczowe w aplikacjach wymagających wysokiej wydajności. ECS stanowi doskonałą bazę do implementacji symulatorów oraz innych systemów, w których liczy się szybkie przetwarzanie dużej liczby obiektów, co czyni go idealnym narzędziem w kontekście bioinformatyki i modelowania ewolucyjnego.

4. Automaty komórkowe

4.1. Wstęp

Automaty komórkowe (ang. *cellular automata*) to dyskretne modele obliczeniowe, w których przestrzeń symulacji dzielona jest na regularną siatkę komórek. Każda komórka może znajdować się w jednym, ze stanów, które należą do zbioru dyskretnego i ograniczonego, a jej ewolucja odbywa się według ustalonych reguł, zależnych od stanów sąsiadujących komórek. Metoda ta umożliwia badanie złożonych układów i procesów dynamicznych przy wykorzystaniu prostych reguł lokalnych, co czyni automaty komórkowe atrakcyjnym narzędziem w symulacjach biologicznych, fizycznych oraz społecznych.

Pierwsze idee dotyczące automatów komórkowych pojawiły się już w latach 40. XX wieku, gdy John von Neumann i Stanisław Ulam badali samoreplikujące się układy. Jednak prawdziwy rozwój tej dziedziny nastąpił w kolejnych dekadach, kiedy to naukowcy zaczęli systematycznie wykorzystywać te modele do badania dynamiki złożonych układów. Jednym z przełomowych momentów była publikacja gry w życie (ang. *Game of Life*) autorstwa Johna Conwaya w 1970 roku, która stała się symbolem możliwości generowania złożonych struktur przy użyciu bardzo prostych zasad [1]. W latach 80. i 90. automaty komórkowe znalazły szerokie zastosowanie w badaniach nad zjawiskami samoorganizacji oraz samoorganizującej się krytyczności (ang. *Self-Organized Criticality*) [2], czyli zdolności układów dynamicznych do spontanicznego przechodzenia w stan krytyczny. Stały się również inspiracją dla badań nad sztucznym życiem, modelując emergentne zachowania przypominające procesy biologiczne [3].

4.2. Definicja

Automat komórkowy definiuje się jako układ składający się z trzech podstawowych elementów:

- **Siatka komórek:** przestrzeń, w której każda komórka ma określoną pozycję (np. w układzie regularnym, takim jak kwadratowa lub heksagonalna siatka).
- **Zbiór stanów:** dyskretny zbiór wartości, które mogą przyjmować poszczególne komórki (np. 0 lub 1, kolor, liczba, itp.).
- **Reguły przejścia:** zbiór zasad, według których stan każdej komórki jest aktualizowany w kolejnych krokach czasowych, zależnie od stanów sąsiadów. Aktualizacja zwykle odbywa się synchronicznie dla wszystkich komórek.

Takie podejście umożliwia obserwację tego, jak proste reguły lokalne mogą prowadzić do powstawania skomplikowanych, globalnych wzorców i struktur. [4]

4.3. Przykłady

Najbardziej znanym przykładem automatu komórkowego jest gra w życie Johna Conwaya [1], w której proste zasady dotyczące narodzin, przetrwania i śmierci komórek prowadzą do złożonych, często nieprzewidywalnych zachowań. Inne przykłady obejmują:

- **Elementarne automaty komórkowe:** Badane przez Stephena Wolframa, gdzie komórki mają tylko dwa stany, a reguły są określone na podstawie stanu sąsiadów w jednym wymiarze. Przykłady takich reguł to reguła 30 czy reguła 110 [5].
- **Automaty oparte o inne struktury siatek:** Automaty działające na siatkach heksagonalnych lub trójwymiarowych, które mogą lepiej modelować niektóre procesy naturalne.
- **Specjalistyczne modele:** Automaty komórkowe stosowane w modelowaniu wzrostu tkanek, rozprzestrzeniania się epidemii czy dynamiki ruchu tłumu. [6], [7]

Te przykłady pokazują, że automaty komórkowe są niezwykle wszechstronnym narzędziem, które znalazło zastosowanie zarówno w teorii, jak i w praktycznych zastosowaniach.

5. Mój model

Zainspirowany automatami komórkowymi stworzyłem symulację ewolucji populacji organizmów. Wyróżniamy dwa rodzaje organizmów: ofiary i drapieżniki.

Plansza, bądź świat, w którym odbywa się symulacja jest dwuwymiarową siatką, zbudowaną z czterech rodzajów kafelków (woda, las, pustynia i trawa). Każdy typ terenu ma swoje własne właściwości jak np. dostępność pożywienia, prędkość odnowy pożywienia, jak szybko i chętnie organizmy poruszają się po nim.

W mojej symulacji istnieją dwa rodzaje organizmów: ofiary i drapieżniki. Ofiary dostają energię z jedzenia, które jest na danym kafelku. Drapieżniki z kolei dostają energię z jedzenia ofiar. Każdy organizm ma swoje własne cechy jak energia, prędkość, rozmiar, próg rozmnażania, tolerancja terenu, drapieżniki mają dodatkowo wydajność polowania i próg głodu.

Wszystkie organizmy potrzebują energii do przeżycia. Jeśli zabraknie tego zasobu, jednostka ginie. Natomiast, jeśli organizm przekroczy próg rozmnażania, to tworzy nową jednostkę. W każdym kroku czasowym każda jednostka zużywa energię, tym więcej im jest szybsza. Drapieżniki polują na ofiary, gdy ich energia spadnie poniżej progu głodu i poszukuje najbliższej ofiary. Każdy drapieżnik ma zasięg polowania, który wynosi jedną komórkę. Wydajność polowania wpływa na to ile energii drapieżnik dostanie z jedzenia.

W sytuacji, gdy na jednej komórce znajduje się więcej niż jedna ofiara, to pożywienie dostaje w pierwszej kolejności osobnik, który jest największy, przy czym ilość uzyskanej energii z jedzenia również zależy od rodzaju terenu, na którym znajduje się i jego przystosowania do tego terenu. Jeśli na jednej komórce znajduje się więcej ofiar, bądź drapieżników niż ustawiony limit to nadwyżka ginie z powodu przeludnienia.

Podczas rozmnażania organizm dziedziczy cechy rodzica, ale może również wystąpić mutacja cech. Mutacja polega na zmianie cechy o losową wartość z przedziału, który zależy od mutacji. Dziecko rodzi się z połową energii rodzica.

5.1. Implementacja

Kod źródłowy jest podzielony na dwie części. Pierwszą z nich jest symulacja napisana w języku **Rust** z wykorzystaniem biblioteki **Bevy**. Natomiast drugą część stanowi wizualizacja wyników do stworzenia, której wykorzystałem język **Python** oraz biblioteki **Matplotlib**, **NumPy** i **Pandas**.

W punkcie 5.2 przedstawiam kod źródłowy wraz z jego krótkim objaśnieniem.

5.2. Symulacja

```
use std::error::Error;
use std::fmt::Display;
use std::fs;
use std::fs::File;
use std::fs::OpenOptions;
use std::io::Write;

use bevy::prelude::*;
use bevy::utils::hashbrown::HashMap;
use noise::NoiseFn;
use noise::Perlin;
use rand::prelude::*;
use serde::Deserialize;
use serde::Serialize;
```

Listing 1: Importy

W listingu 1 przedstawione są zaimportowane biblioteki, które są potrzebne do napisania symulacji są to m. in.: `std::error::Error` do obsługi błędów, `std::fs` do operacji na plikach, `std::io::Write` do zapisywania danych do pliku. Wykorzystuję również bibliotekę `bevy`, która ma odpowiednie funkcje i struktury do tworzenia multimedialnego projektu, stosując przy tym architekturę ECS, `noise` do generowania szumu, którego użycie wprowadza niedeterminizm do mojego modelu, `rand` do generowania liczb pseudolosowych oraz `serde` do serializacji i deserializacji danych, które później wykorzystuję do wizualizacji wyników w języku **Python**.

```
#[derive(Deserialize, Debug, Serialize, Clone)]
struct BiomeDataConfig {
    food_availability: f32,
    max_food_availability: f32,
}

#[derive(Deserialize, Debug, Resource, Serialize, Clone)]
pub struct Config {
    width: usize,
    height: usize,
    initial_organisms: usize,
    initial_predators: usize,
    headless: bool,
    log_data: bool,
    forest: BiomeDataConfig,
    desert: BiomeDataConfig,
    water: BiomeDataConfig,
    grassland: BiomeDataConfig,
    initial_organism_energy: f32,
    initial_predator_energy: f32,
    initial_organism_speed: f32,
    initial_predator_speed: f32,
    initial_organism_size: f32,
    initial_predator_size: f32,
    initial_organism_reproduction_threshold: f32,
    initial_predator_reproduction_threshold: f32,
    initial_predator_hunting_efficiency: f32,
    initial_predator_satiation_threshold: f32,
    organism_mutability: f32,
    predator_mutability: f32,
    overcrowding_threshold_for_organisms: usize,
    overcrowding_threshold_for_predators: usize,
    max_total_entities: usize,
    seed: u64,
}
```

Listing 2: Struktury konfiguracyjne

W listingu 2 przedstawione są struktury konfiguracyjne. Struktura BiomeDataConfig przechowuje informacje o dostępności pożywienia na danym terenie oraz maksymalnej dostępności pożywienia. Struktura Config zawiera informacje o szerokości i wysokości planszy, liczbie początkowych ofiar i drapieżników, czy dane mają być zapisywane do pliku, a także parametry początkowe dla ofiar i drapieżników, takie jak *energia*, *prędkość*, *rozmiar*, *próg rozmnażania*, *wydajność polowania*, *próg głodu*, *mutowalność cech*, *próg przeludnienia* oraz *ziarno generatora liczb losowych*. Dodatkowo w tej strukturze możemy wybrać czy symulacja ma być uruchomiona w trybie bez okna. Struktura ta jest serializowana i deserializowana za pomocą biblioteki *serde* oraz jest dostępna jako zasób w **Bevy**, co powoduje, że jest dostępna dla każdego systemu.

Ziarno generatora liczb pseudolosowych jest potrzebne, by symulacja była deterministyczna, a wyniki były zawsze takie same, co jest ważne przy testowaniu i reprodukowaniu. Plik konfiguracyjny jest zapisany w formacie **TOML** i zawiera następujące wartości parametrów:

```
width = 100
height = 100
initial_organisms = 5
initial_predators = 5
headless = true
log_data = true
initial_organism_energy = 3.0
initial_predator_energy = 15.0
initial_organism_speed = 1.0
initial_predator_speed = 1.5
initial_organism_size = 1.2
initial_predator_size = 1.0
initial_organism_reproduction_threshold = 5.0
initial_predator_reproduction_threshold = 16.0
initial_predator_hunting_efficiency = 1.5
initial_predator_satiation_threshold = 14.0
organism_mutability = 0.01
predator_mutability = 0.01
overcrowding_threshold_for_organisms = 25
overcrowding_threshold_for_predators = 10
max_predator_energy = 12500.0
predator_energy_decay_rate = 0.1
organism_reproduction_cooldown = 7.0
predator_reproduction_cooldown = 2.0
seed = 420692137
max_total_entities = 100000
generation_limit = 1000
printing = false

[forest]
food_availability = 0.2
max_food_availability = 2600.0
temperature = 20.0
humidity = 0.6

[desert]
food_availability = 0.01
max_food_availability = 300.0
temperature = 35.0
humidity = 0.1

[water]
food_availability = 0.0
max_food_availability = 0.0
temperature = 15.0
humidity = 0.9
```



```
[grassland]
food_availability = 0.1
max_food_availability = 1500.0
temperature = 25.0
humidity = 0.4
```

Dzięki temu, że symulacja wczytuje plik konfiguracyjny, można łatwo zmieniać jej parametry bez konieczności zmiany kodu źródłowego i ponownej kompilacji programu.

```
#[derive(Debug, Clone, PartialEq, Eq, Hash, Serialize, Copy)]
pub enum Biome {
    Forest,
    Desert,
    Water,
    Grassland,
}

#[derive(Debug, Clone, Serialize)]
pub struct Tile {
    pub biome: Biome,
    pub temperature: f32,
    pub humidity: f32,
    pub food_availability: f32,
}

#[derive(Debug, Resource, Serialize, Clone)]
pub struct World {
    pub width: usize,
    pub height: usize,
    pub grid: Vec<Vec<Tile>>,
}
```

Listing 3: Struktury reprezentujące świat

W listingu 3 przedstawione są struktury opisujące świat, w którym przebiega symulacja. Enum `Biome` określa możliwe typy terenu występujące na planszy: las, pustynię, wodę oraz łąkę. Struktura `Tile` odwzorowuje pojedynczy kafelek mapy i zawiera informacje o typie terenu, temperaturze, wilgotności oraz dostępności pożywienia. Struktura `World` definiuje całą planszę symulacji, przechowując jej szerokość, wysokość oraz dwuwymiarową siatkę kafelków. Obiekt `World` jest zasobem w **Bevy**, co oznacza, że jest dostępny dla wszystkich systemów, podobnie jak `Config`.

```

impl Display for Biome {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Biome::Forest => write!(f, "Forest"),
            Biome::Desert => write!(f, "Desert"),
            Biome::Water => write!(f, "Water"),
            Biome::Grassland => write!(f, "Grassland"),
        }
    }
}

impl Tile {
    pub fn regenerate_food(&mut self, config: &Config) {
        match self.biome {
            Biome::Forest => {
                if self.food_availability > config.forest.max_food_availability {
                    return;
                }

                self.food_availability += config.forest.food_availability;
            }
            Biome::Desert => {
                if self.food_availability > config.desert.max_food_availability {
                    return;
                }

                self.food_availability += config.desert.food_availability;
            }
            Biome::Grassland => {
                if self.food_availability > config.grassland.max_food_availability {
                    return;
                }

                self.food_availability += config.grassland.food_availability;
            }
            _ => {}
        }
    }
}

```

Listing 4: Implementacje metod dla struktur Biome i Tile

```

impl World {
    pub fn new(width: usize, height: usize, random_seed: u64) -> Self {
        let mut rng = StdRng::seed_from_u64(random_seed);
        let seed = rng.gen::<u32>();

        let perlin = Perlin::new(seed);
        let scale = 10.0;

        let mut grid = vec![vec![]; height];
        for y in 0..height {
            for x in 0..width {
                let noise_value = perlin.get([x as f64 / scale, y as f64 / scale]);

                let biome = if noise_value < -0.3 {
                    Biome::Water
                } else if noise_value < -0.1 {
                    Biome::Desert
                } else if noise_value < 0.5 {
                    Biome::Grassland
                } else {
                    Biome::Forest
                };

                grid[y].push(Tile {
                    biome,
                    temperature: 20.0,
                    humidity: 0.5,
                    food_availability: rng.gen_range(1.0..100.0),
                });
            }
        }

        Self {
            width,
            height,
            grid,
        }
    }
}

impl Default for World {
    fn default() -> Self {
        Self::new(10, 10, 0)
    }
}

```

Listing 5: Implementacje metod dla struktury World

W listingach 4 i 5 przedstawione są implementacje metod dla struktur Biome, Tile oraz World. Metoda Display dla Biome pozwala na wyświetlenie nazwy terenu w bardziej przyjazny dla użytkownika sposób, co oznacza, że zamiast tekstu Biome::Forest pojawi się po prostu Forest. Metoda regenerate_food dla Tile pozwala na odnowienie pożywienia na danym kafelku w zależności od rodzaju terenu. Jeśli zostanie osiągnięty limit dla danego terenu, żywność nie jest odnawiana. Dla wody jedzenie nie jest odnawiane, ponieważ w mojej implementacji założyłem, że woda nie jest terenem, na którym mogą żyć jednostki. Metoda new dla World tworzy nowy świat o podanej szerokości i wysokości, generując teren z wykorzystaniem szumu **Perlin’a** [8] oraz ziarna generatora liczb losowych. Metoda Default dla World tworzy domyślny świat o szerokości 10 i wysokości 10 z ziarnem 0. Dzięki wykorzystaniu szumu **Perlin’a** teren jest bardziej naturalny i przypomina rzeczywisty świat, a rodzaj terenu na danym kafelku zależy od wartości szumu. Dla wartości szumu mniejszych niż -0.3 teren jest wodny, dla mniejszych niż -0.1 pustynny, dla wartości mniejszych niż 0.5 łąkowy, a dla pozostałych

leśny. Każdy kafelek posiada również losowe wartości temperatury, wilgotności oraz dostępności pożywienia.

```
#[derive(Component, Serialize, Clone)]
pub struct Organism {
    pub energy: f32,
    pub speed: f32,
    pub size: f32,
    pub reproduction_threshold: f32,
    pub biome_tolerance: HashMap<Biome, f32>,
}

#[derive(Component, Serialize, Copy, Clone)]
pub struct Predator {
    pub energy: f32,
    pub speed: f32,
    pub size: f32,
    pub reproduction_threshold: f32,
    pub hunting_efficiency: f32,
    pub satiation_threshold: f32,
}

#[derive(Component, Debug, Serialize, Copy, Clone)]
pub struct Position {
    pub x: usize,
    pub y: usize,
}

#[derive(Component)]
pub struct TileComponent {
    pub biome: Biome,
}
```

Listing 6: Komponenty w symulacji

W listingu 6 przedstawione są struktury reprezentujące jednostki. Struktura `Organism` reprezentuje ofiarę i przechowuje informacje o energii, prędkości, rozmiarze, progu rozmnażania oraz tolerancji terenu. Struktura `Predator` reprezentuje drapieżnika i przechowuje informacje o energii, prędkości, rozmiarze, progu rozmnażania, wydajności polowania oraz progu głodu. Struktura `Position` zawiera informacje o pozycji jednostki na planszy. `TileComponent` przechowuje informacje o rodzaju terenu na danym kafelku. Ten komponent jest wykorzystywany do wyświetlania odpowiedniego koloru kafelka w zależności od rodzaju terenu na planszy. Wszystkie te struktury są **komponentami** w ECS i są dostępne dla systemów.

```
#[derive(Default, Resource, Serialize)]
pub struct Generation(pub usize);

const TILE_SIZE_IN_PIXELS: f32 = 32.0;
```

Listing 7: Zasób przechowujący informacje o pokoleniu oraz stała przechowująca rozmiar kafelka w pikselach

W powyższym listingu przedstawiony jest zasób `Generation`, który przechowuje informacje o aktualnym pokoleniu w symulacji oraz stała `TILE_SIZE_IN_PIXELS`, która przechowuje rozmiar kafelka w pikselach.

```

fn main() {
    let config = get_config();

    println!("{:?}", config);

    let headless = config.headless;
    let mut app = App::new();

    match headless {
        true => {
            app.add_plugins(MinimalPlugins);
        }
        false => {
            app.add_plugins(DefaultPlugins);
        }
    }

    app.insert_resource(World::new(config.width, config.height, config.seed))
        .insert_resource(config)
        .insert_resource(Generation(0))
        .add_systems(
            Startup,
            (
                spawn_world,
                spawn_organisms,
                spawn_predators,
                initialize_log_file,
            ),
        )
        .add_systems(Update, hunting)
        .add_systems(
            Update,
            (
                render_organisms,
                render_predators,
                organism_movement,
                predator_movement,
                organism_sync,
                predator_sync,
                despawn_dead_organisms,
                despawn_dead_predators,
                regenerate_food,
                consume_food,
                overcrowding,
                biome_adaptation,
                reproduction,
                predator_reproduction,
                increment_generation,
                log_organism_data,
                log_world_data,
                handle_camera_movement,
            ),
        )
        .after(hunting),
    )
    .run();
}

```

Listing 8: Główna funkcja programu

W listingu 8 przedstawiona jest funkcja programu, od której zaczyna się wykonywanie kodu. Na samym początku wczytywany jest plik konfiguracyjny i wypisywana jest jego treść do punktu wyjścia. Następnie pobierana jest flaga z ustawień konfiguracyjnych, która odpowiada za to czy symulacja powinna być uruchomiona w trybie bezokienkowym czy okienkowym. Zmienna `app` prze-

chowa aplikację **Bevy**. W zależności od wartości flagi, dodawane są odpowiednie wtyczki. Wtyczka `MinimalPlugins` dodaje minimalny zestaw wtyczek, który jest potrzebny do uruchomienia symulacji w trybie bezokienkowym, a `DefaultPlugins` dodaje domyślny zestaw wtyczek, który jest potrzebny do uruchomienia symulacji w trybie okienkowym. Następnie wstawiany jest zasób `World` z nowym światem, któremu przekazywane są ustawienia z pliku konfiguracyjnego, `Config` z ustawieniami konfiguracyjnymi, `Generation` z aktualnym pokoleniem. Następnie dodawane są systemy, które mają zostać wykonane w trakcie działania symulacji. Istnieją dwie kategorie systemów `Startup` i `Update`. Systemy należące do grupy `Startup` są uruchamiane tylko raz na początku symulacji, a systemy z grupy `Update` są wykonywane w każdej klatce symulacji. Systemy z grupy `Startup` odpowiadają za inicjalizację symulacji oraz potrzebnych plików do zapisywania danych, natomiast te z grupy `Update` - odpowiadają za aktualizację stanu symulacji, ruch jednostek, polowanie, rozmnażanie, zapisywanie danych, obsługę kamery oraz usuwanie martwych jednostek. Na końcu uruchamiana jest symulacja za pomocą metody `run`. Systemy są wykonywane równolegle, co pozwala na zwiększenie wydajności symulacji. Wyjątkiem jest system `hunting`, który jest wykonywany przed systemami: `render_organisms`, `render_predators`, `organism_movement`, `predator_movement`, `organism_sync`, `predator_sync`, `despawn_dead_organisms`, `despawn_dead_predators`, `regenerate_food`, `consume_food`, `overcrowding`, `biome_adaptation`, `reproduction`, `predator_reproduction`, `increment_generation`, `log_organism_data`, `log_world_data`, `handle_camera_movement`. Rozwiązanie to jest spowodowane tym, że gdy ofiara zostanie zjedzona, to musi zniknąć z planszy, przez co mogło zdarzyć się, że system odpowiedzialny za polowanie próbował zjeść ofiarę, której nie było już na planszy, co powodowało błąd krytyczny w programie. W ten sposób można uniknąć tego problemu. Negatywną konsekwencją tego rozwiązania jest to, że planer (ang. *scheduler*) musi wykonać więcej pracy i może to wpłynąć niekorzystnie na wydajność symulacji.

```

fn spawn_world(
    mut commands: Commands,
    world: Res<World>,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
) {
    let tile_size = Vec2::new(TILE_SIZE_IN_PIXELS, TILE_SIZE_IN_PIXELS);

    let shape = meshes.add(Rectangle::new(tile_size.x, tile_size.y));

    for (y, row) in world.grid.iter().enumerate() {
        for (x, tile) in row.iter().enumerate() {
            let color = match tile.biome {
                Biome::Forest => Color::hsl(120.0, 1.0, 0.1),
                Biome::Desert => Color::hsl(60.0, 1.0, 0.5),
                Biome::Water => Color::hsl(240.0, 1.0, 0.5),
                Biome::Grassland => Color::hsl(100.0, 1.0, 0.7),
            };

            commands
                .spawn((Mesh2d(shape.clone()), MeshMaterial2d(materials.add(color))))
                .insert(TileComponent {
                    biome: tile.biome.clone(),
                })
                .insert(Transform {
                    translation: Vec3::new(x as f32 * tile_size.x, y as f32 *
tile_size.y, 0.0),
                    ..Default::default()
                });
        }
    }

    let center_x = world.width as f32 * TILE_SIZE_IN_PIXELS / 2.0;
    let center_y = world.height as f32 * TILE_SIZE_IN_PIXELS / 2.0;

    commands.spawn((
        Camera2d::default(),
        Transform::from_xyz(center_x, center_y, 10.0),
    ));
}

```

Listing 9: System tworzący świat

W listingu 9 przedstawiony jest system `spawn_world`, który odpowiada za stworzenie świata na podstawie danych z zasobu `World`. Dla każdego kafelka na planszy dobierany jest odpowiedni kolor, w zależności od rodzaju terenu. Następnie tworzony jest kafelek na planszy z odpowiednim kolorem i pozycją. Na końcu dodawana jest kamera, która śledzi planszę. Kamera znajduje się w środku planszy i ma wysokość 10.0 jednostek.

```

fn spawn_organisms(mut commands: Commands, world: Res<World>, config: Res<Config>) {
    let mut rng = StdRng::seed_from_u64(config.seed);
    let organism_count = config.initial_organisms;

    for _ in 0..organism_count {
        let x = rng.gen_range(0..world.width);
        let y = rng.gen_range(0..world.height);

        let tile_biome = &world.grid[y][x].biome;

        let biome_tolerance = get_biome_tolerance(tile_biome, config.seed);

        commands.spawn((
            Organism {
                energy: config.initial_organism_energy,
                speed: config.initial_organism_speed,
                size: config.initial_organism_size,
                reproduction_threshold:
config.initial_organism_reproduction_threshold,
                biome_tolerance,
            },
            Position { x, y },
        ));
    }
}

```

Listing 10: System tworzący ofiary

```

fn get_biome_tolerance(tile_biome: &Biome, seed: u64) -> HashMap<Biome, f32> {
    let mut biome_tolerance = HashMap::new();
    let mut rng = StdRng::seed_from_u64(seed);

    for biome in &[Biome::Forest, Biome::Desert, Biome::Water, Biome::Grassland] {
        let tolerance = if *biome == *tile_biome {
            rng.gen_range(1.0..1.5)
        } else {
            rng.gen_range(0.1..0.8)
        };

        biome_tolerance.insert(biome.clone(), tolerance);
    }

    biome_tolerance
}

```

Listing 11: Funkcja generująca tolerancję terenu dla ofiary

Listing 10 przedstawia system `spawn_organisms`, który odpowiada za stworzenie ofiar na planszy. Dla każdej ofiary losowana jest pozycja na planszy. Następnie dodawana jest tolerancja terenu w zależności od rodzaju terenu, na którym znajduje się ofiara. Wyższa wartość tolerancji terenu oznacza lepsze przystosowanie do niego. Im wartość tolerancji terenu bliższe 0, tym ofiara gorzej przystosowana jest do danego terenu. Tolerancja terenu jest losową liczbą z przedziału $[0.1, 0.8]$ dla terenów, na których nie znajduje się ofiara oraz z przedziału $[1.0, 1.5]$ dla terenu, na którym ofiara się znajduje. Rozwiązanie te ma odzwierciedlać lepsze przystosowanie jednostek do terenu, na którym żyją.


```

fn spawn_predators(mut commands: Commands, world: Res<World>, config: Res<Config>) {
    let mut rng = StdRng::seed_from_u64(config.seed);
    let predator_count = config.initial_predators;

    for _ in 0..predator_count {
        let x = rng.gen_range(0..world.width);
        let y = rng.gen_range(0..world.height);

        commands.spawn((
            Predator {
                energy: config.initial_predator_energy,
                speed: config.initial_predator_speed,
                size: config.initial_predator_size,
                reproduction_threshold:
config.initial_predator_reproduction_threshold,
                hunting_efficiency: config.initial_predator_hunting_efficiency,
                satiation_threshold: config.initial_predator_satiation_threshold,
            },
            Position { x, y },
        ));
    }
}

```

Listing 12: System tworzący drapieżniki

Listing 12 przedstawia system `spawn_predators`, który odpowiada za stworzenie drapieżników na planszy. Dla każdego drapieżnika losowana jest pozycja na planszy. Następnie jest on dodawany do planszy z odpowiednimi parametrami początkowymi.

```

fn render_organisms(
    mut commands: Commands,
    query: Query<(Entity, &Position), (Without<Predator>, Without<Mesh2d>)>,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
) {
    let tile_size = Vec2::new(TILE_SIZE_IN_PIXELS, TILE_SIZE_IN_PIXELS);
    let organism_size = Vec2::new(16.0, 16.0);

    let shape = meshes.add(Circle::new((organism_size.x) / 2.0));

    let color = Color::linear_rgb(0.0, 155.0, 12.0);

    for (entity, position) in query.iter() {
        commands.entity(entity).insert((
            Mesh2d(shape.clone()),
            MeshMaterial2d(materials.add(color)),
            Transform::from_xyz(
                position.x as f32 * tile_size.x,
                position.y as f32 * tile_size.y,
                1.0,
            ),
        ));
    }
}

```

Listing 13: System wyświetlający ofiary

```

fn render_predators(
    mut commands: Commands,
    query: Query<(Entity, &Position), (Without<Organism>, Without<Mesh2d>)>,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
) {
    let tile_size = Vec2::new(TILE_SIZE_IN_PIXELS, TILE_SIZE_IN_PIXELS);
    let organism_size = Vec2::new(16.0, 16.0);

    let color = Color::srgb(255.0, 0.0, 0.0);
    let shape = meshes.add(Rectangle::new(organism_size.x, organism_size.y));

    for (entity, position) in query.iter() {
        commands.entity(entity).insert((
            Mesh2d(shape.clone()),
            MeshMaterial2d(materials.add(color)),
            Transform::from_xyz(
                position.x as f32 * tile_size.x,
                position.y as f32 * tile_size.y,
                1.0,
            ),
        ));
    }
}

```

Listing 14: System wyświetlający drapieżniki

Listingi 13 i 14 przedstawiają systemy `render_organisms` i `render_predators`, które odpowiadają za wyświetlenie ofiar i drapieżników na planszy. Ofiary są wyświetlane jako niebieskie koła, a drapieżniki jako czerwone prostokąty. Wymienione systemy korzystają ze składni i mechanizmów charakterystycznych dla architektury ECS, takich jak struktura `Query`, która przechowuje zbiór encji, które spełniają określone kryteria. W tym przypadku zwraca encje, które dla systemu `render_organisms` nie posiadają komponentu `Predator` oraz `Mesh2d`, a dla systemu `render_predators` nie posiadają komponentu `Organism` oraz `Mesh2d`, natomiast posiadają komponent `Position`. Dzięki temu system otrzymuje encje, które zawierają tylko te dane, które są mu potrzebne do działania.

```

fn organism_movement(
    mut query: Query<(&mut Position, &mut Organism)>,
    world: Res<World>,
    config: Res<Config>,
) {
    let directions: Vec<(i32, i32)> = vec![
        (-1, -1),
        (0, -1),
        (1, -1),
        (-1, 0),
        (1, 0),
        (-1, 1),
        (0, 1),
        (1, 1),
    ];

    let mut rng = StdRng::seed_from_u64(config.seed);

    for (mut position, mut organism) in query.iter_mut() {
        if organism.energy <= 0.0 {
            continue;
        }

        let mut best_direction = (0, 0);
        let mut best_cost = f32::MAX;
    }
}

```

```

        for (dx, dy) in directions.iter() {
            let new_x = (position.x as isize + dx).clamp(0, (world.width - 1) as
isize) as usize;
            let new_y = (position.y as isize + dy).clamp(0, (world.height - 1) as
isize) as usize;
            let tile = &world.grid[new_y][new_x];

            let base_cost = match tile.biome {
                Biome::Water => 100.0,
                Biome::Desert => 50.0,
                Biome::Grassland => 10.0,
                Biome::Forest => 20.0,
            };

            let tolerance =
organism.biome_tolerance.get(&tile.biome).unwrap_or(&1.0);
            let cost = base_cost / tolerance;

            let cost = cost + rng.gen_range(0.0..5.0);

            if cost < best_cost {
                best_cost = cost;
                best_direction = (*dx, *dy);
            }
        }

        position.x =
            (position.x as isize + best_direction.0).clamp(0, (world.width - 1) as
isize) as usize;
        position.y =
            (position.y as isize + best_direction.1).clamp(0, (world.height - 1) as
isize) as usize;

        let energy_to_consume = 0.1 * organism.speed * organism.size;

        organism.energy -= energy_to_consume;

        let tile = &world.grid[position.y][position.x];
        if tile.biome == Biome::Water {
            organism.energy = -1.0;
        }
    }
}

```

Listing 15: System odpowiedzialny za ruch ofiar

W listingu 15 przedstawiony jest system `organism_movement`, który odpowiada za ruch ofiar na planszy. Dla każdej ofiary losowana jest nowa pozycja na planszy. Następnie obliczany jest najlepszy kierunek ruchu na podstawie kosztu ruchu. Koszt ruchu zależy od rodzaju terenu, na którym znajduje się ofiara oraz tolerancji terenu. Im większa tolerancja terenu, tym mniejszy koszt ruchu. Koszt ruchu jest losowany z przedziału $[0.0, 5.0]$. Następnie obliczana jest nowa pozycja ofiary na planszy, a także zużyta energia na podstawie prędkości i rozmiaru ofiary. Jeśli ofiara znajduje się na wodzie, to energia przyjmuje wartość -1.0 , co oznacza, że ofiara umiera. W ten sposób ofiary są bardziej przystosowane do terenu, na którym się znajdują. Możliwych jest osiem kierunków ruchu: góra, dół, lewo, prawo oraz po skosach. Energia ofiary zmniejsza się w zależności od prędkości i rozmiaru. Im większa prędkość i rozmiar ofiary, tym więcej energii zużywa na ruch.

```

fn predator_movement(
    mut predator_query: Query<(&mut Position, &mut Predator)>,
    prey_query: Query<(&Position, &Organism), Without<Predator>>,
    world: Res<World>,

```

```

    config: Res<Config>,
) {
    let directions: Vec<(isize, isize)> = vec![
        (-1, -1),
        (0, -1),
        (1, -1),
        (-1, 0),
        (1, 0),
        (-1, 1),
        (0, 1),
        (1, 1),
    ];

    let mut rng = StdRng::seed_from_u64(config.seed);

    for (mut predator_position, mut predator) in predator_query.iter_mut() {
        if predator.energy <= 0.0 {
            continue;
        }

        let mut closest_prey: Option<&Position> = None;
        let mut min_distance = f32::MAX;
        let predator_range_attack = 1.0;

        for (prey_position, _) in prey_query.iter() {
            let dx = predator_position.x as f32 - prey_position.x as f32;
            let dy = predator_position.y as f32 - prey_position.y as f32;
            let distance = dx * dx + dy * dy;

            if distance < min_distance && distance <= predator_range_attack {
                min_distance = distance;
                closest_prey = Some(prey_position);
            }
        }

        if let Some(prey_position) = closest_prey {
            let dx = prey_position.x as isize - predator_position.x as isize;
            let dy = prey_position.y as isize - predator_position.y as isize;

            predator_position.x = (predator_position.x as isize + dx.signum())
                .clamp(0, (world.width - 1) as isize) as usize;
            predator_position.y = (predator_position.y as isize + dy.signum())
                .clamp(0, (world.height - 1) as isize) as usize;
        } else {
            let mut best_direction = (0, 0);
            let mut best_cost = f32::MAX;

            for (dx, dy) in directions.iter() {
                let new_x = (predator_position.x as isize + dx).clamp(0, (world.width
- 1) as isize)
                    as usize;
                let new_y = (predator_position.y as isize + dy)
                    .clamp(0, (world.height - 1) as isize) as usize;

                let tile = &world.grid[new_y][new_x];

                let cost = match tile.biome {
                    Biome::Water => 100.0,
                    Biome::Desert => 10.0,
                    Biome::Grassland => 5.0,
                    Biome::Forest => 6.0,
                };
            }
        }
    }
}

```

```

        let cost = cost + rng.gen_range(0.0..5.0);

        if cost < best_cost {
            best_cost = cost;
            best_direction = (*dx, *dy);
        }
    }

    predator_position.x = (predator_position.x as isize + best_direction.0)
        .clamp(0, (world.width - 1) as isize) as usize;
    predator_position.y = (predator_position.y as isize + best_direction.1)
        .clamp(0, (world.height - 1) as isize) as usize;
}

predator.energy -= 0.1 * predator.speed * predator.size;
}
}

```

Listing 16: System odpowiedzialny za ruch drapieżników

System `predator_movement` przedstawiony w listingu 16 odpowiada za ruch drapieżników na planszy. Dla każdego drapieżnika obliczam najbliższą ofiarę (na podstawie odległości euklidesowej). Jeśli ofiara znajduje się w zasięgu ataku drapieżnika, to drapieżnik rusza w jej kierunku. W przeciwnym razie drapieżnik rusza w losowym kierunku. Ruch drapieżnika jest dozwolony w 8 kierunkach: góra, dół, lewo, prawo oraz po skosach. Energia drapieżnika zmniejsza się w zależności od prędkości i rozmiaru drapieżnika. Im większe wartości tych cech, tym więcej energii zużywa na ruch. Jeśli drapieżnik znajdzie się na wodzie, to energia przyjmuje wartość -1.0 , co oznacza, że drapieżnik umiera.

W obu systemach `organism_movement` i `predator_movement` następuje sprawdzenie, czy dana jednostka posiada dodatnią energię, by w ogóle mogła się ruszać.

```

fn despawn_dead_organisms(mut commands: Commands, query: Query<(Entity, &Organism)>)
{
    for (entity, organism) in query.iter() {
        if organism.energy <= 0.0 {
            commands.entity(entity).despawn_recursive();
        }
    }
}

fn despawn_dead_predators(mut commands: Commands, query: Query<(Entity, &Predator)>)
{
    for (entity, predator) in query.iter() {
        if predator.energy <= 0.0 {
            commands.entity(entity).despawn_recursive();
        }
    }
}

```

Listing 17: Systemy usuwające martwe jednostki

W listingu 17 przedstawione są systemy `despawn_dead_organisms` i `despawn_dead_predators`, które odpowiadają za usuwanie martwych jednostek z planszy. Dla każdej jednostki sprawdzam, czy jej energia jest mniejsza lub równa 0.0 . W takim przypadku usuwam jednostkę z planszy.

```

fn organism_sync(mut query: Query<(&Position, &mut Transform, &Organism)>) {
    for (position, mut transform, organism) in query.iter_mut() {
        transform.translation.x = position.x as f32 * TILE_SIZE_IN_PIXELS;
        transform.translation.y = position.y as f32 * TILE_SIZE_IN_PIXELS;
        transform.scale = Vec3::new(organism.size, organism.size, 1.0);
    }
}

```

```
fn predator_sync(mut query: Query<(&Position, &mut Transform, &Predator)>) {
    for (position, mut transform, predator) in query.iter_mut() {
        transform.translation.x = position.x as f32 * TILE_SIZE_IN_PIXELS;
        transform.translation.y = position.y as f32 * TILE_SIZE_IN_PIXELS;
        transform.scale = Vec3::new(predator.size, predator.size, 1.0);
    }
}
```

Listing 18: Systemy synchronizujące pozycję jednostek

W listingu 18 przedstawione są systemy `organism_sync` i `predator_sync`, które odpowiadają za synchronizację pozycji jednostek na planszy. Dla każdej jednostki jest ustawiana odpowiednia pozycja oraz skala.

```
fn regenerate_food(mut world: ResMut<World>, config: Res<Config>) {
    for row in world.grid.iter_mut() {
        for tile in row.iter_mut() {
            tile.regenerate_food(&config);
        }
    }
}
```

Listing 19: System odnawiający pożywienie na planszy

W listingu 19 przedstawiony jest system `regenerate_food`, który odpowiada za odnowienie pożywienia na planszy. Dla każdego kafelka na planszy pożywienie jest odnawiane na podstawie ustawień konfiguracyjnych i zależy od rodzaju terenu. Im bardziej żyzny teren, tym więcej pożywienia jest odnawiane.

```
fn consume_food(mut world: ResMut<World>, mut query: Query<(Entity, &mut Organism, &Position)>) {
    let mut organisms_by_tile: HashMap<(usize, usize), Vec<(Entity, Mut<Organism>)>>
    = HashMap::new();

    for (entity, organism, position) in query.iter_mut() {
        organisms_by_tile
            .entry((position.x, position.y))
            .or_default()
            .push((entity, organism));
    }

    for ((x, y), organisms) in organisms_by_tile.iter_mut() {
        let tile = &mut world.grid[*y][*x];
        if tile.food_availability < 0.0 {
            continue;
        }

        organisms.sort_by(|a, b| {
            b.1.size
                .partial_cmp(&a.1.size)
                .unwrap_or(std::cmp::Ordering::Equal)
        });

        let mut remaining_food = tile.food_availability;
        for (_, organism) in organisms.iter_mut() {
            if remaining_food <= 0.0 {
                break;
            }

            let food_needed = organism.size * 0.2 * organism.speed;

            let food_consumed = food_needed.min(remaining_food);
            remaining_food -= food_consumed;
        }
    }
}
```

```

        organism.energy += food_consumed * 2.0;

        tile.food_availability -= food_consumed;
    }
}

```

Listing 20: System spożywania dla ofiar

W listingu 20 przedstawiony jest system `consume_food`, który odpowiada za spożywanie jedzenia przez ofiary. Na początku zbierane są ofiary na danym kafelku. Następnie sprawdzenie czy jest dostępne jedzenie. Jeśli go nie ma, to rozpatrywany jest następny kafelek. Następnie wykonywane jest sortowanie ofiar od największego do najmniejszego. Dla każdej ofiary obliczam, ile jedzenia potrzebuje, na podstawie rozmiaru oraz prędkości jednostki. Od ilości obecnego jedzenia odejmowana jest ilość jedzenia spożytego przez daną ofiarę. Następnie dodawana jest energia ofiarze na podstawie zjedzonego jedzenia pomnożonego przez wartość dobraną w sposób arbitralny - 2.0. Jednostka nie może zjeść więcej jedzenia, niż jest dostępne na danym kafelku.

```

fn biome_adaptation(mut query: Query(&mut Organism, &Position)>, world: Res<World>)
{
    for (mut organism, position) in query.iter_mut() {
        let tile = &world.grid[position.y][position.x];
        let tolerance = organism.biome_tolerance.get(&tile.biome).unwrap_or(&1.0);

        match tile.biome {
            Biome::Forest => {
                organism.energy += 0.1 * tolerance;
            }
            Biome::Desert => {
                organism.energy -= 0.1 / tolerance;
            }
            Biome::Water => {
                organism.energy -= f32::MAX;
            }
            Biome::Grassland => {
                organism.energy += 0.05 * tolerance;
            }
        }
    }
}

```

Listing 21: System adaptacji do terenu dla ofiar

W listingu 21 przedstawiony jest system `biome_adaptation`, który odpowiada za adaptację ofiar do terenu. W zależności od rodzaju terenu, na którym znajduje się ofiara, energia ofiary zmniejsza się lub zwiększa. Lasy są bogate w jedzenie, więc znajdujące się na tym terenie jednostki zyskują energię. Pustynie są ubogie w jedzenie, więc ofiary tracą energię. Woda nie jest dobrym miejscem dla ofiar, więc jednostki tracą całą energię. Jednostki znajdujące się na łąkach zyskują energię. Tolerancja terenu wpływa na to, jak bardzo ofiara jest do niego przystosowana, co ma wpływ na zwiększenie bądź zmniejszenie ilości zyskanej lub utraconej energii.

```

fn reproduction(
    mut commands: Commands,
    mut query: Query<(&mut Organism, &Position)>,
    predators_query: Query<&Predator>,
    world: Res<World>,
    config: Res<Config>,
) {
    let organisms_count = query.iter().count();
    let predators_count = predators_query.iter().count();
    let total_entities = organisms_count + predators_count;

    if total_entities >= config.max_total_entities {
        return;
    }

    let mut rng = StdRng::seed_from_u64(config.seed);

    for (mut organism, position) in query.iter_mut() {
        if organism.energy > organism.reproduction_threshold {
            let mutation_factor = config.organism_mutability;

            let tile_biome = &world.grid[position.y][position.x].biome;

            let mut biome_tolerance = get_biome_tolerance(tile_biome, config.seed);
            for (_, tolerance) in biome_tolerance.iter_mut() {
                *tolerance *= 1.0 + rng.gen_range(-mutation_factor..mutation_factor);
            }

            let reproduction_threshold = organism.reproduction_threshold
                * (1.0 + rng.gen_range(-mutation_factor..mutation_factor));

            let size = organism.size * (1.0 + rng.gen_range(-
mutation_factor..mutation_factor));
            let speed = (organism.speed * (1.1 + rng.gen_range(-
mutation_factor..mutation_factor)))
                - (size * 0.1);

            let child = Organism {
                energy: organism.energy / 2.0,
                speed: speed,
                size: size,
                reproduction_threshold,
                biome_tolerance,
            };

            let x_offset = rng.gen_range(-1..=1);
            let y_offset = rng.gen_range(-1..=1);

            let child_position = Position {
                x: (position.x as isize + x_offset).clamp(0, world.width as isize -
1) as usize,
                y: (position.y as isize + y_offset).clamp(0, world.height as isize -
1) as usize,
            };

            commands.spawn((child, child_position));

            organism.energy /= 2.0;
        }
    }
}

```

Listing 22: System rozmnażania dla ofiar

W listingu 22 przedstawiony jest system reproduction, który odpowiada za rozmnażanie ofiar. Dla każdej ofiary następuje sprawdzenie, czy jej energia jest większa od progu reprodukcji. Jeśli tak, to tworzony jest nowy potomek. Nowo powstałe osobniki mogą ulec mutacji. Współczynnik odpowiadający za mutację jest losowany z przedziału $[-mutation_factor, mutation_factor]$. Potomek posiada połowę energii rodzica, a prędkość, rozmiar oraz próg reprodukcji są modyfikowane w zależności od współczynnika mutacji. Na końcu energia rodzica jest zmniejszona o połowę.

Jeśli liczba ofiar i drapieżników przekroczy maksymalną liczbę jednostek na planszy, to nie będą mogły się rozmnażać.

```
fn hunting(
    mut commands: Commands,
    mut predator_query: Query<(&mut Predator, &Position)>,
    prey_query: Query<(Entity, &Position, &Organism), Without<Predator>>,
) {
    for (mut predator, predator_position) in predator_query.iter_mut() {
        if predator.energy >= predator.satiation_threshold {
            continue;
        }

        for (prey_entity, prey_position, prey) in prey_query.iter() {
            if predator_position.x == prey_position.x && predator_position.y ==
prey_position.y {
                let energy_gained = prey.size * predator.hunting_efficiency;
                predator.energy += energy_gained;
                commands.entity(pre_entity).try_despawn_recursive();
                break;
            }
        }
    }
}
```

W listingu 23 przedstawiony jest system hunting odpowiedzialny za polowanie drapieżników na ofiary. Dla każdego drapieżnika jest sprawdzenie, czy jego energia jest mniejsza od progu sytości oraz określone jest, czy drapieżnik znajduje się na tym samym kafelku co ofiara. Jeśli tak, to drapieżnik zjada ofiarę i zyskuje energię na podstawie rozmiaru ofiary oraz efektywności polowania drapieżnika. Następnie ofiara jest usuwana z planszy. Dzięki temu drapieżniki mogą zyskiwać energię.

```
fn predator_reproduction(
    mut commands: Commands,
    mut query: Query<(&mut Predator, &Position)>,
    organisms_query: Query<&Organism>,
    world: Res<World>,
    config: Res<Config>,
) {
    let predators_count = query.iter().count();
    let organisms_count = organisms_query.iter().count();
    let total_entities = predators_count + organisms_count;

    if total_entities >= config.max_total_entities {
        return;
    }

    let mut rng = StdRng::seed_from_u64(config.seed);

    for (mut predator, position) in query.iter_mut() {
        if predator.energy > predator.reproduction_threshold {
            let mutation_factor = config.predator_mutability;

            let size = predator.size * (1.0 + rng.gen_range(-
```

```

mutation_factor..mutation_factor));
    let speed = predator.speed * (1.1 + rng.gen_range(-
mutation_factor..mutation_factor))
    - (size * 0.1);

    let child = Predator {
        energy: predator.energy / 2.0,
        speed: speed,
        size: size,
        hunting_efficiency: predator.hunting_efficiency
            * (1.0 + rng.gen_range(-mutation_factor..mutation_factor)),
        satiation_threshold: predator.satiation_threshold
            * (1.0 + rng.gen_range(-mutation_factor..mutation_factor)),
        reproduction_threshold: predator.reproduction_threshold
            * (1.0 + rng.gen_range(-mutation_factor..mutation_factor)),
    };

    let x_offset = rng.gen_range(-1..=1);
    let y_offset = rng.gen_range(-1..=1);

    let child_position = Position {
        x: (position.x as isize + x_offset).clamp(0, world.width as isize -
1) as usize,
        y: (position.y as isize + y_offset).clamp(0, world.height as isize -
1) as usize,
    };

    commands.spawn((child, child_position));

    predator.energy /= 2.0;
}
}
}

```

Listing 24: System rozmnażania dla drapieżników

System rozmnażania dla drapieżników przedstawiony w listingu 24 działa podobnie do systemu rozmnażania dla ofiar.

```

fn overcrowding(
    mut query: Query<(&mut Organism, &Position)>,
    mut predator_query: Query<(&mut Predator, &Position)>,
    config: Res<Config>,
) {
    let overcrowding_threshold_for_organisms =
config.overcrowding_threshold_for_organisms;
    let overcrowding_threshold_for_predators =
config.overcrowding_threshold_for_predators;

    let mut organisms_by_tile: HashMap<(usize, usize), Vec<Mut<Organism>>> =
HashMap::new();

    for (organism, position) in query.iter_mut() {
        organisms_by_tile
            .entry((position.x, position.y))
            .or_default()
            .push(organism);
    }

    for (_, organisms) in organisms_by_tile.iter_mut() {
        if organisms.len() > overcrowding_threshold_for_organisms {
            organisms.sort_by(|a, b| {
                a.energy

```

```

        .partial_cmp(&b.energy)
        .unwrap_or(std::cmp::Ordering::Equal)
    });

    let num_to_remove = organisms.len() -
overcrowding_threshold_for_organisms;
    for organism in organisms.iter_mut().take(num_to_remove) {
        organism.energy = -1.0;
    }
}

let mut predators_by_tile: HashMap<(usize, usize), Vec<Mut<Predator>>> =
HashMap::new();

for (predator, position) in predator_query.iter_mut() {
    predators_by_tile
        .entry((position.x, position.y))
        .or_default()
        .push(predator);
}

for (_, predators) in predators_by_tile.iter_mut() {
    if predators.len() > overcrowding_threshold_for_predators {
        predators.sort_by(|a, b| {
            a.energy
                .partial_cmp(&b.energy)
                .unwrap_or(std::cmp::Ordering::Equal)
        });

        let num_to_remove = predators.len() -
overcrowding_threshold_for_predators;
        for predator in predators.iter_mut().take(num_to_remove) {
            predator.energy = -1.0;
        }
    }
}
}

```

Listing 25: System przeludnienia

System przeludnienia przedstawiony w listingu 25 odpowiada za usuwanie jednostek z planszy, gdy przekroczona zostanie ich maksymalna liczba na kafelku. Jeśli liczba ofiar na kafelku przekracza próg przeludnienia, jednostki są sortowane od najmniejszej energii do największej. Następnie organizmy o najmniejszej energii są usuwane.

```

fn increment_generation(mut generation: ResMut<Generation>) {
    generation.0 += 1;
}

```

Listing 26: System inkrementacji numeru pokolenia

W listingu 26 przedstawiony jest system `increment_generation`, który odpowiada za inkrementację numeru pokolenia.

```

fn initialize_log_file(config: Res<Config>) {
    if !config.log_data {
        return;
    }

    let world_file = File::create("world_data.jsonl").expect("Failed to create log
file");
    world_file.set_len(0).expect("Failed to clear log file");
}

```

Listing 27: System inicjalizacji pliku danych

W listingu 27 przedstawiony jest system `initialize_log_file`, który odpowiada za inicjalizację pliku danych symulacji. Jeśli w konfiguracji ustawiono, że dane mają być zapisywane do pliku, to tworzony jest pusty plik `world_data.jsonl`.

```
fn log_world_data(
    config: Res<Config>,
    world: Res<World>,
    generation: Res<Generation>,
    organisms_query: Query<(&Organism, &Position)>,
    predators_query: Query<(&Predator, &Position)>,
) {
    if !config.log_data {
        return;
    }

    let mut file = OpenOptions::new()
        .create(true)
        .append(true)
        .open("world_data.jsonl")
        .expect("Failed to open log file");

    let organisms_with_position = organisms_query
        .iter()
        .map(|(organism, position)| OrganismWithPosition {
            organism: organism.clone(),
            position: position.clone(),
        })
        .collect::<Vec<_>>();

    let predators_with_position = predators_query
        .iter()
        .map(|(predator, position)| PredatorWithPosition {
            predator: predator.clone(),
            position: position.clone(),
        })
        .collect::<Vec<_>>();

    let export_data = ExportData {
        generation: generation.0,
        world: world.clone(),
        config: config.clone(),
        organisms: organisms_with_position,
        predators: predators_with_position,
    };

    let json = serde_json::to_string(&export_data).expect("Failed to serialize
data");

    writeln!(file, "{}", json).expect("Failed to write to log file");
}
```

Listing 28: System zapisu danych do pliku

W listingu 28 przedstawiony jest system `log_world_data`, który odpowiada za zapis danych symulacji. Jeśli w konfiguracji ustawiono, że dane mają być zapisywane, to otwierany jest plik `world_data.jsonl` w trybie dodawania i tworzenia pliku, a następnie zapisywane do niego są dane symulacji takie jak numer pokolenia, stan planszy, ustawienia konfiguracyjne, ofiary oraz drapieżniki na planszy. Dane są zapisywane w formacie JSON.

```
fn handle_camera_movement(
    mut query: Query<(&mut Transform, &Camera)>,
```

```

        keys: Res<ButtonInput<KeyCode>>,
    ) {
        for (mut transform, _) in query.iter_mut() {
            let mut translation = transform.translation;

            if keys.pressed(KeyCode::KeyW) {
                translation.y += 5.0;
            }
            if keys.pressed(KeyCode::KeyS) {
                translation.y -= 5.0;
            }
            if keys.pressed(KeyCode::KeyA) {
                translation.x -= 5.0;
            }
            if keys.pressed(KeyCode::KeyD) {
                translation.x += 5.0;
            }

            transform.translation = translation;
        }
    }
}

```

Listing 29: System obsługujący ruch kamery

System `handle_camera_movement` odpowiada za obsługę ruchu kamery przy pomocy klawiszy WSAD.

```

fn load_config() -> Result<Config, Box<dyn Error>> {
    let exe_dir = std::env::current_exe()
        .expect("Failed to get current executable path")
        .parent()
        .expect("Executable must be in a directory")
        .to_path_buf();

    let config_path = exe_dir.join("config.toml");

    let config = fs::read_to_string(config_path)?;
    let config: Config = toml::from_str(&config)?;

    Ok(config)
}

```

```

fn default_config() -> Config {
    Config {
        width: 10,
        height: 10,
        initial_organisms: 10,
        initial_predators: 1,
        headless: false,
        log_data: false,
        forest: BiomeDataConfig {
            food_availability: 1.0,
            max_food_availability: 100.0,
        },
        desert: BiomeDataConfig {
            food_availability: 1.0,
            max_food_availability: 100.0,
        },
        water: BiomeDataConfig {
            food_availability: 1.0,
            max_food_availability: 100.0,
        },
    }
}

```

```

    grassland: BiomeDataConfig {
        food_availability: 1.0,
        max_food_availability: 100.0,
    },
    initial_organism_energy: 100.0,
    initial_predator_energy: 100.0,
    initial_organism_speed: 1.0,
    initial_predator_speed: 1.0,
    initial_organism_size: 1.0,
    initial_predator_size: 1.0,
    initial_organism_reproduction_threshold: 100.0,
    initial_predator_reproduction_threshold: 100.0,
    initial_predator_hunting_efficiency: 1.0,
    initial_predator_satiation_threshold: 100.0,
    organism_mutability: 0.1,
    predator_mutability: 0.1,
    overcrowding_threshold_for_organisms: 10,
    overcrowding_threshold_for_predators: 10,
    seed: 0,
    max_total_entities: 1000,
}
}

fn get_config() -> Config {
    #[cfg(target_arch = "wasm32")]
    let config = default_config();
    #[cfg(not(target_arch = "wasm32"))]
    let config = load_config().expect("Failed to load config file");

    config
}

```

Listing 30: Funkcje do obsługi konfiguracji

W listingu 30 przedstawione są funkcje do obsługi konfiguracji. `load_config` wczytuje konfigurację z pliku `config.toml`, `default_config` zwraca domyślną konfigurację, jeśli nie uda się wczytać konfiguracji z pliku, `get_config` zwraca konfigurację. Jeśli program jest uruchomiony w przeglądarce, to zwracana jest domyślna konfiguracja. W przeciwnym przypadku zwracana jest wczytana konfiguracja z pliku.

5.3. Wizualizacja

W punkcie 5.3 przedstawiam kod odpowiedzialny za wizualizację oraz przetwarzanie danych. Skupię się przede wszystkim na tym, jak poradziłem sobie z wczytywaniem i analizą bardzo dużych zbiorów danych, dbając o efektywne wykorzystanie pamięci operacyjnej. Wykorzystałem język **Python**.

```

import json
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

```

Listing 31: Importowanie bibliotek

W pierwszej kolejności importowane są potrzebne biblioteki do wizualizacji danych. Biblioteka `json` służy do wczytania danych z pliku JSON, `numpy` do operacji na macierzach, `pandas` do wykonywania operacji na danych, `matplotlib` do tworzenia wykresów, a `seaborn` do tworzenia wykresów statystycznych jak wykres mapy cieplnej.

```

jsonl_file = "world_data.jsonl"
generations = []
organism_count = 0
predator_count = 0

```

```

width, height = None, None

gen_list = []
organism_counts = []
predator_counts = []
biome_counts = {"Forest": [], "Desert": [], "Water": [], "Grassland": []}

heatmap_grid = None
last_snapshot = None

organism_avg_size_list = []
organism_avg_speed_list = []
organism_avg_energy_list = []
predator_avg_size_list = []
predator_avg_speed_list = []
predator_avg_energy_list = []
organism_avg_reproduction_threshold_list = []
predator_avg_reproduction_threshold_list = []
predator_avg_hunting_efficiency_list = []
predator_avg_satiation_threshold_list = []

average_food_per_generation = []

```

Listing 32: Inicjalizacja zmiennych

W powyższym listingu inicjalizowane są zmienne, które będą potrzebne do stworzenia wykresów. Zmienna `jsonl_file` przechowuje ścieżkę do pliku z danymi. `generations`, `organism_count`, `predator_count`, `width`, `height` przechowują informacje o pokoleniach, liczbie ofiar, liczbie drapieżników, szerokości i wysokości planszy. Zmienna `gen_list` zawiera listę pokoleń, `organism_counts` i `predator_counts` przechowują listę liczby ofiar i drapieżników w każdym pokoleniu. `biome_counts` odpowiada za liczbę kafelków z danym rodzajem terenu w każdym pokoleniu. Zmienna `heatmap_grid` przechowuje mapę ciepłą, a `last_snapshot` ostatni stan planszy. Pozostałe zmienne przechowują średnie wartości cech ofiar i drapieżników w każdym pokoleniu.

```

with open(jsonl_file, 'r') as f:
    for line in f:
        if not line.strip():
            continue
        data = json.loads(line.strip())
        last_snapshot = data

        generation = data["generation"]
        generations.append(generation)
        organism_count = len(data["organisms"])
        predator_count = len(data["predators"])

        if width is None:
            width, height = data["config"]["width"], data["config"]["height"]
            heatmap_grid = np.zeros((height, width))

        gen_list.append(generation)
        organism_counts.append(organism_count)
        predator_counts.append(predator_count)

        if organism_count > 0:
            organism_avg_size_list.append(np.mean([o["organism"]["size"] for o in
data["organisms"]]))
            organism_avg_speed_list.append(np.mean([o["organism"]["speed"] for o in
data["organisms"]]))
            organism_avg_energy_list.append(np.mean([o["organism"]["energy"] for o in
data["organisms"]]))
            organism_avg_reproduction_threshold_list.append(np.mean([o["organism"]

```

```

["reproduction_threshold"] for o in data["organisms"]]))
    else:
        organism_avg_size_list.append(0)
        organism_avg_speed_list.append(0)
        organism_avg_energy_list.append(0)
        organism_avg_reproduction_threshold_list.append(0)

        if predator_count > 0:
            predator_avg_size_list.append(np.mean([p["predator"]["size"] for p in
data["predators"]]))
            predator_avg_speed_list.append(np.mean([p["predator"]["speed"] for p in
data["predators"]]))
            predator_avg_energy_list.append(np.mean([p["predator"]["energy"] for p in
data["predators"]]))
            predator_avg_reproduction_threshold_list.append(np.mean([p["predator"]
["reproduction_threshold"] for p in data["predators"]]))
            predator_avg_hunting_efficiency_list.append(np.mean([p["predator"]
["hunting_efficiency"] for p in data["predators"]]))
            predator_avg_satiation_threshold_list.append(np.mean([p["predator"]
["satiation_threshold"] for p in data["predators"]]))
        else:
            predator_avg_size_list.append(0)
            predator_avg_speed_list.append(0)
            predator_avg_energy_list.append(0)
            predator_avg_reproduction_threshold_list.append(0)
            predator_avg_hunting_efficiency_list.append(0)
            predator_avg_satiation_threshold_list.append(0)

    for org in data["organisms"]:
        x, y = org["position"]["x"], org["position"]["y"]
        heatmap_grid[y, x] += 1
    for pred in data["predators"]:
        x, y = pred["position"]["x"], pred["position"]["y"]
        heatmap_grid[y, x] += 1

    biome_tally = {"Forest": 0, "Desert": 0, "Water": 0, "Grassland": 0}
    for org in data["organisms"]:
        max_biome = max(org["organism"]["biome_tolerance"], key=org["organism"]
["biome_tolerance"].get)
        biome_tally[max_biome] += 1
    for biome in biome_counts:
        biome_counts[biome].append(biome_tally[biome])

    total_food = sum(tile["food_availability"] for row in data["world"]["grid"]
for tile in row)
    average_food_per_generation.append(total_food / (width * height))

    if len(gen_list) % 100 == 0:
        print(f"Processed {len(gen_list)} generations...")

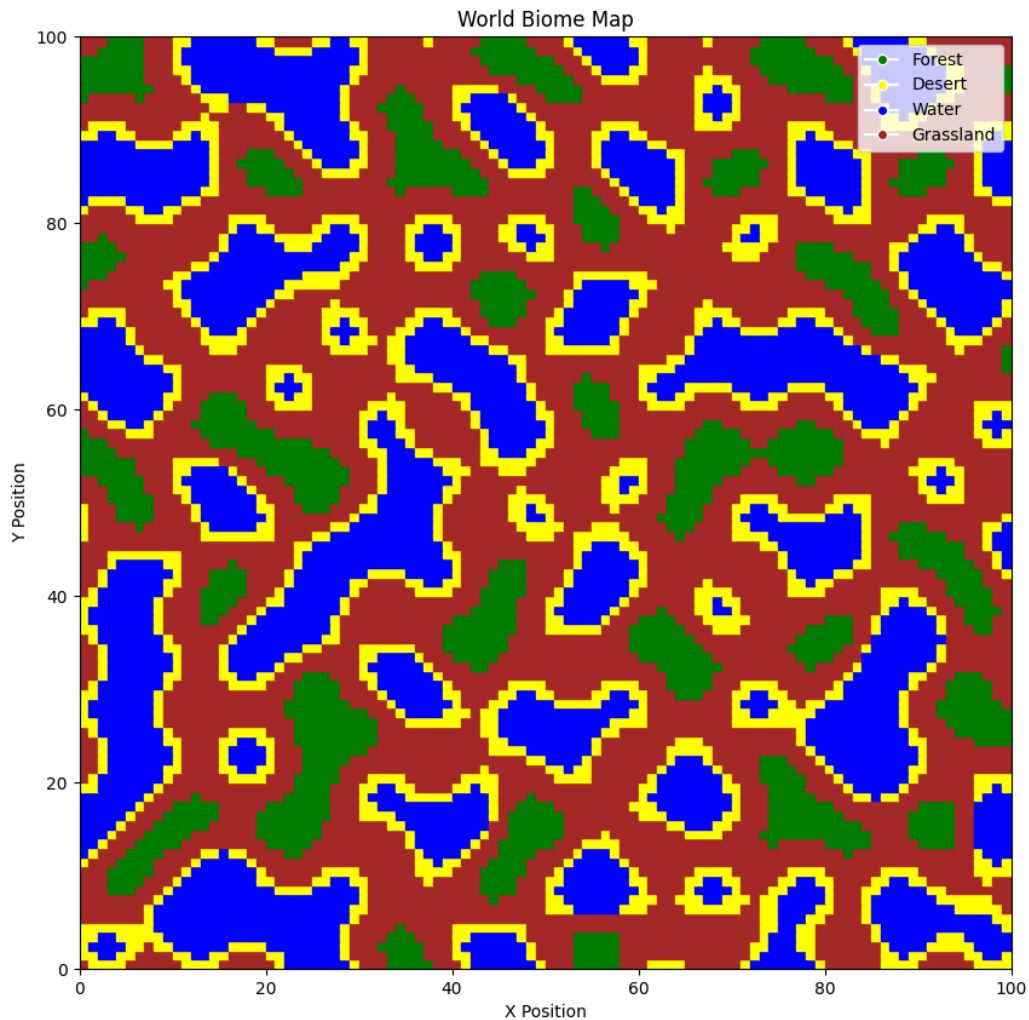
```

Listing 33: Przetwarzanie danych

Dane z pliku `world_data.jsonl` są wczytywane i przetwarzane. Istotne jest to, że dane przetwarzane są linijka po linijce, gdyż plik może być bardzo duży i w przeciwnym wypadku program mógłby zużyć dużo pamięci, której komputer nie posiada. Wczytywane są dane o pokoleniu, liczbie ofiar i drapieżników, szerokości i wysokości planszy. Następnie tworzona jest mapa cieplna planszy, a także obliczane są średnie wartości cech ofiar i drapieżników w każdym pokoleniu. Obliczane jest również liczba kafelków z danym rodzajem terenu w każdym pokoleniu oraz średnia ilość jedzenia na planszy. Dodatkowo co 100 pokoleń wypisywana jest informacja o przetworzonych pokoleniach.

5.4. Wyniki

W niniejszym podrozdziale przedstawię kilka scenariuszy symulacji z różnymi ustawieniami konfiguracyjnymi.



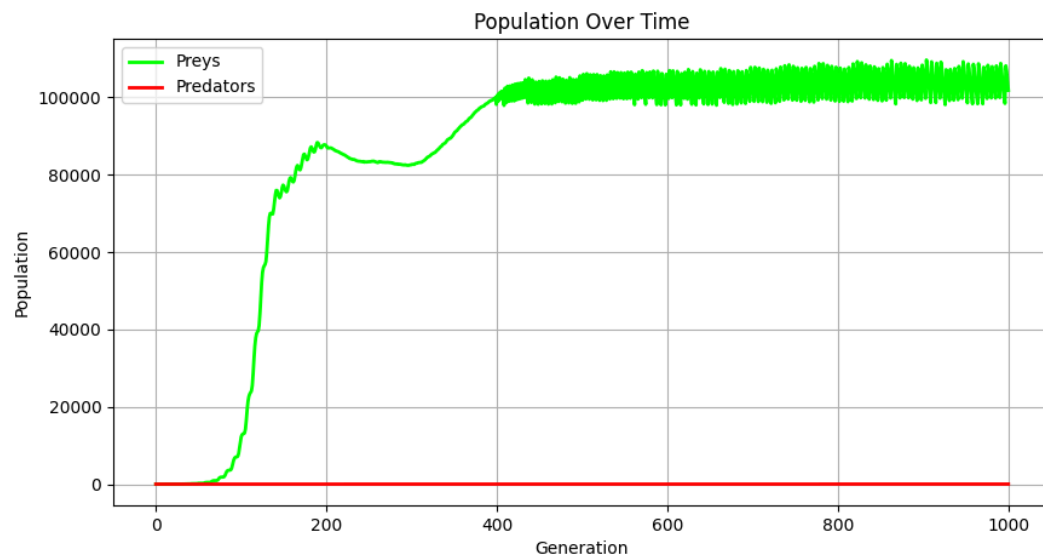
Rysunek 1: Plansza świata z rodzajami terenów

5.4.1. Scenariusz 1 - Wpływ poziomu mutacji na adaptację populacji

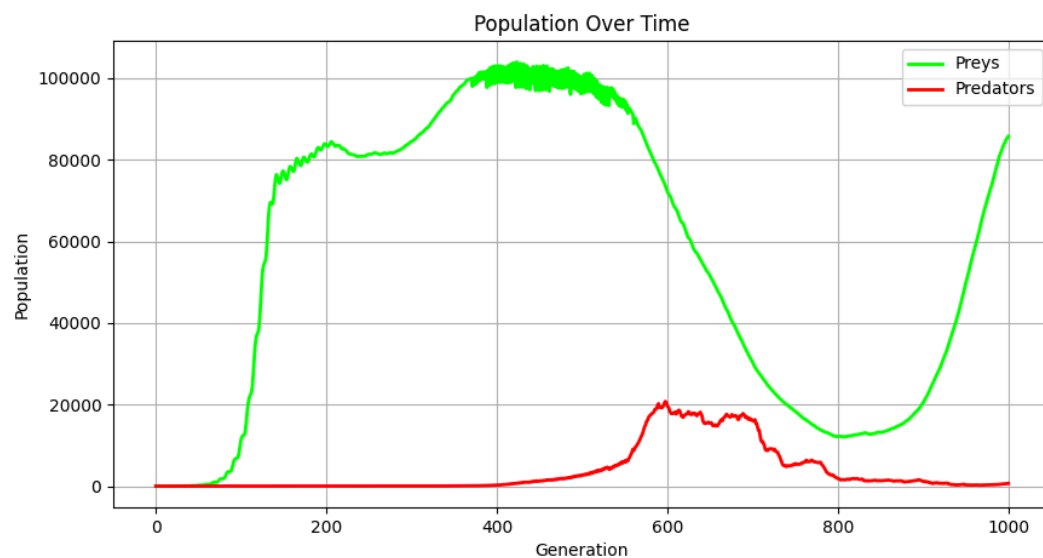
W tym scenariuszu badam jak zmienia się zdolność przetrwania ofiar i drapieżników w zależności od poziomu mutacji.

- Konfiguracja pierwszego wariantu scenariusza (bardzo niski poziom mutacji):
 - Początkowa liczba ofiar: 5
 - Początkowa liczba drapieżników: 5
 - Poziom mutacji ofiar: 0.01
 - Poziom mutacji drapieżników: 0.01
- Konfiguracja drugiego wariantu scenariusza (niski poziom mutacji):
 - Początkowa liczba ofiar: 5
 - Początkowa liczba drapieżników: 5
 - Poziom mutacji ofiar: 0.05

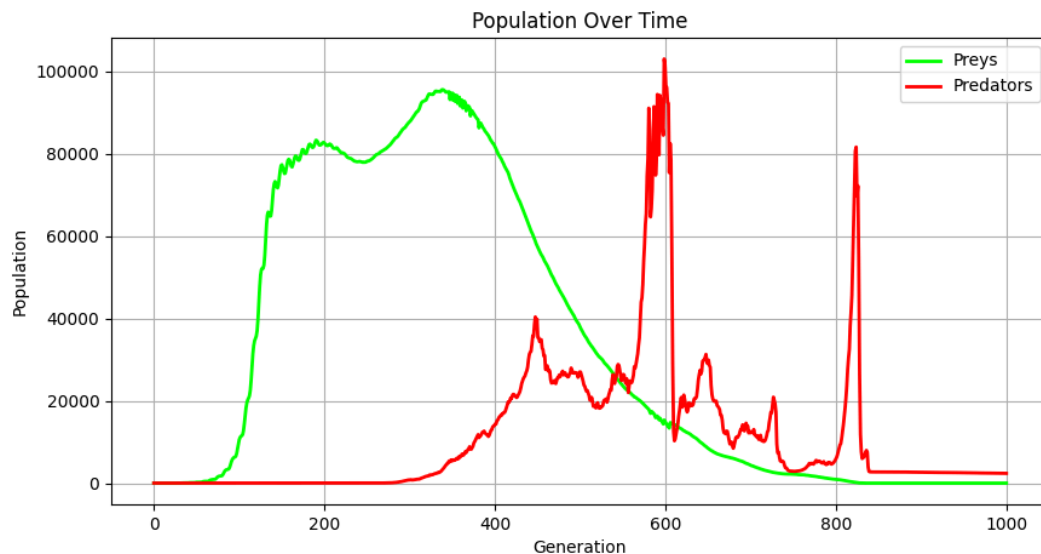
- Poziom mutacji drapieżników: 0.05
- Konfiguracja trzeciego wariantu scenariusza (średni poziom mutacji):
 - Początkowa liczba ofiar: 5
 - Początkowa liczba drapieżników: 5
 - Poziom mutacji ofiar: 0.1
 - Poziom mutacji drapieżników: 0.1
- Konfiguracja czwartego wariantu scenariusza (wysoki poziom mutacji):
 - Początkowa liczba ofiar: 5
 - Początkowa liczba drapieżników: 5
 - Poziom mutacji ofiar: 0.2
 - Poziom mutacji drapieżników: 0.2



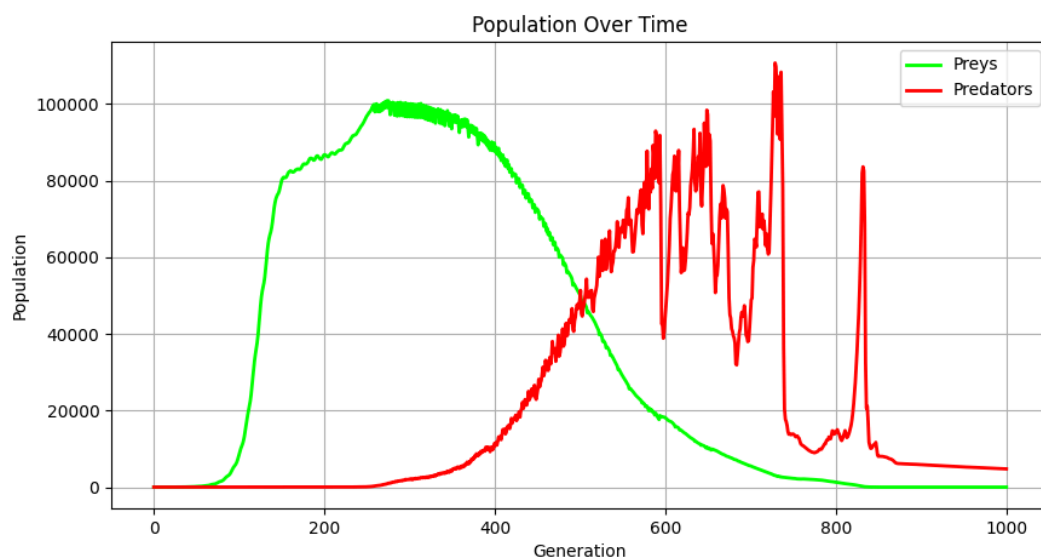
Rysunek 2: Wykres populacji ofiar i drapieżników w czasie (bardzo niski poziom mutacji)



Rysunek 3: Wykres populacji ofiar i drapieżników w czasie (niski poziom mutacji)

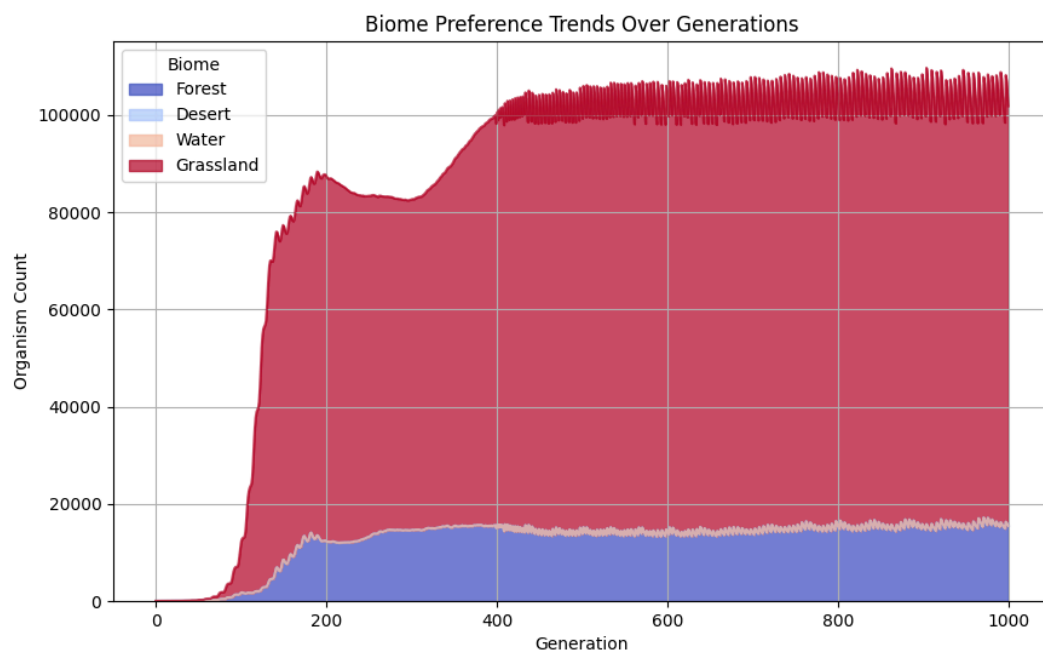


Rysunek 4: Wykres populacji ofiar i drapieżników w czasie (średni poziom mutacji)

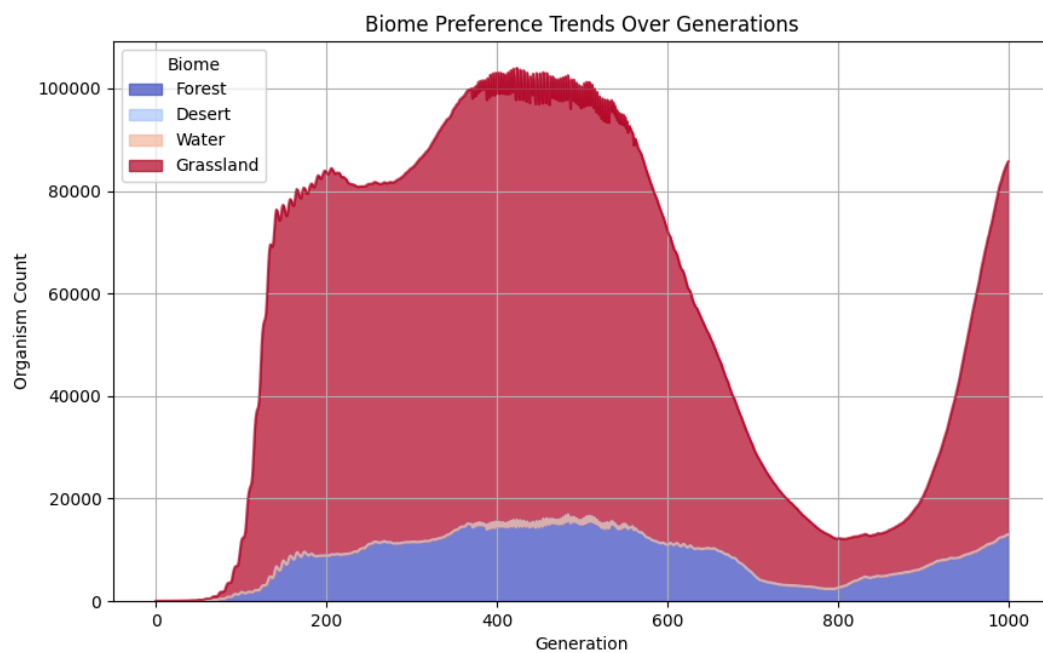


Rysunek 5: Wykres populacji ofiar i drapieżników w czasie (wysoki poziom mutacji)

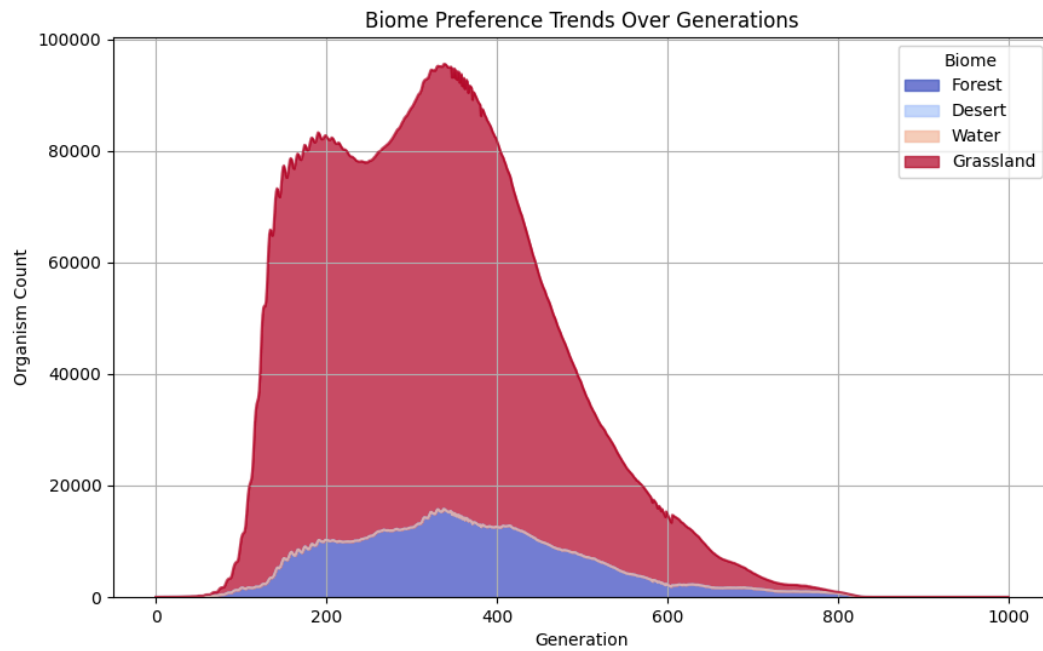
Rys. 2-5 pokazują przebieg zmian liczebności populacji ofiar i drapieżników dla czterech poziomów mutacji (bardzo niski, niski, średni i wysoki). Dla bardzo niskiego poziomu mutacji (rys. 2) widoczny jest powolny wzrost populacji ofiar, jednak drapieżniki nie nadążają z reprodukcją i ostatecznie wygasają. Przy niskim poziomie mutacji (rys. 3) populacja ofiar osiąga stabilny stan, ale wciąż brakuje wystarczającej różnorodności cech, by drapieżniki utrzymały się przez dłuższy czas. Średni poziom mutacji (rys. 4) daje najbardziej zbalansowany przebieg: ofiary i drapieżniki osiągają współegzystencję o niewielkich fluktuacjach, co sugeruje optymalny kompromis między ekspolracją nowych cech, a utrzymaniem adaptacyjności. Natomiast wysoki poziom mutacji (rys. 5) prowadzi do silnych wahań liczebności obu populacji, co jest efektem niestabilności genetycznej i zbyt gwałtownych zmian w progu reprodukcji.



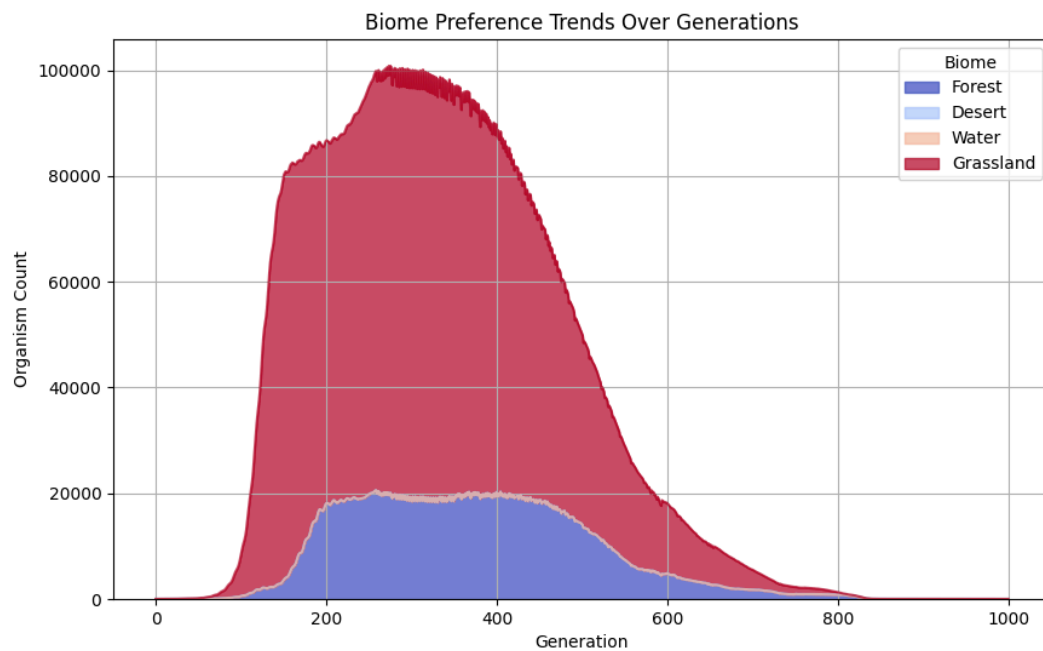
Rysunek 6: Trendy preferencji terenowych ofiar w czasie (bardzo niski poziom mutacji)



Rysunek 7: Trendy preferencji terenowych ofiar w czasie (niski poziom mutacji)

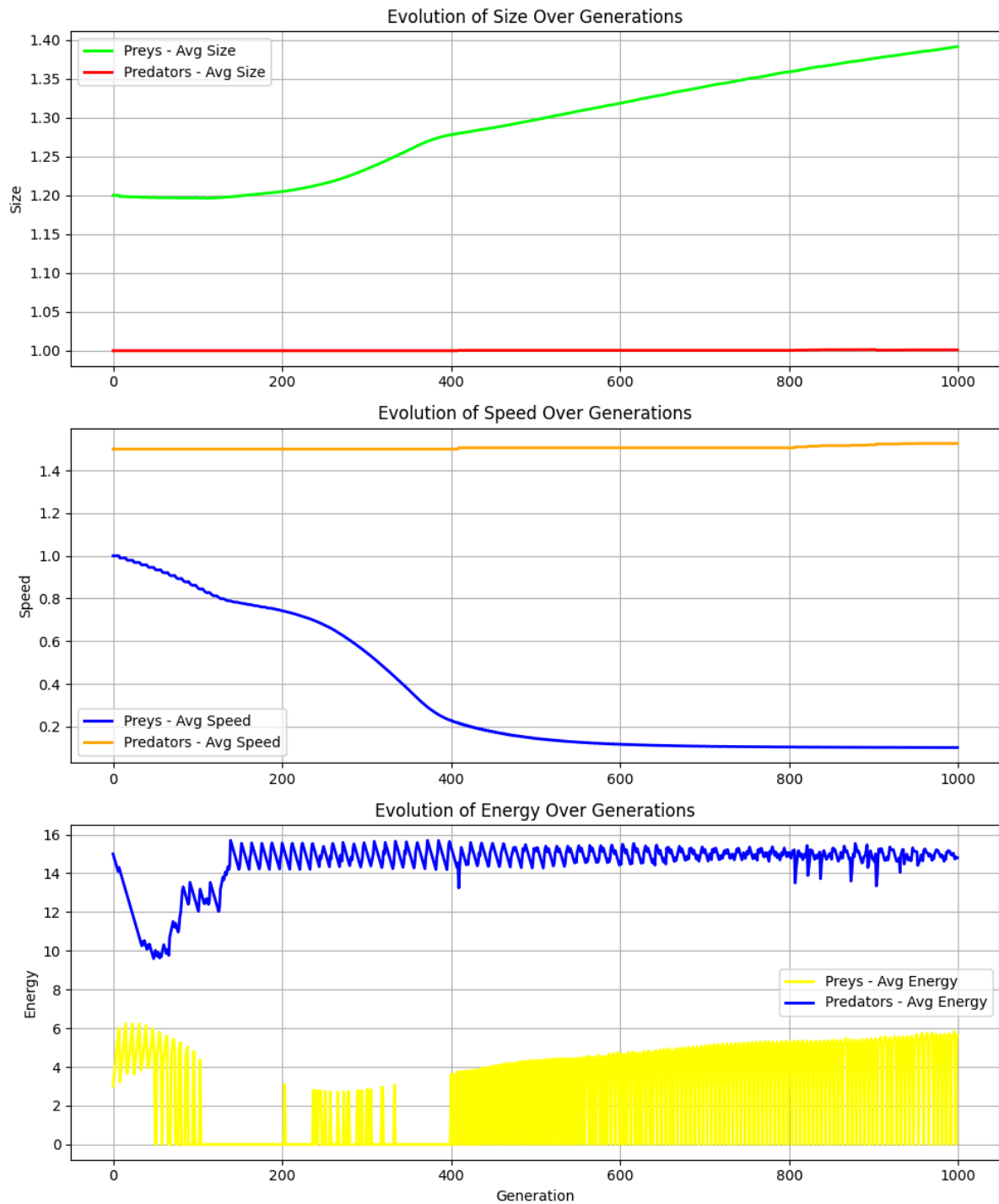


Rysunek 8: Trendy preferencji terenowych ofiar w czasie (średni poziom mutacji)

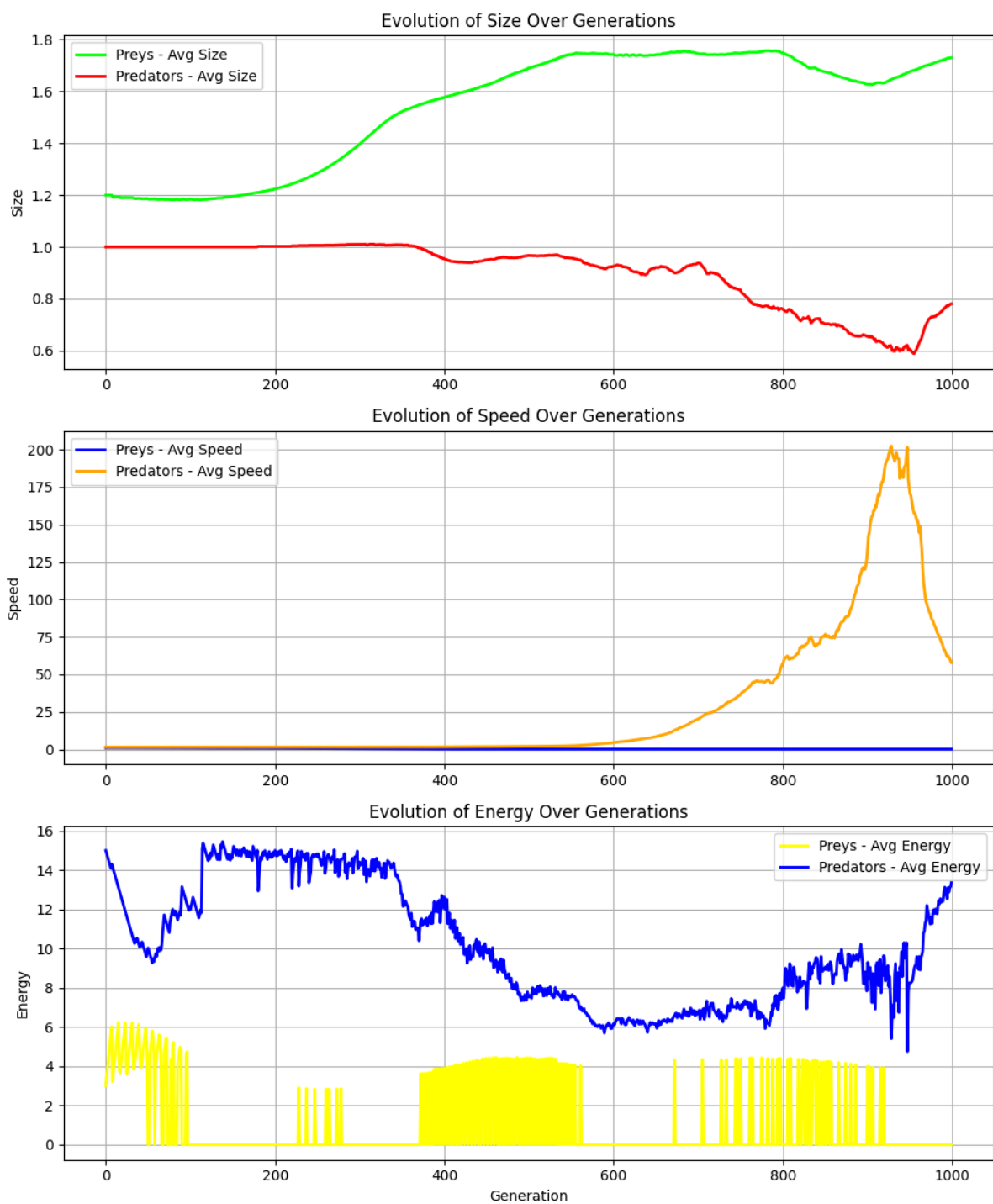


Rysunek 9: Trendy preferencji terenowych ofiar w czasie (wysoki poziom mutacji)

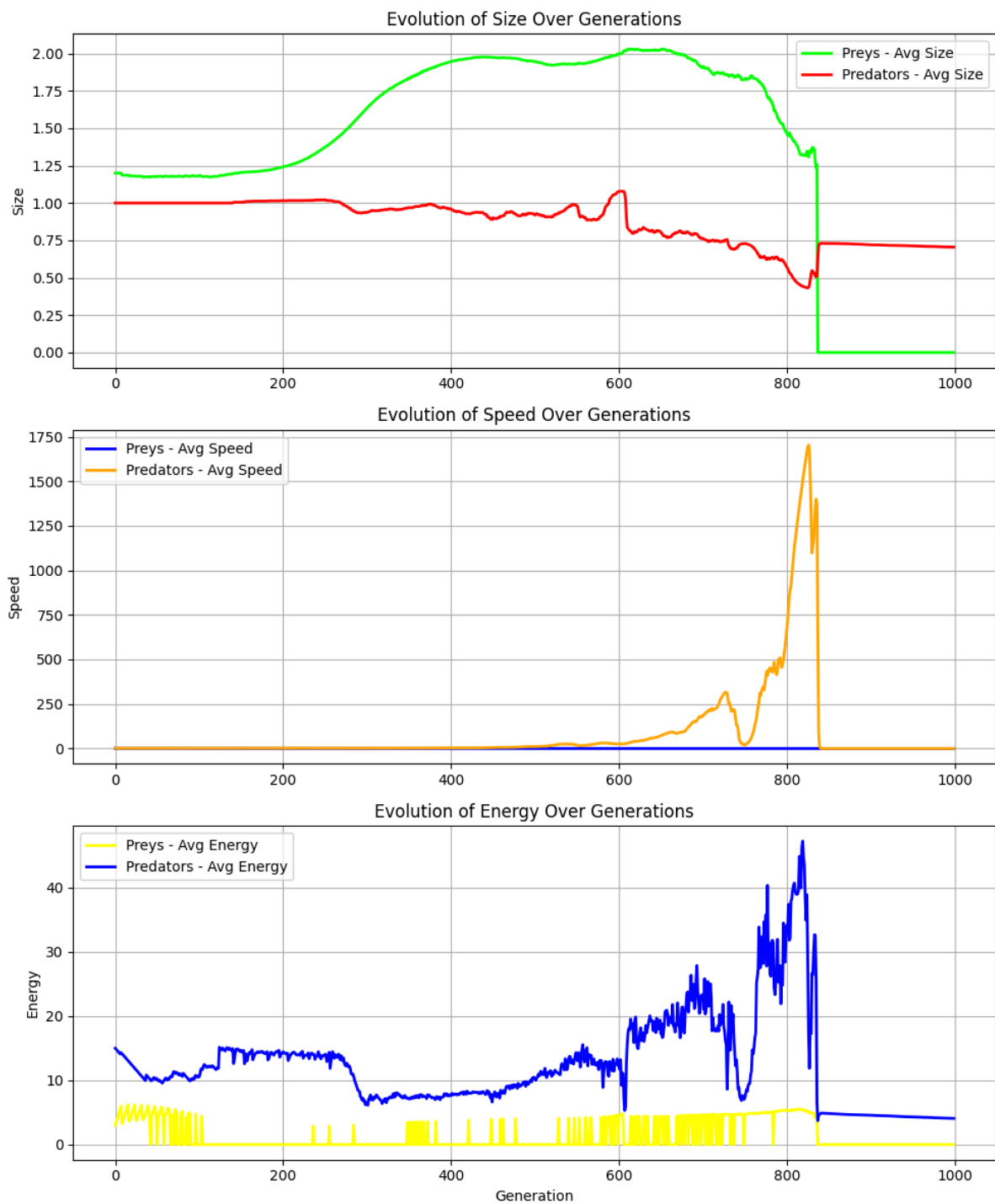
Rys. 6-9 ilustrują, jak zmieniają się preferencje terenowe ofiar w czasie dla kolejnych poziomów mutacji. W warunkach bardzo niskiej mutacji (rys. 6) ofiary silnie preferują biotopy o najwyższej początkowej tolerancji, co prowadzi do szybkiego przeludnienia tych stref. Zwiększenie mutowalności powoduje stopniowe rozproszenie preferencji - przy średnim poziomie mutacji (rys. 8) zaobserwowano bardziej równomierny rozkład tolerancji, co przekłada się na lepsze wykorzystanie zasobów. Przy wysokim poziomie mutacji (rys. 9) adaptacja do każdego typu terenu jest na tyle dynamiczna, że nie tworzą się długotrwałe ogniska przeludnienia, lecz populacja eksperymentuje ze wszystkimi biotopami.



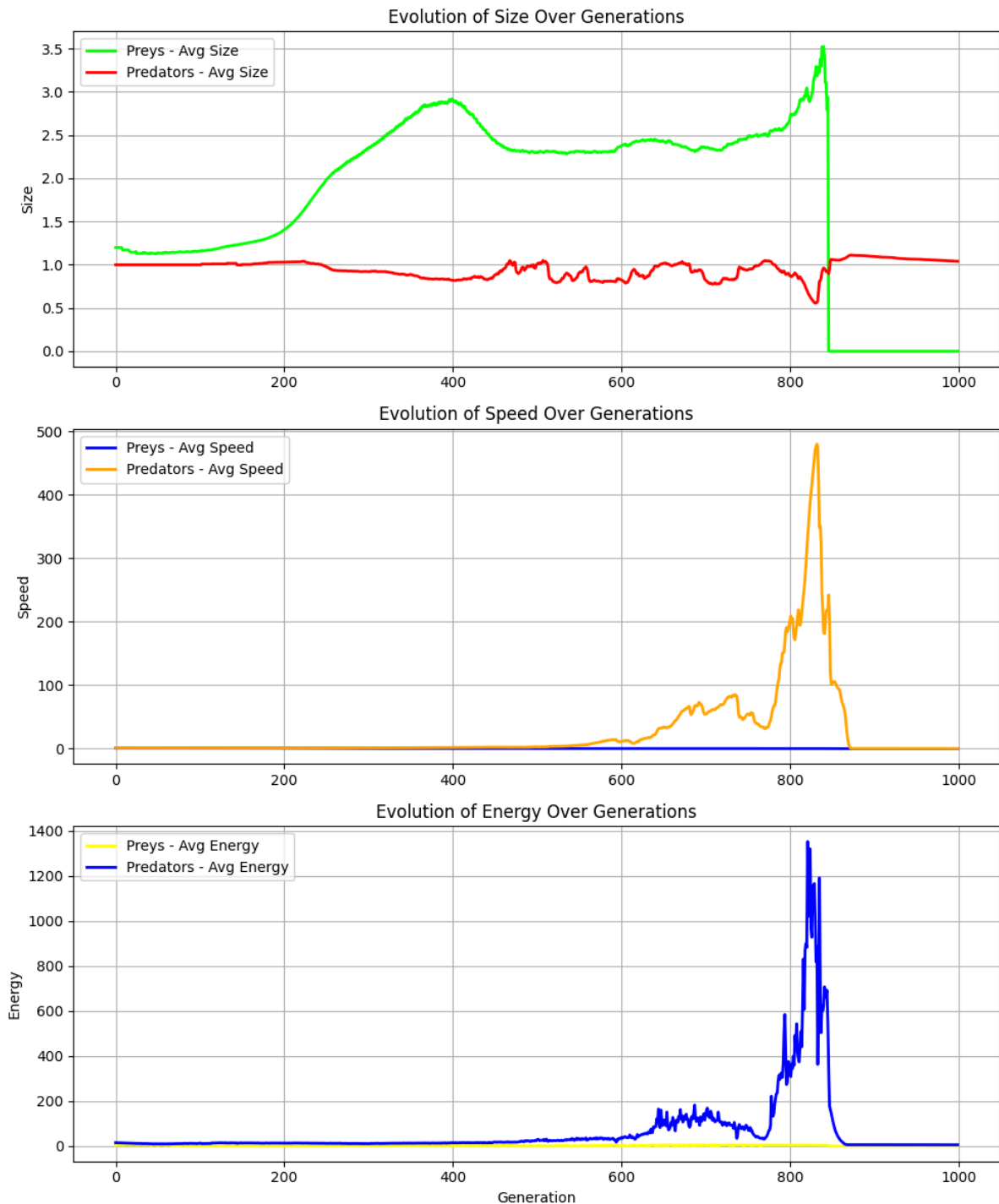
Rysunek 10: Ewolucja cech ofiar i drapieżników w czasie (bardzo niski poziom mutacji)



Rysunek 11: Ewolucja cech ofiar i drapieżników w czasie (niski poziom mutacji)

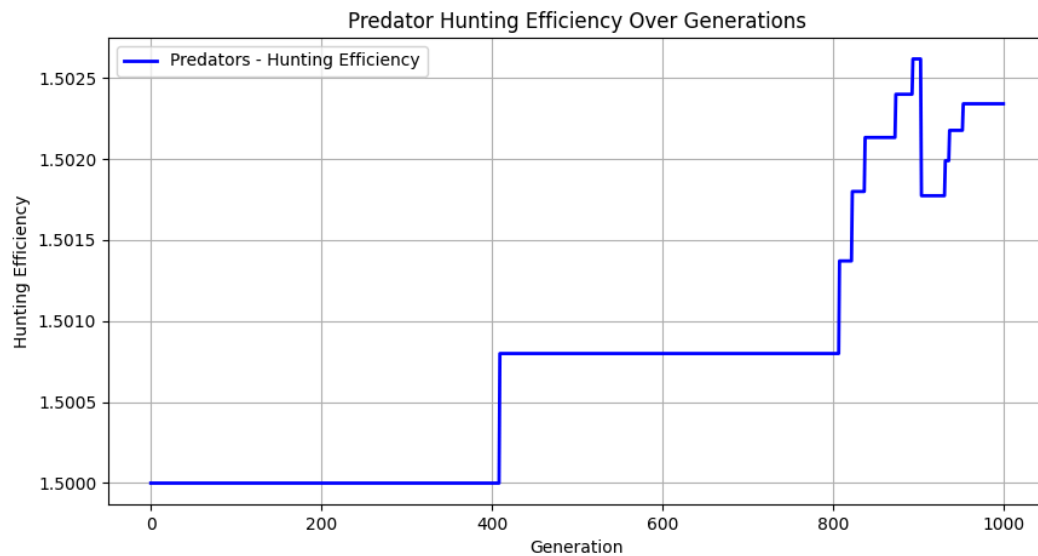


Rysunek 12: Ewolucja cech ofiar i drapieżników w czasie (średni poziom mutacji)

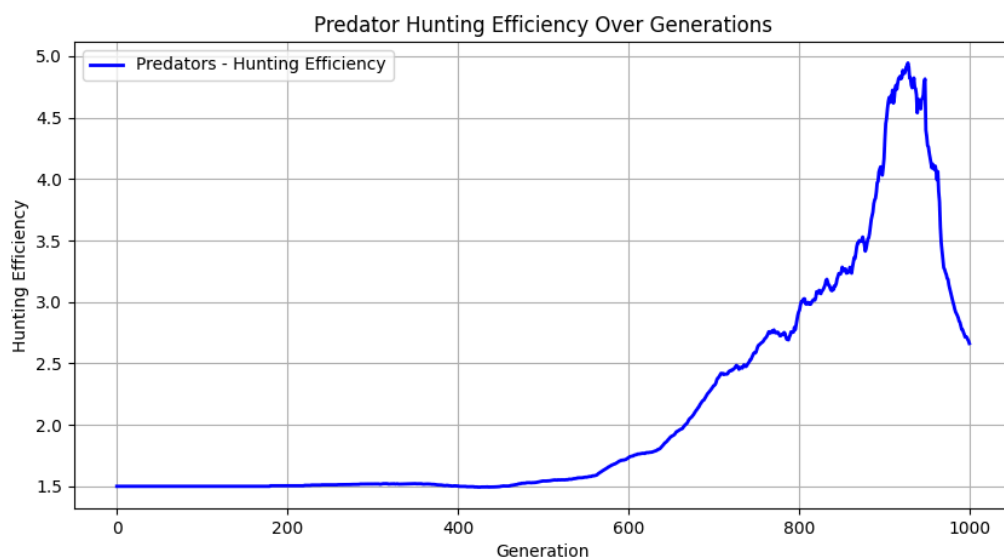


Rysunek 13: Ewolucja cech ofiar i drapieżników w czasie (wysoki poziom mutacji)

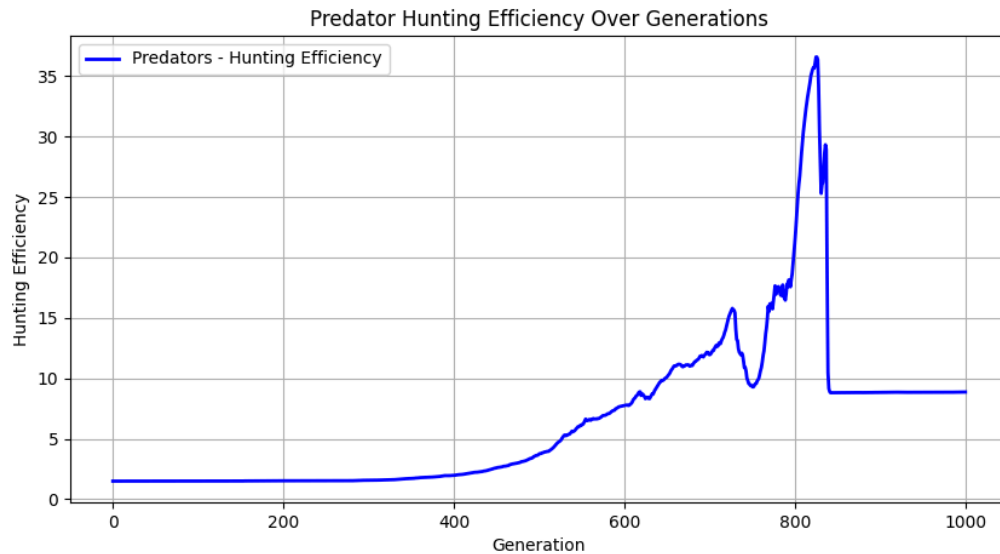
Rys. 10-13 pokazują, jak w kolejnych pokoleniach zmieniają się średnie wartości kluczowych cech (rozmiar, prędkość, energia, próg rozmnażania) ofiar i drapieżników przy czterech poziomach mutacji. W wariancie bardzo niskiej mutacji (rys. 10) zaobserwowano niemal płaskie krzywe, co świadczy o minimalnej zmianie cech w czasie. Przy niskiej mutacji (rys. 11) widoczny jest powolny, ale systematyczny wzrost średniej energii i prędkości ofiar, sugerujący łagodną presję selekcyjną. Średni poziom mutacji (rys. 12) prowadzi do szybszej optymalizacji cech, gdzie drapieżniki stają się efektywniejsze, a ofiary nieznacznie zwiększają rozmiar i prędkość, co wskazuje na dobrze zbalansowaną adaptację. Natomiast przy wysokim poziomie mutacji (rys. 13) krzywe gwałtownie oscylują, sygnalizując niestabilność genetyczną i brak trwałego trendu adaptacyjnego.



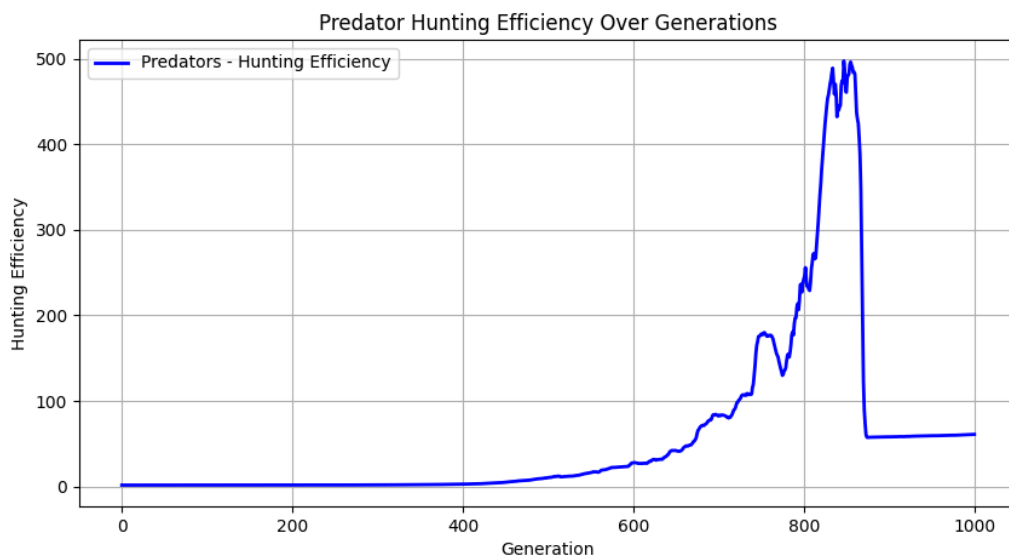
Rysunek 14: Wydajność polowania drapieżników w czasie (bardzo niski poziom mutacji)



Rysunek 15: Wydajność polowania drapieżników w czasie (niski poziom mutacji)

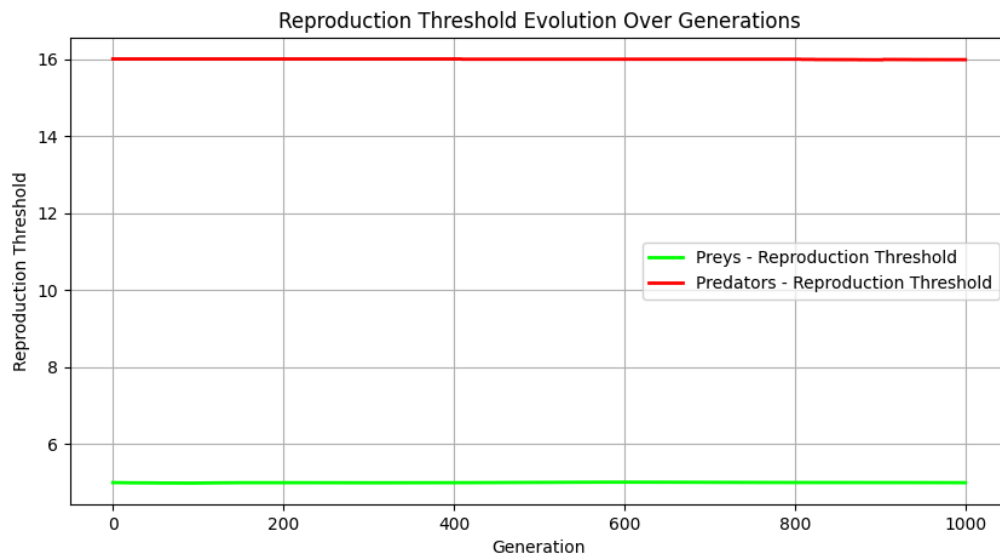


Rysunek 16: Wydajność polowania drapieżników w czasie (średni poziom mutacji)

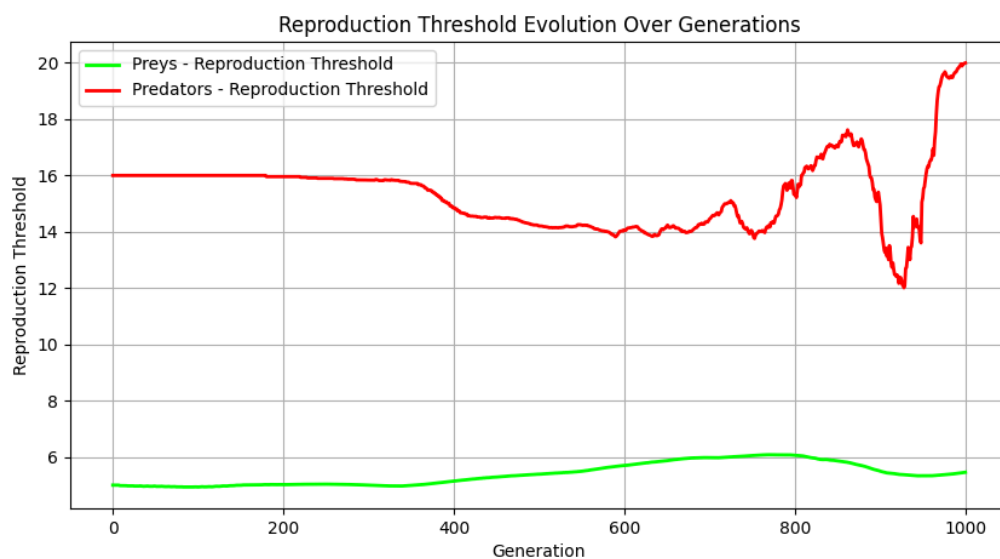


Rysunek 17: Wydajność polowania drapieżników w czasie (wysoki poziom mutacji)

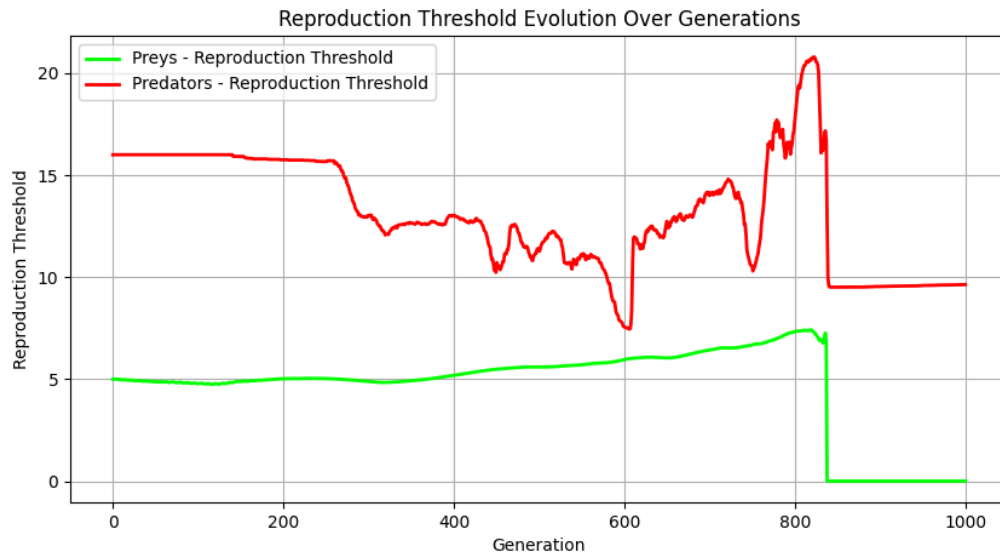
Rys. 14-17 przedstawiają zmiany średniej wydajności polowania drapieżników dla tych samych czterech poziomów mutacji. Przy bardzo niskiej mutacji (rys. 14) efektywność utrzymuje się blisko wartości początkowej, co oznacza słabą adaptację do strategii łowieckiej. Niska mutacja (rys. 15) skutkuje stopniowym wzrostem efektywności. Średni poziom mutacji (rys. 16) sprzyja szybkiemu osiągnięciu maksymalnej efektywności i jej stabilizacji, natomiast przy wysokim poziomie mutacji (rys. 17) wydajność podlega dużym wahaniom, co wskazuje na ciągłe testowanie eksteremalnych strategii łowieckich.



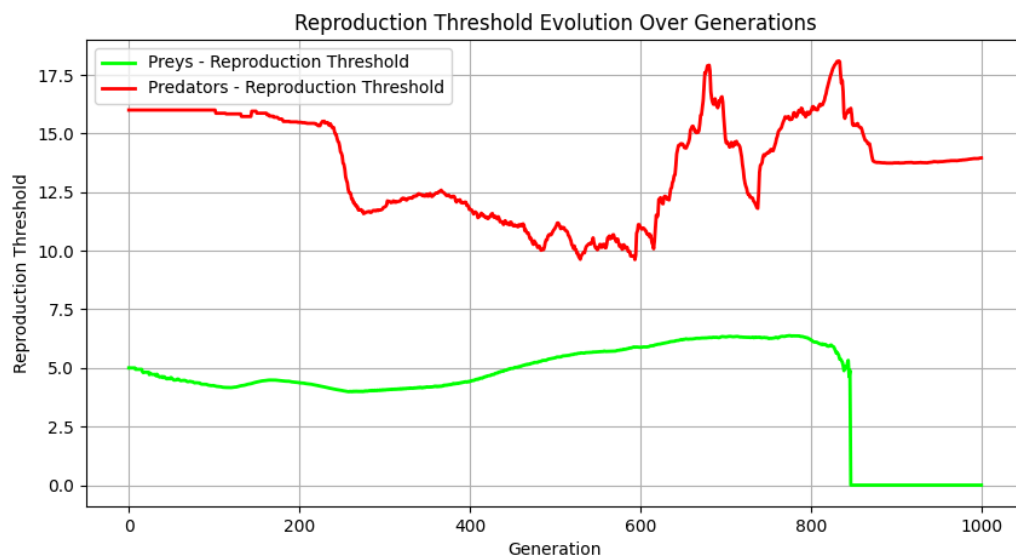
Rysunek 18: Trendy progu reprodukcji ofiar i drapieżników w czasie (bardzo niski poziom mutacji)



Rysunek 19: Trendy progu reprodukcji ofiar i drapieżników w czasie (niski poziom mutacji)



Rysunek 20: Trendy progu reprodukcji ofiar i drapieżników w czasie (średni poziom mutacji)



Rysunek 21: Trendy progu reprodukcji ofiar i drapieżników w czasie (wysoki poziom mutacji)

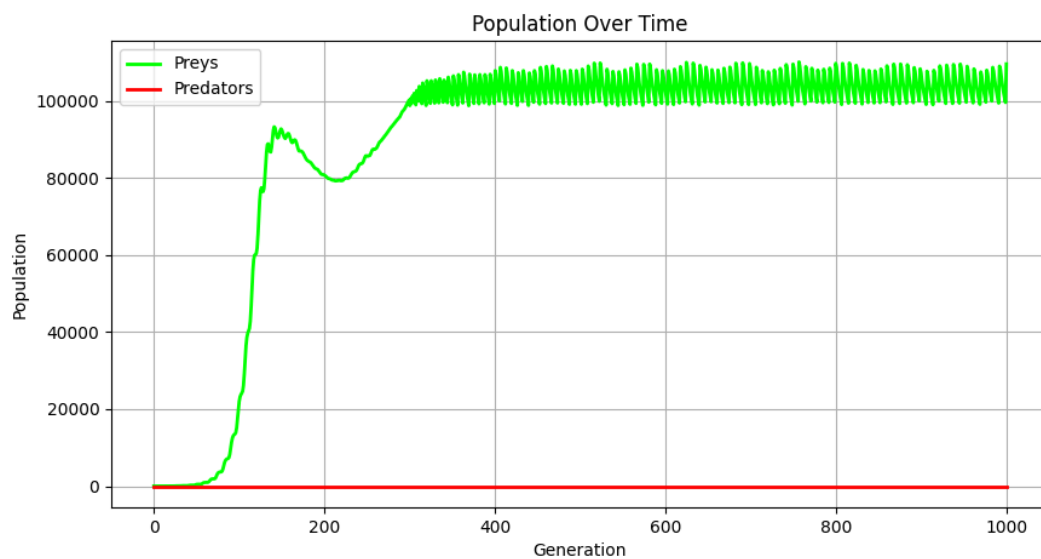
Rys. 18-21 ilustrują ewolucję średniego progu rozmnażania w czterech wariantach mutacji. W przypadku bardzo niskiej mutacji (rys. 18) próg niemal nie zmienia się, co świadczy o braku selekcji w kierunku optymalnego kosztu rozmnażania. Niska mutacja (rys. 19) prowadzi do łagodnego obniżenia progu, co przyspiesza reprodukcję, ale bez destabilizacji systemu. Średni poziom mutacji (rys. 20) skutkuje wyraźnym spadkiem progu do wartości równoważącej tempo reprodukcji i przeżywalność, natomiast przy wysokiej mutacji (rys. 21) próg znacząco oscyluje, odzwierciedlając nieregularne zmiany genetyczne.

5.4.2. Scenariusz 2 - Wpływ drapieżników na populację ofiar

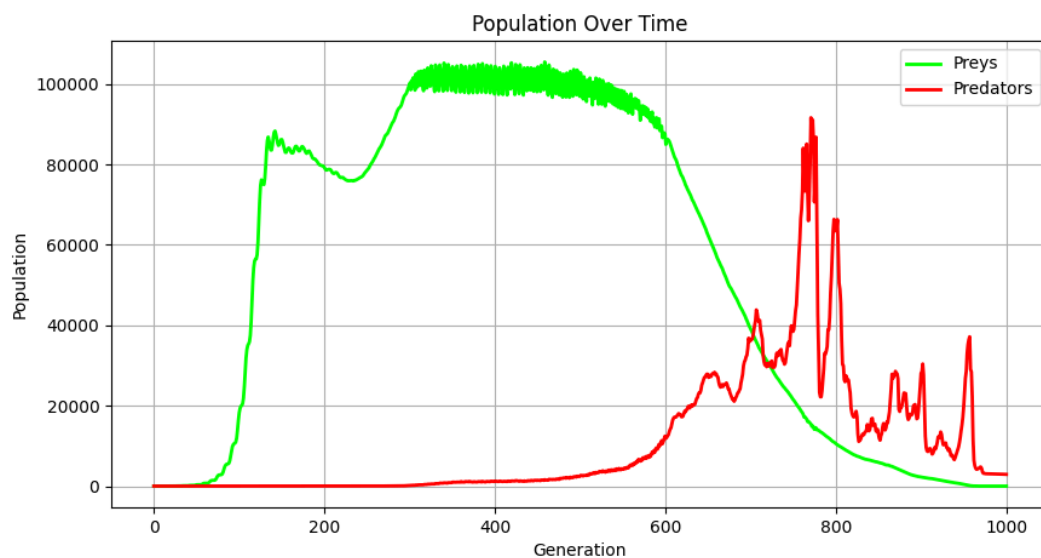
W scenariuszu 2 badam, czy ofiary bez drapieżników szybciej się rozmnażają lub cierpią z powodu przełudnienia oraz jak silna presja drapieżników wpływa na populację ofiar.

- Konfiguracja pierwszego wariantu scenariusza (brak drapieżników):
 - Początkowa liczba ofiar: 10
 - Początkowa liczba drapieżników: 0

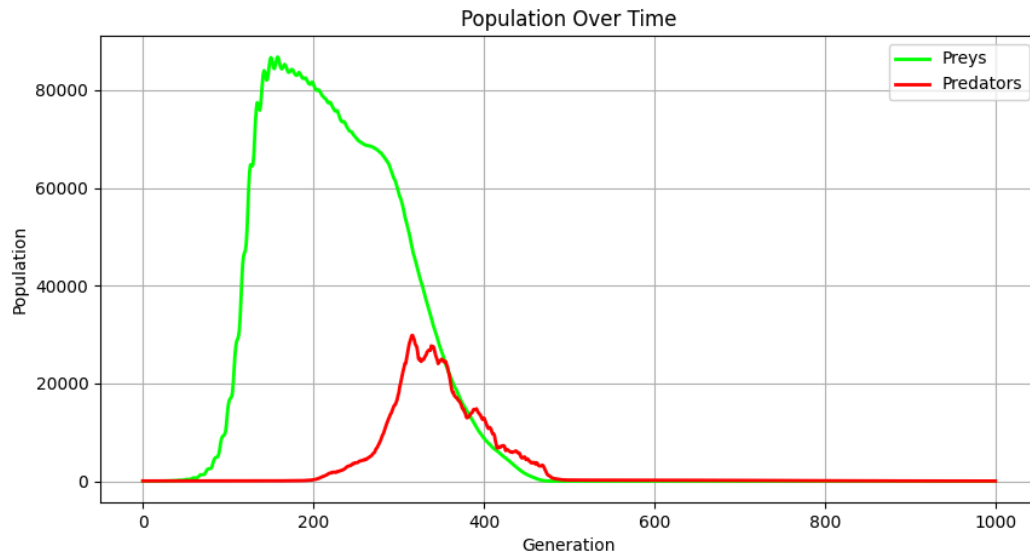
- Poziom mutacji ofiar: 0.1
- Konfiguracja drugiego wariantu scenariusza (normalna presja drapieżników):
 - Początkowa liczba ofiar: 10
 - Początkowa liczba drapieżników: 5
 - Poziom mutacji ofiar: 0.1
 - Poziom mutacji drapieżników: 0.1
- Konfiguracja trzeciego wariantu scenariusza (wysoka presja drapieżników):
 - Początkowa liczba ofiar: 10
 - Początkowa liczba drapieżników: 40
 - Poziom mutacji ofiar: 0.1
 - Poziom mutacji drapieżników: 0.1



Rysunek 22: Wykres populacji ofiar i drapieżników w czasie (brak drapieżników)

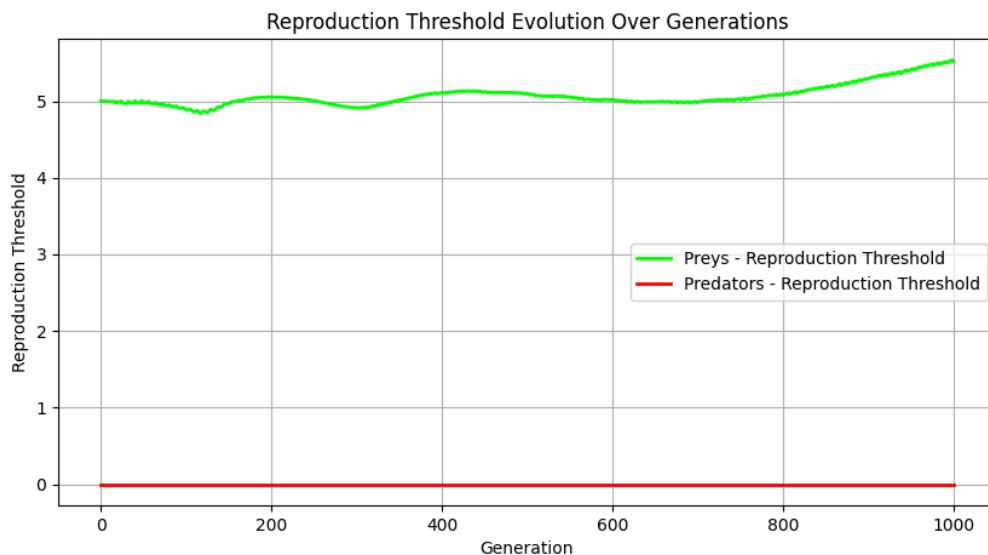


Rysunek 23: Wykres populacji ofiar i drapieżników w czasie (normalna presja drapieżników)

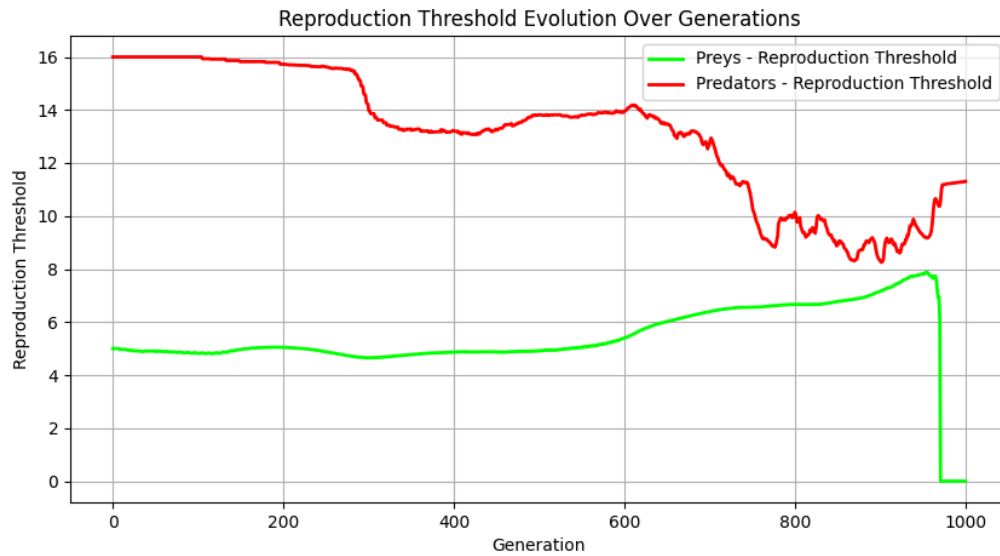


Rysunek 24: Wykres populacji ofiar i drapieżników w czasie (wysoka presja drapieżników)

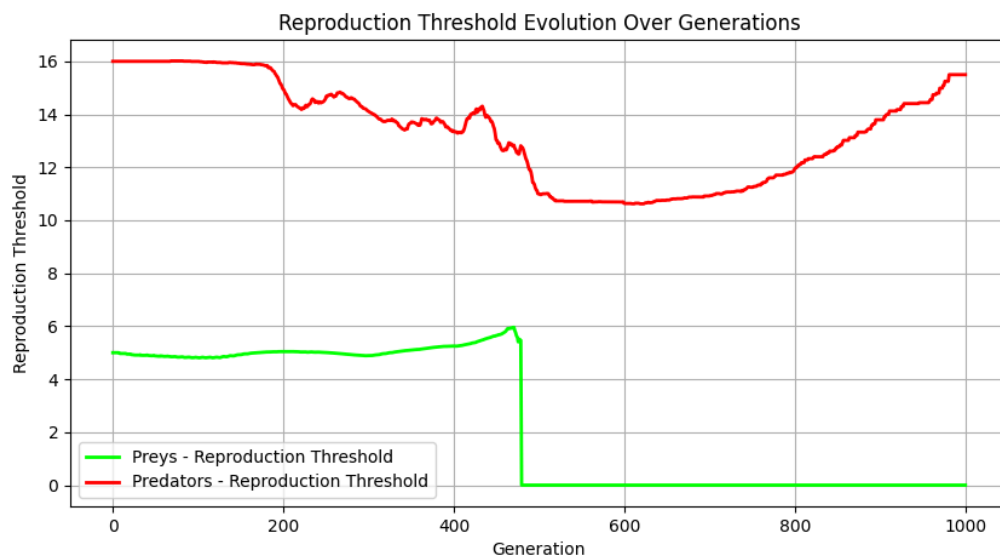
Rys. 22-24 pokazują dynamikę populacji w trzech wariantach presji drapieżników: brak, normalna i wysoka. Bez drapieżników (rys. 22) ofiary rosną niemal wykładniczo, co prowadzi do przeludnienia i wzrostu presji na zasoby. W wariantcie z umiarkowaną presją (rys. 23) widoczna jest stabilizacja liczebności, drapieżniki utrzymują populację ofiar w równowadze, co sprzyja również utrzymaniu zróżnicowania cech. Natomiast wysoka presja drapieżników (rys. 24) prowadzi do wyginięcia ofiar, a w konsekwencji do katastrofalnego spadku drapieżników, co obrazuje kruchą współzależność obu populacji.



Rysunek 25: Trendy proggu reprodukcji ofiar i drapieżników w czasie (brak drapieżników)

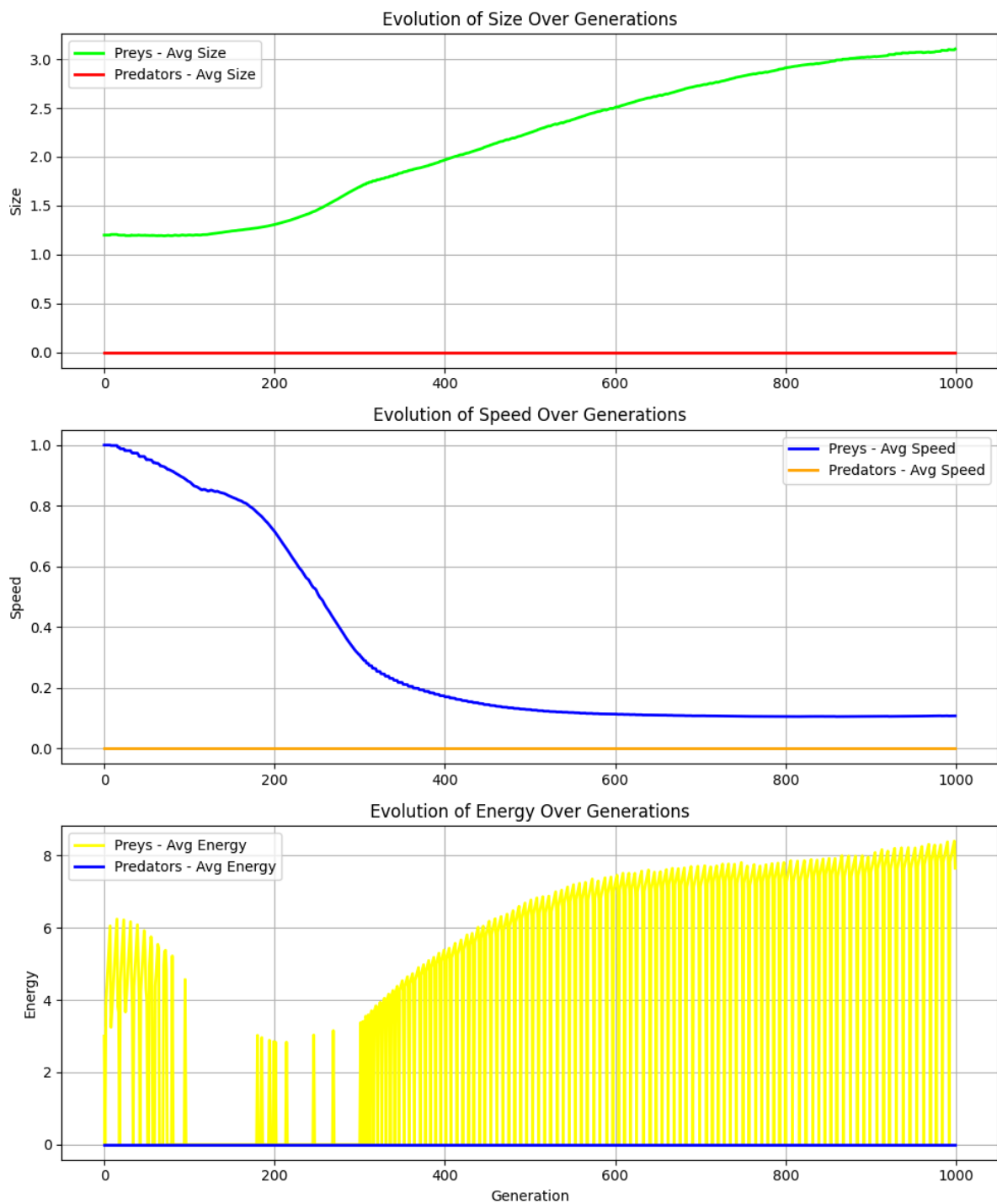


Rysunek 26: Trendy progu reprodukcji ofiar i drapieżników w czasie (normalna presja drapieżników)

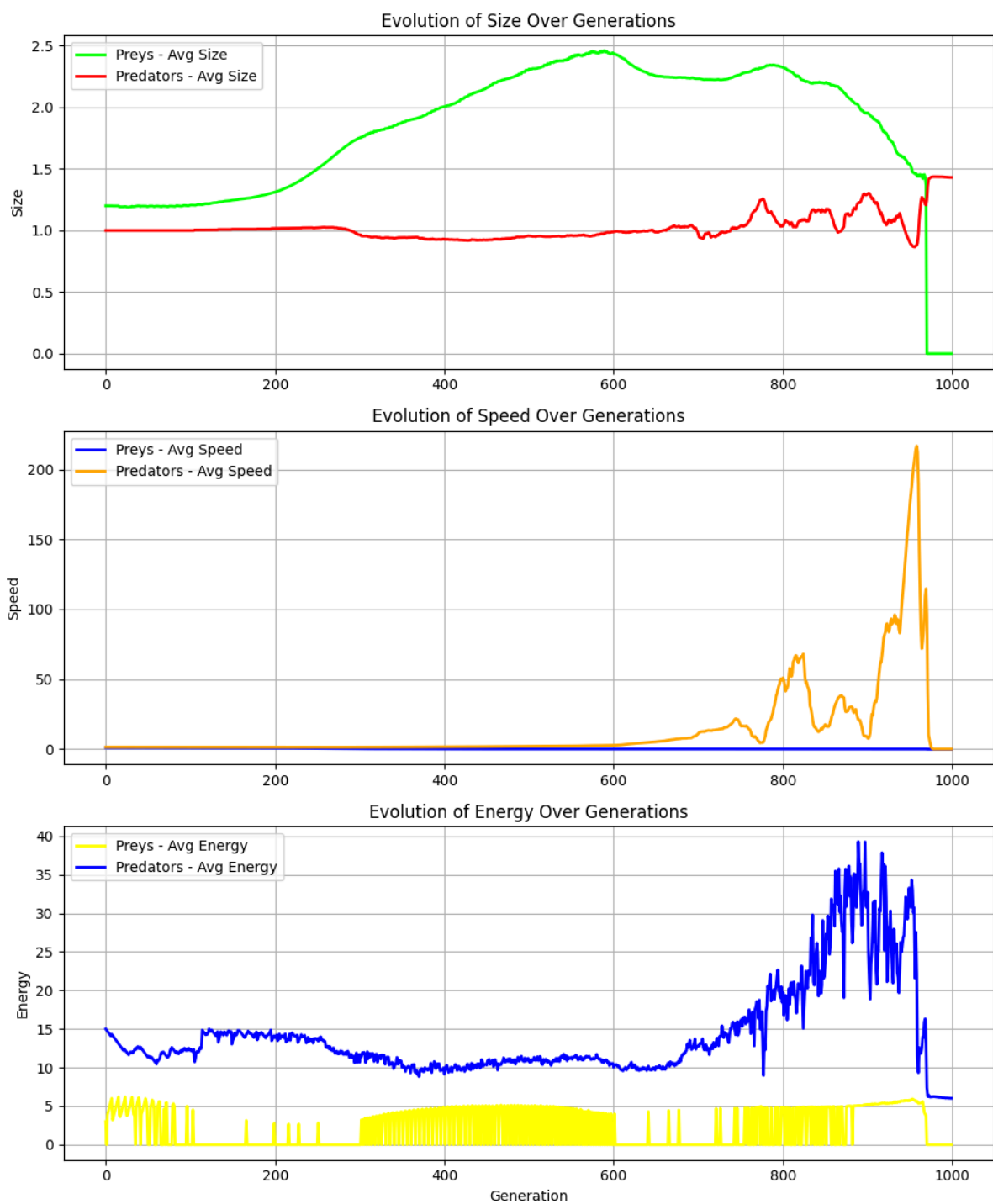


Rysunek 27: Trendy progu reprodukcji ofiar i drapieżników w czasie (wysoka presja drapieżników)

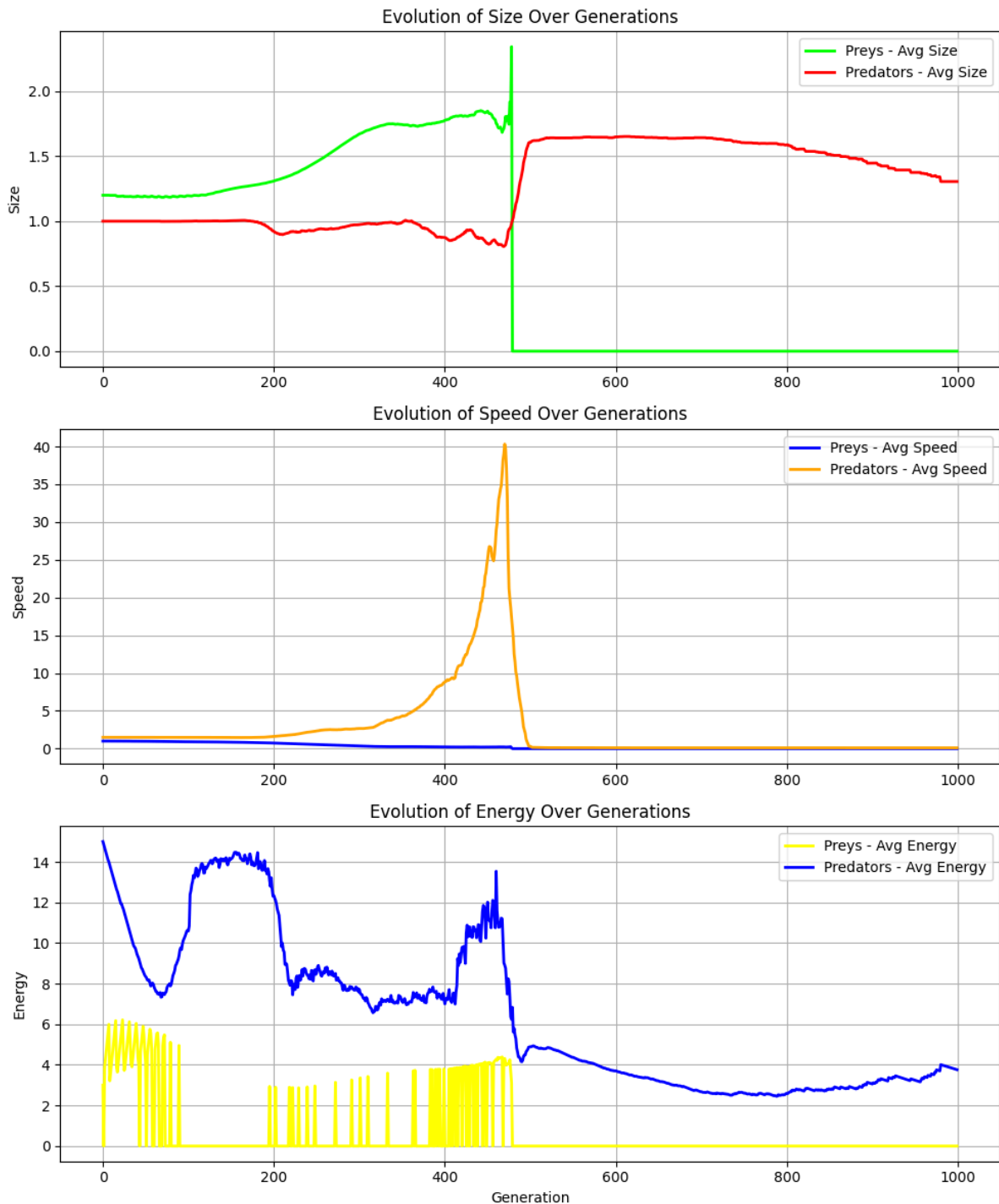
Rys. 25-27 ukazują zachowanie średniego progu reprodukcji ofiar i drapieżników w scenariuszu 2: bez drapieżników (rys. 25), przy normalnej presji (rys. 26) i wysokiej presji drapieżników (rys. 27). W wariancie braku drapieżników próg reprodukcji ofiar nieznacznie rośnie. W normalnej presji drapieżników (rys. 26) próg reprodukcji ofiar wzrasta, a drapieżników ciągle spada, co sugeruje, że ofiary gromadzą dużo energii i to powoduje, że ich próg reprodukcji rośnie, natomiast u drapieżników próg reprodukcji maleje przez to, że sporą część energii zużywają na polowanie. W wariancie wysokiej presji drapieżników (rys. 27) trend ofiar jest taki sam jak w przypadku normalnej presji, natomiast u drapieżników do momentu wyginięcia ofiar trend jest malejący, a następnie rośnie.



Rysunek 28: Ewolucja cech ofiar i drapieżników w czasie (brak drapieżników)

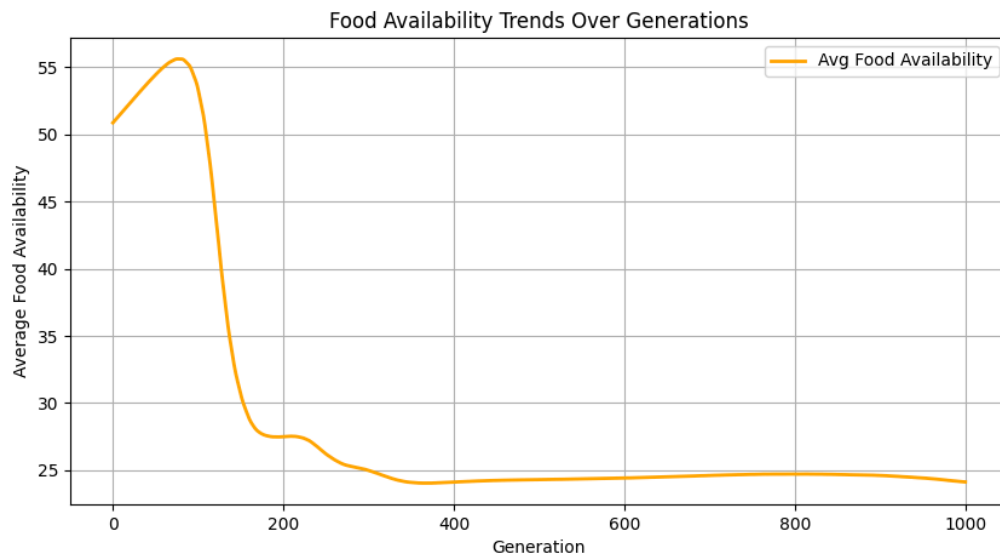


Rysunek 29: Ewolucja cech ofiar i drapieżników w czasie (normalna presja drapieżników)

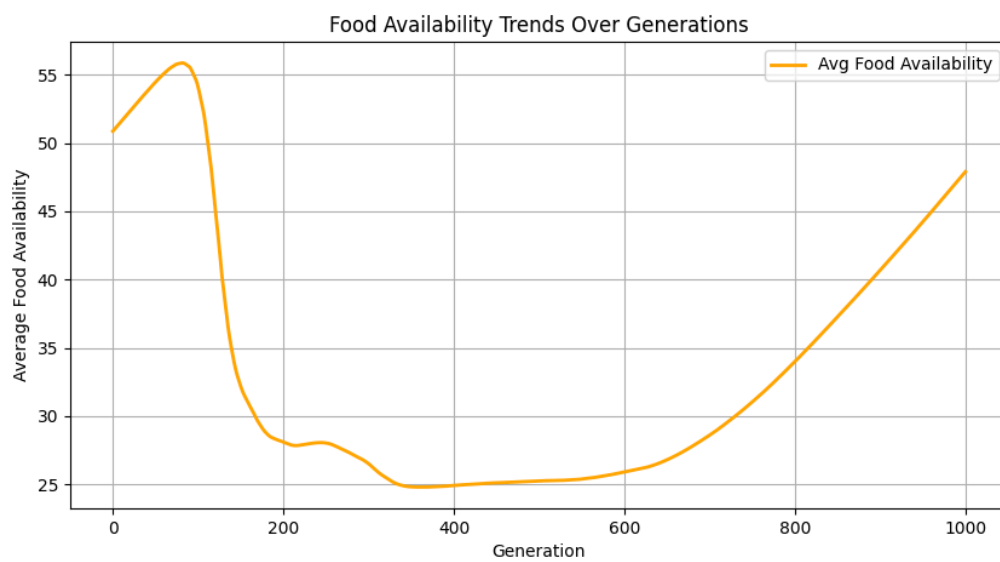


Rysunek 30: Ewolucja cech ofiar i drapieżników w czasie (wysoka presja drapieżników)

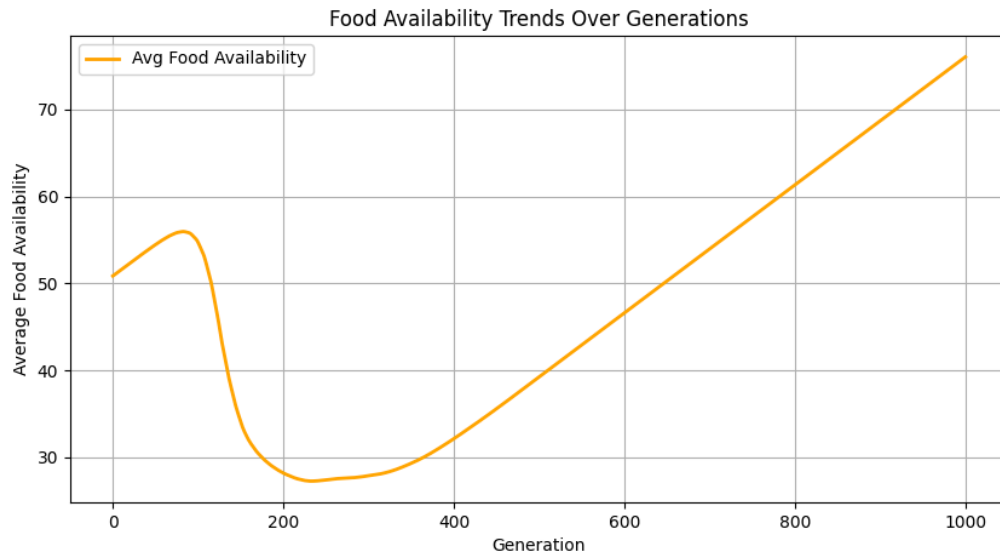
Rys. 28-30 przedstawiają zmiany kluczowych cech (rozmiar, prędkość, energia) ofiar i drapieżników w scenariuszu 2: brak drapieżników (rys. 28), normalna presja drapieżników (rys. 29) i wysoka presja drapieżników (rys. 30). W przypadku braku drapieżników, cechy ewoluują powoli i kierunkowo ku większym rozmiarom i mniejszym prędkościom, bez silnej presji selekcyjnej. W normalnej presji ofiary stają się coraz większe, a ich prędkość jest stabilna, co jest zaskakujące, ponieważ sądziłem, że drapieżniki będą wymuszać większą prędkość i mniejsze rozmiary. Wysoka presja drapieżników prowadzi do szybszego tempa ewolucji cech drapieżników, którzy stają się jeszcze bardziej efektywni, co skutkuje szybkim wyginięciem ofiar i drapieżników.



Rysunek 31: Dostępność jedzenia jedzenia w czasie (brak drapieżników)



Rysunek 32: Dostępność jedzenia jedzenia w czasie (normalna presja drapieżników)



Rysunek 33: Dostępność jedzenia w czasie (wysoka presja drapieżników)

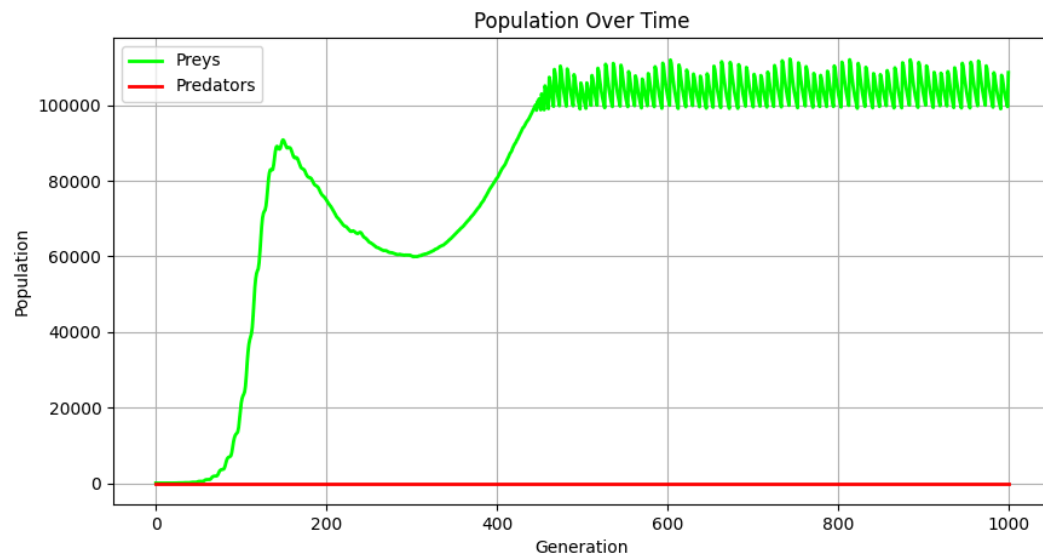
Rys. 31-33 ilustrują zmiany dostępności pożywienia w zależności od obecności drapieżników. W warunkach braku drapieżników (rys. 31) zasoby pokarmowe szybko się wyczerpują w wyniku nadmiernego rozmnażania ofiar. W normalnej presji (rys. 32) tempo zużycia jest wolniejsze, co pozwala na cykliczne odnawianie się pokarmu i utrzymanie stabilnej populacji. W wariancie wysokiej presji drapieżników (rys. 33) dostępność jedzenia spada najszybciej, a kolejne pokolenia doświadczają gwałtownych niedoborów, co prowadzi do wyginięcia ofiar i drapieżników.

5.4.3. Scenariusz 3 - Wpływ dostępności jedzenia na populację ofiar

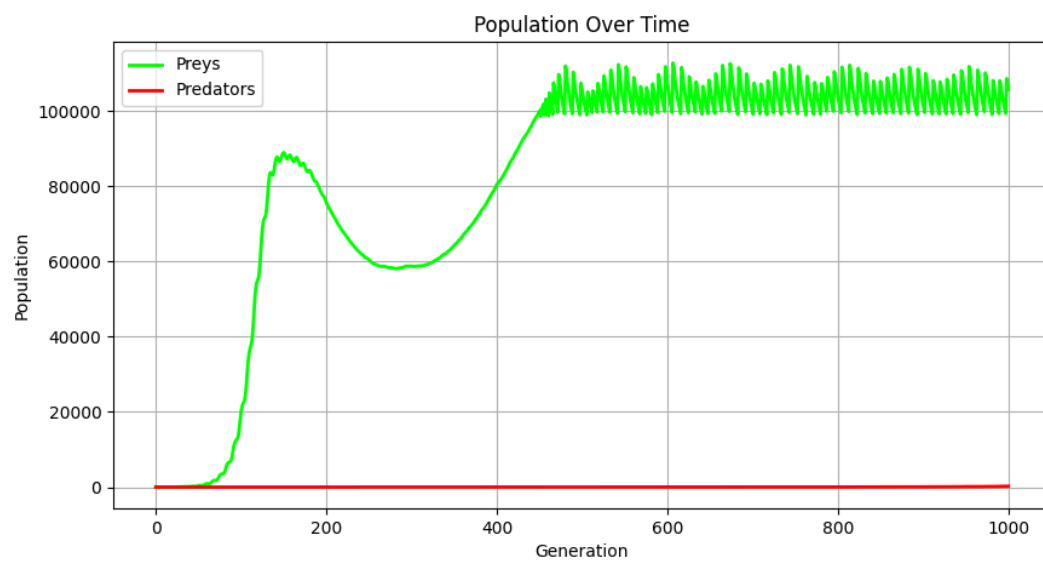
W scenariuszu 3 badam, jak wpływa na populację zmiana zasobności środowiska w jedzenie. Czy organizmy lepiej przystosowują się do terenu bogatego w jedzenie? Czy populacja jest w stanie przetrwać w jałowym środowisku?

- Konfiguracja pierwszego wariantu scenariusza (jałowe środowisko, brak drapieżników):
 - Las - poziom regeneracji jedzenia: 0.05
 - Pustynia - poziom regeneracji jedzenia: 0.001
 - Woda - poziom regeneracji jedzenia: 0 - w tym terenie nie ma jedzenia
 - Teren trawiasty - poziom regeneracji jedzenia: 0.01
- Konfiguracja pierwszego wariantu scenariusza (jałowe środowisko):
 - Jak powyżej, natomiast ilość drapieżników wynosi 1
- Konfiguracja drugiego wariantu scenariusza (normalne środowisko, brak drapieżników):
 - Las - poziom regeneracji jedzenia: 0.2
 - Pustynia - poziom regeneracji jedzenia: 0.01
 - Woda - poziom regeneracji jedzenia: 0
 - Teren trawiasty - poziom regeneracji jedzenia: 0.1
- Konfiguracja drugiego wariantu scenariusza (normalne środowisko):
 - Jak powyżej, natomiast ilość drapieżników wynosi 1
- Konfiguracja trzeciego wariantu scenariusza (bardzo bogate środowisko, brak drapieżników):
 - Las - poziom regeneracji jedzenia: 0.4
 - Pustynia - poziom regeneracji jedzenia: 0.01
 - Woda - poziom regeneracji jedzenia: 0
 - Teren trawiasty - poziom regeneracji jedzenia: 0.3
- Konfiguracja trzeciego wariantu scenariusza (bardzo bogate środowisko):
 - Jak powyżej, natomiast ilość drapieżników wynosi 1

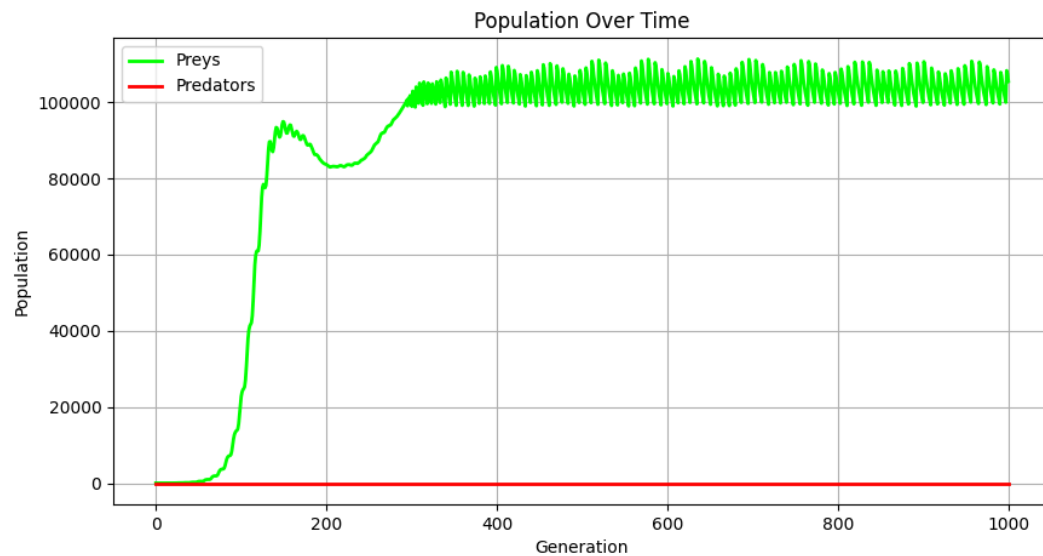
W każdej konfiguracji limit wszystkich organizmów wynosi 100 000.



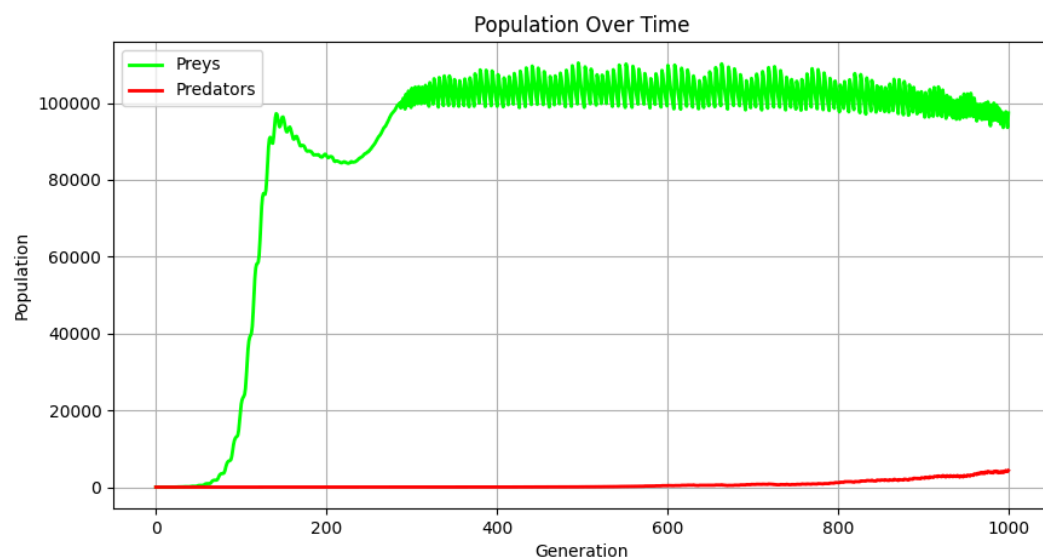
Rysunek 34: Wykres populacji ofiar i drapieżników w czasie (jałowe środowisko, brak drapieżników)



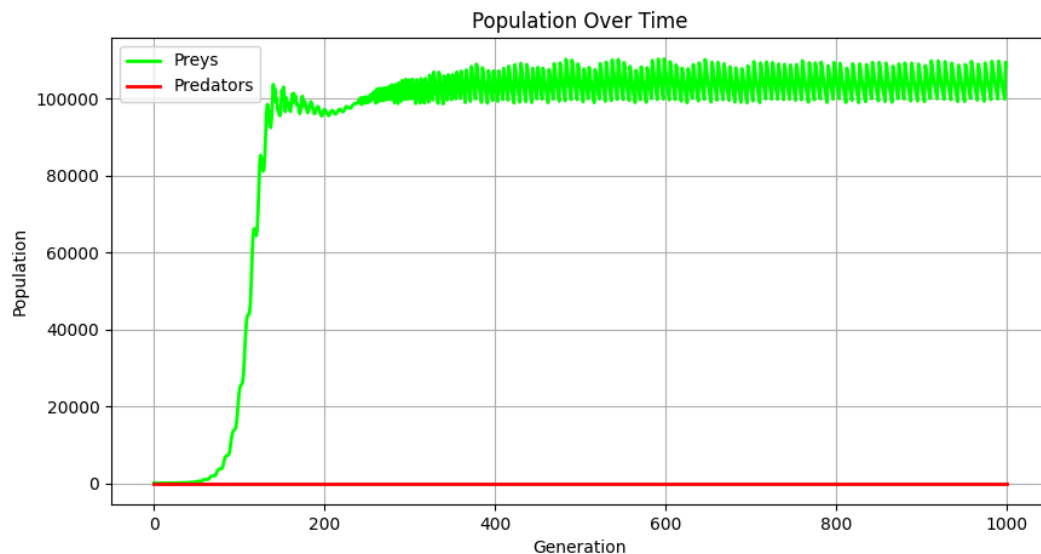
Rysunek 35: Wykres populacji ofiar i drapieżników w czasie (jałowe środowisko)



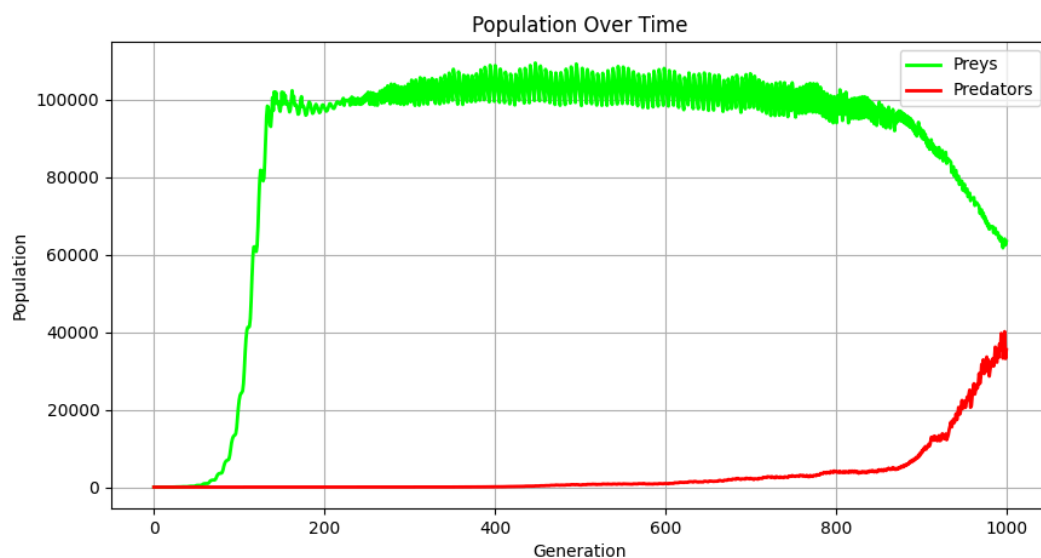
Rysunek 36: Wykres populacji ofiar i drapieżników w czasie (normalne środowisko, brak drapieżników)



Rysunek 37: Wykres populacji ofiar i drapieżników w czasie (normalne środowisko)

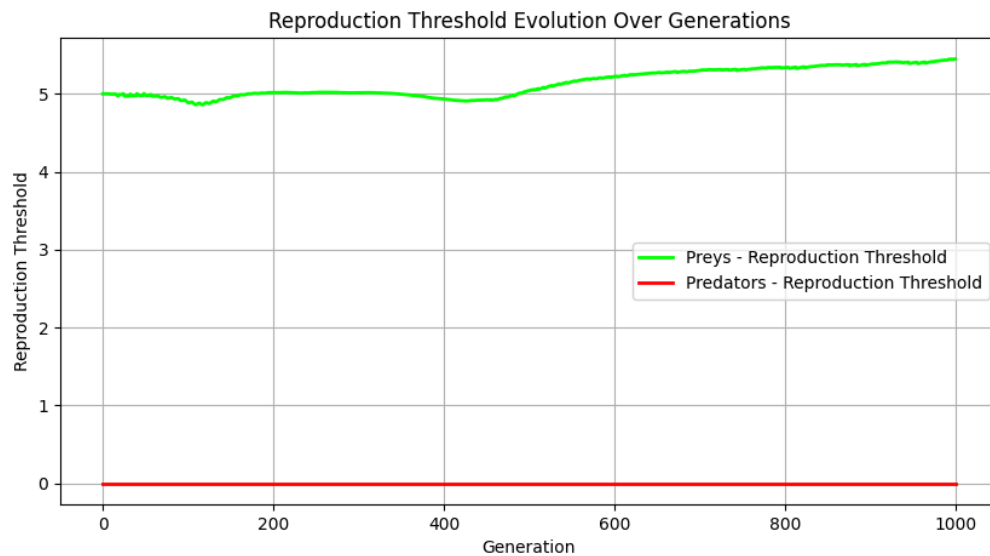


Rysunek 38: Wykres populacji ofiar i drapieżników w czasie (bardzo bogate środowisko, brak drapieżników)

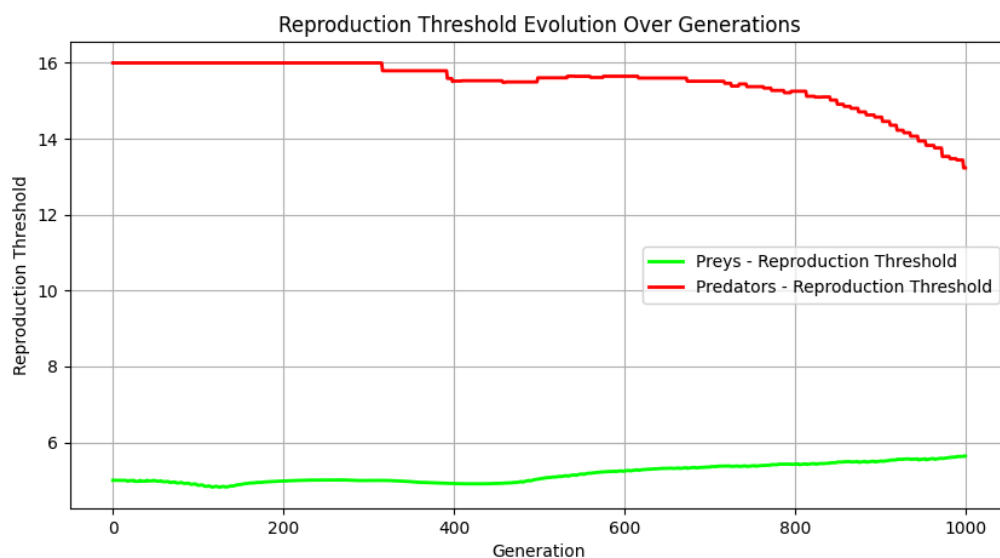


Rysunek 39: Wykres populacji ofiar i drapieżników w czasie (bardzo bogate środowisko)

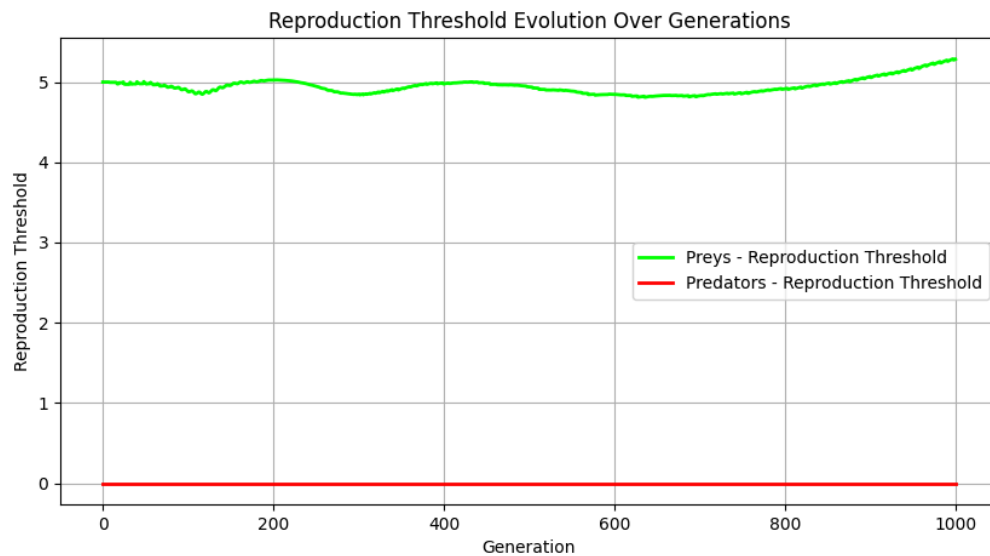
Rys. 34-39 przedstawiają dynamikę populacji ofiar i drapieżników w trzech scenariuszach: jałowe środowisko bez drapieżników (rys. 34), jałowe środowisko z drapieżnikami (rys. 35), normalne środowisko bez drapieżników (rys. 36), normalne środowisko z drapieżnikami (rys. 37), bardzo bogate środowisko bez drapieżników (rys. 38) oraz bardzo bogate środowisko z drapieżnikami (rys. 39). To co można zaobserwować to, że im bardziej bogate środowisko to szybciej populacja ofiar osiąga limit ustawiony w konfiguracji symulacji.



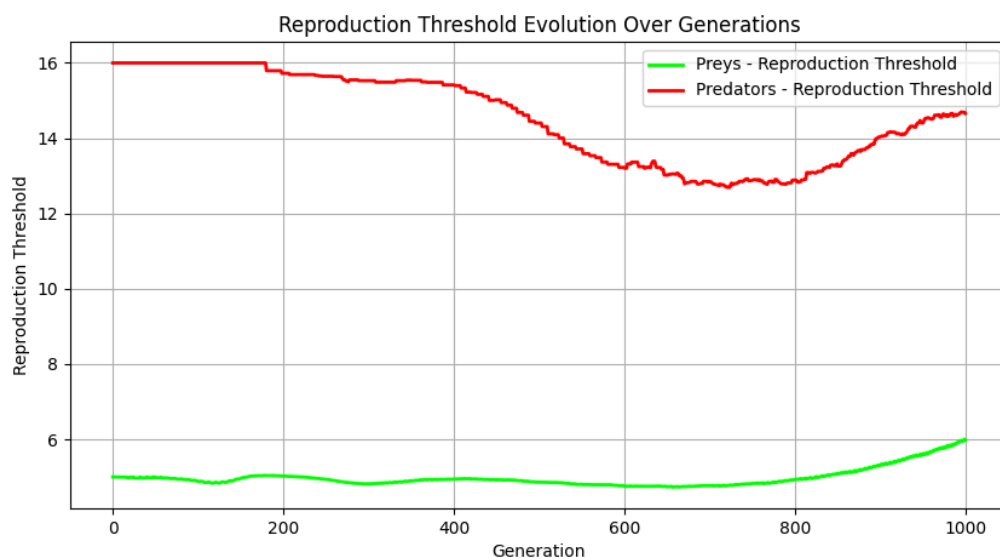
Rysunek 40: Trendy progu reprodukcji ofiar i drapieżników w czasie (jałowe środowisko, brak drapieżników)



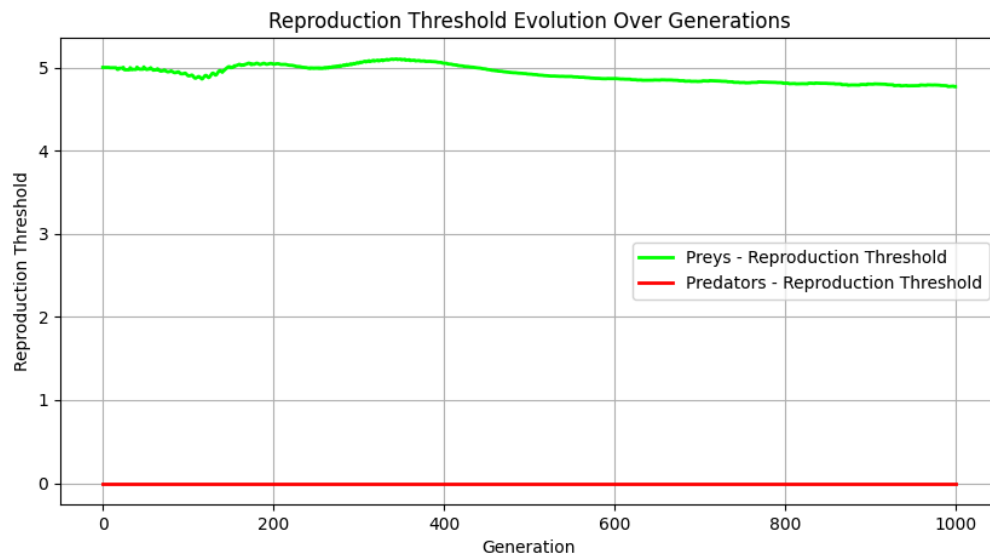
Rysunek 41: Trendy progu reprodukcji ofiar i drapieżników w czasie (jałowe środowisko)



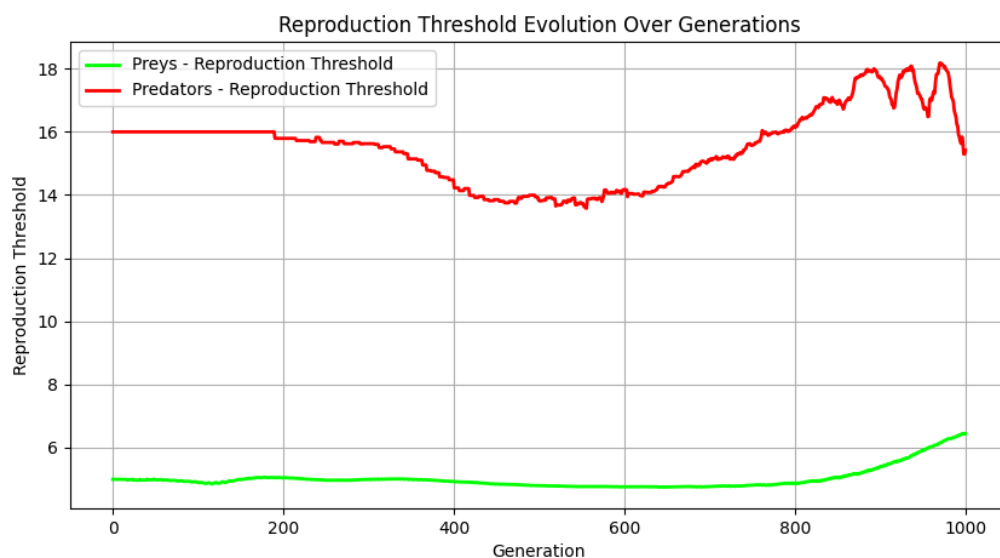
Rysunek 42: Trendy progu reprodukcji ofiar i drapieżników w czasie (normalne środowisko, brak drapieżników)



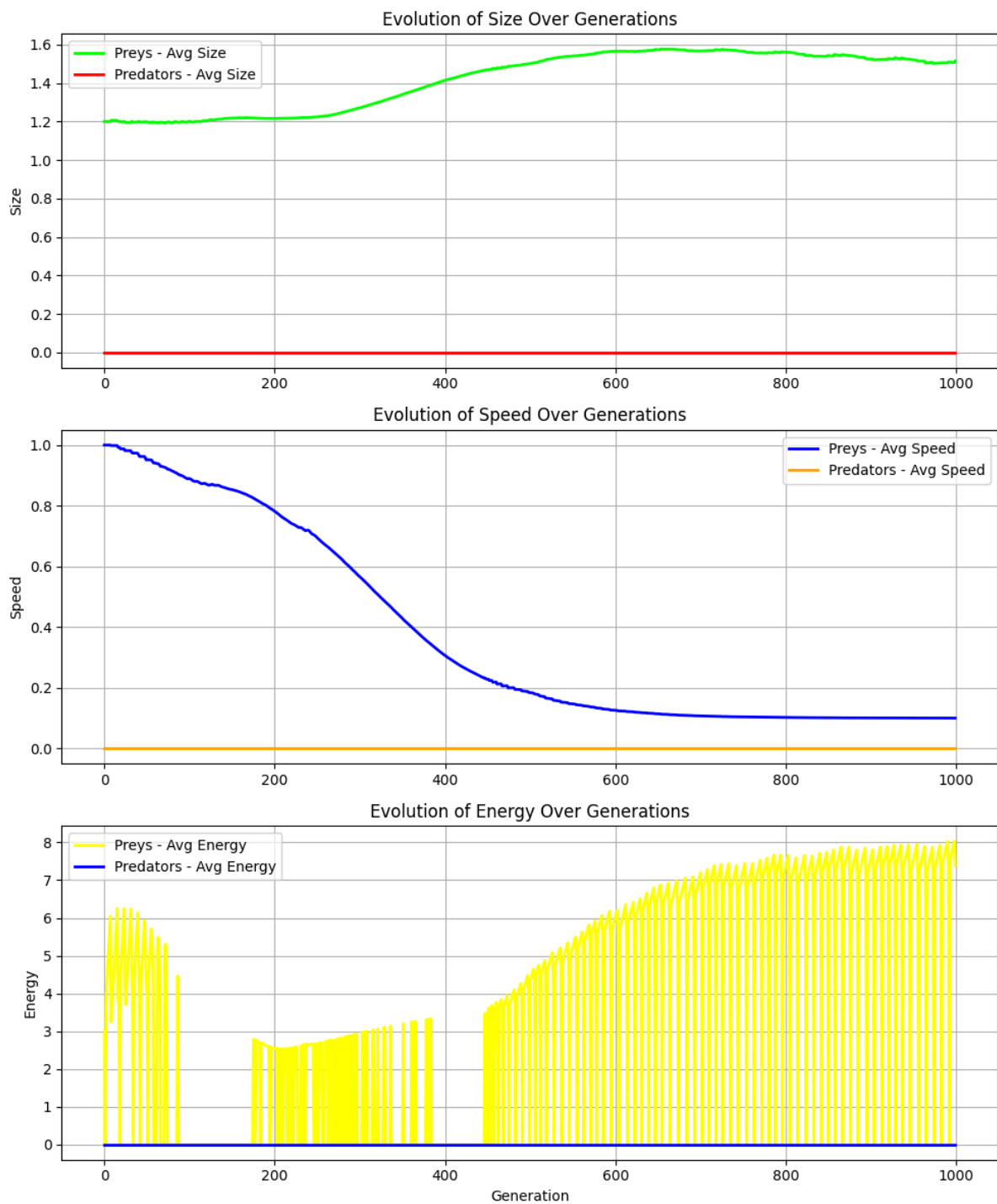
Rysunek 43: Trendy progu reprodukcji ofiar i drapieżników w czasie (normalne środowisko)



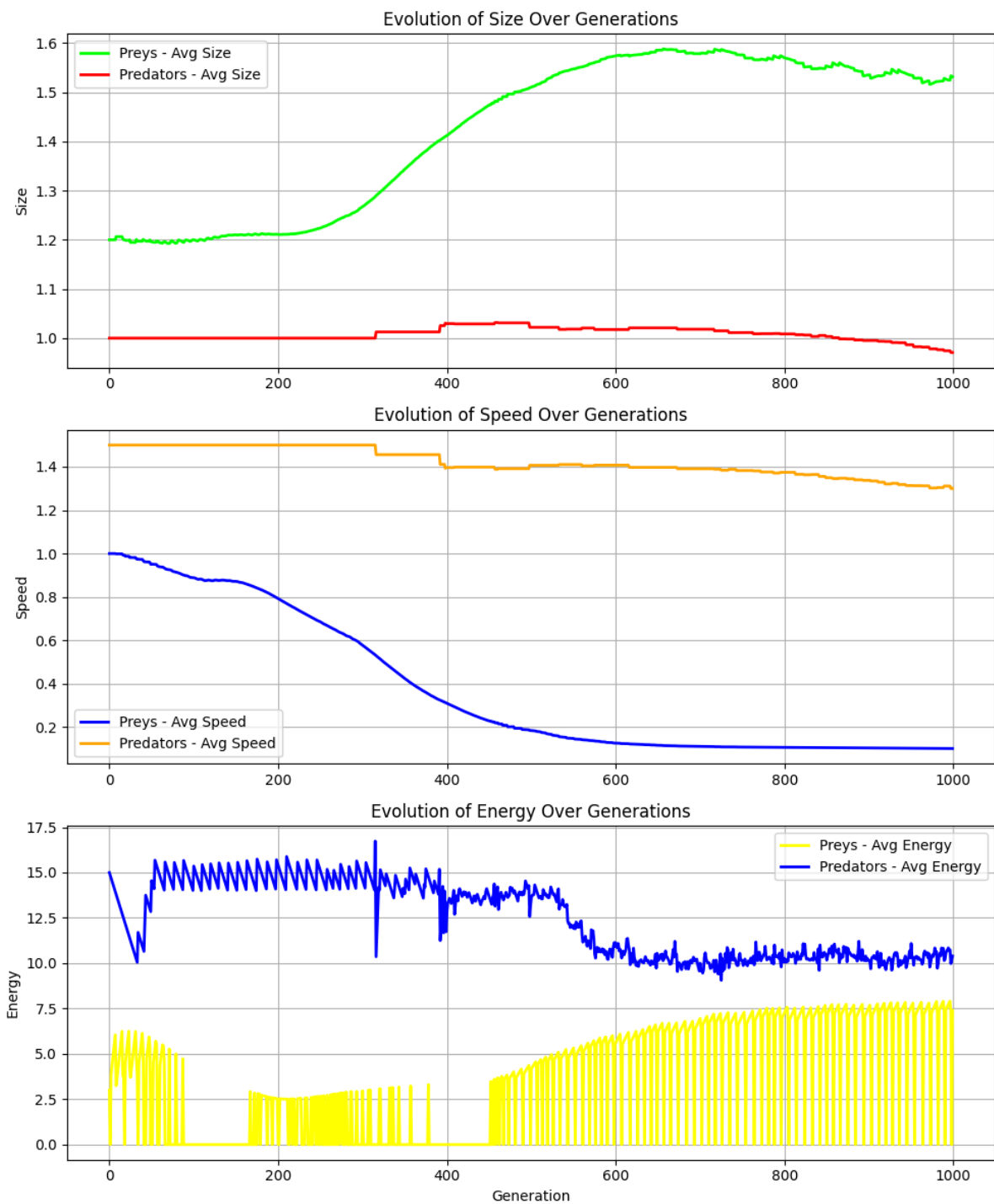
Rysunek 44: Trendy progu reprodukcji ofiar i drapieżników w czasie (bardzo bogate środowisko, brak drapieżników)



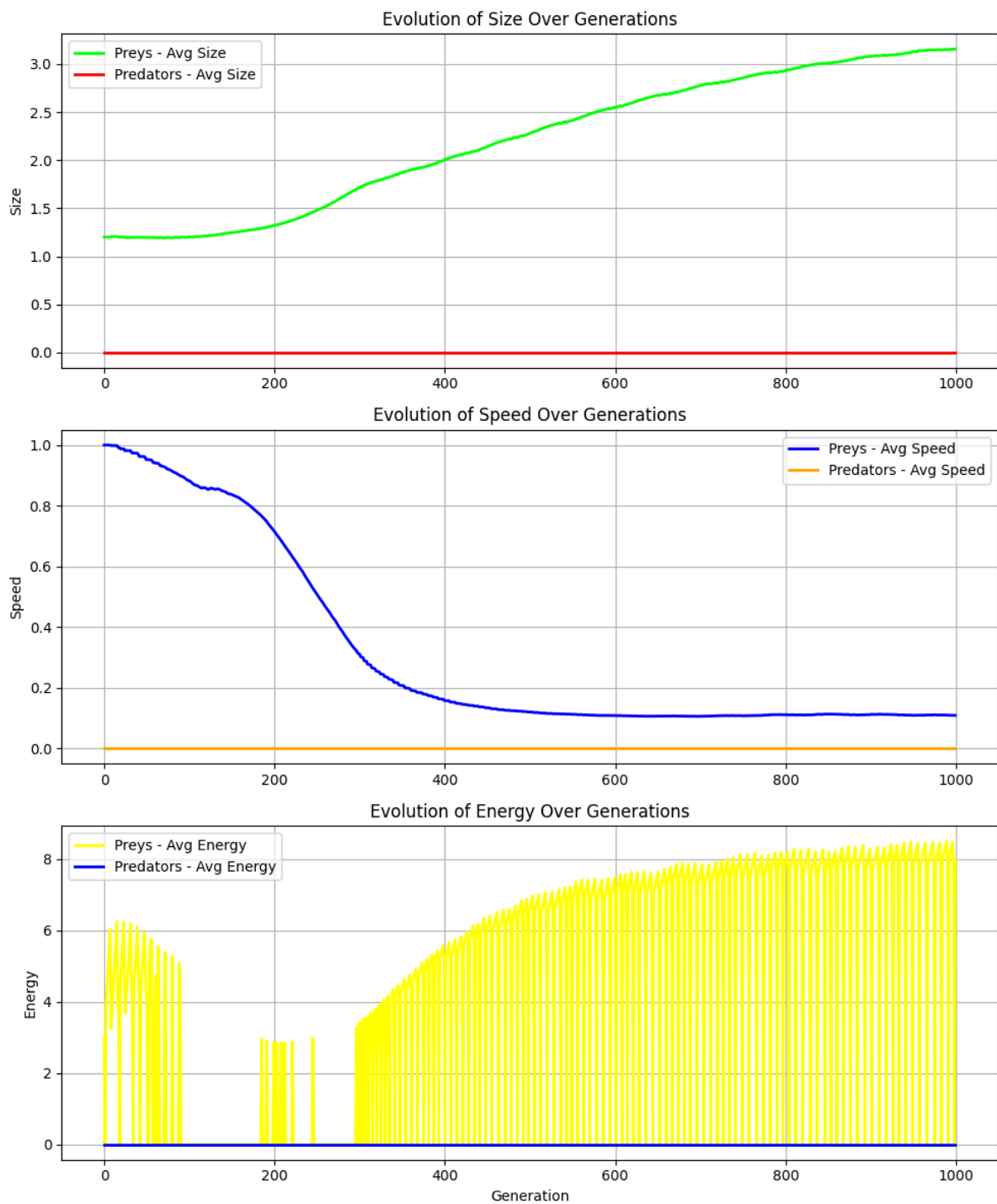
Rysunek 45: Trendy progu reprodukcji ofiar i drapieżników w czasie (bardzo bogate środowisko)



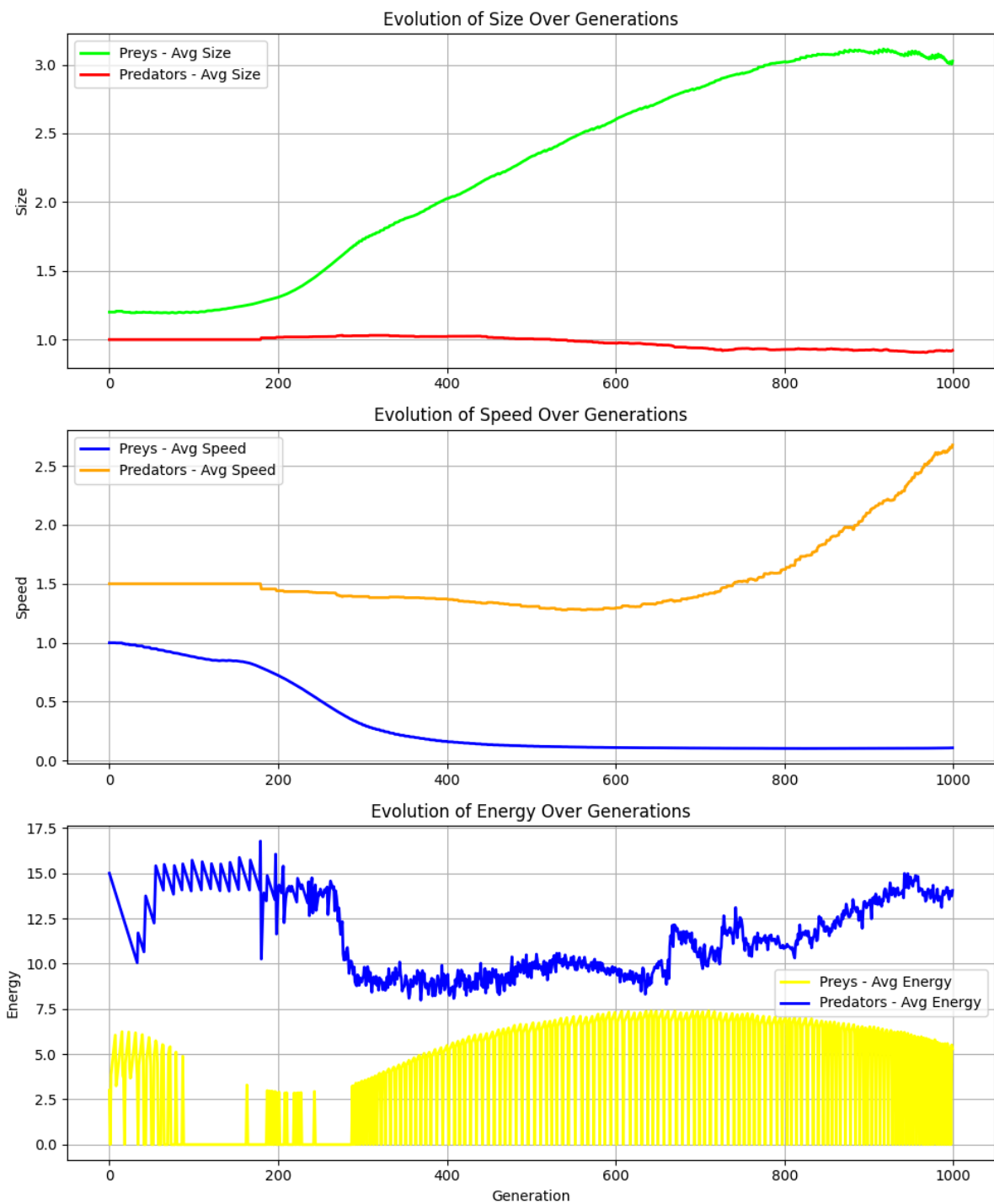
Rysunek 46: Ewolucja cech ofiar i drapieżników w czasie (jałowe środowisko, brak drapieżników)



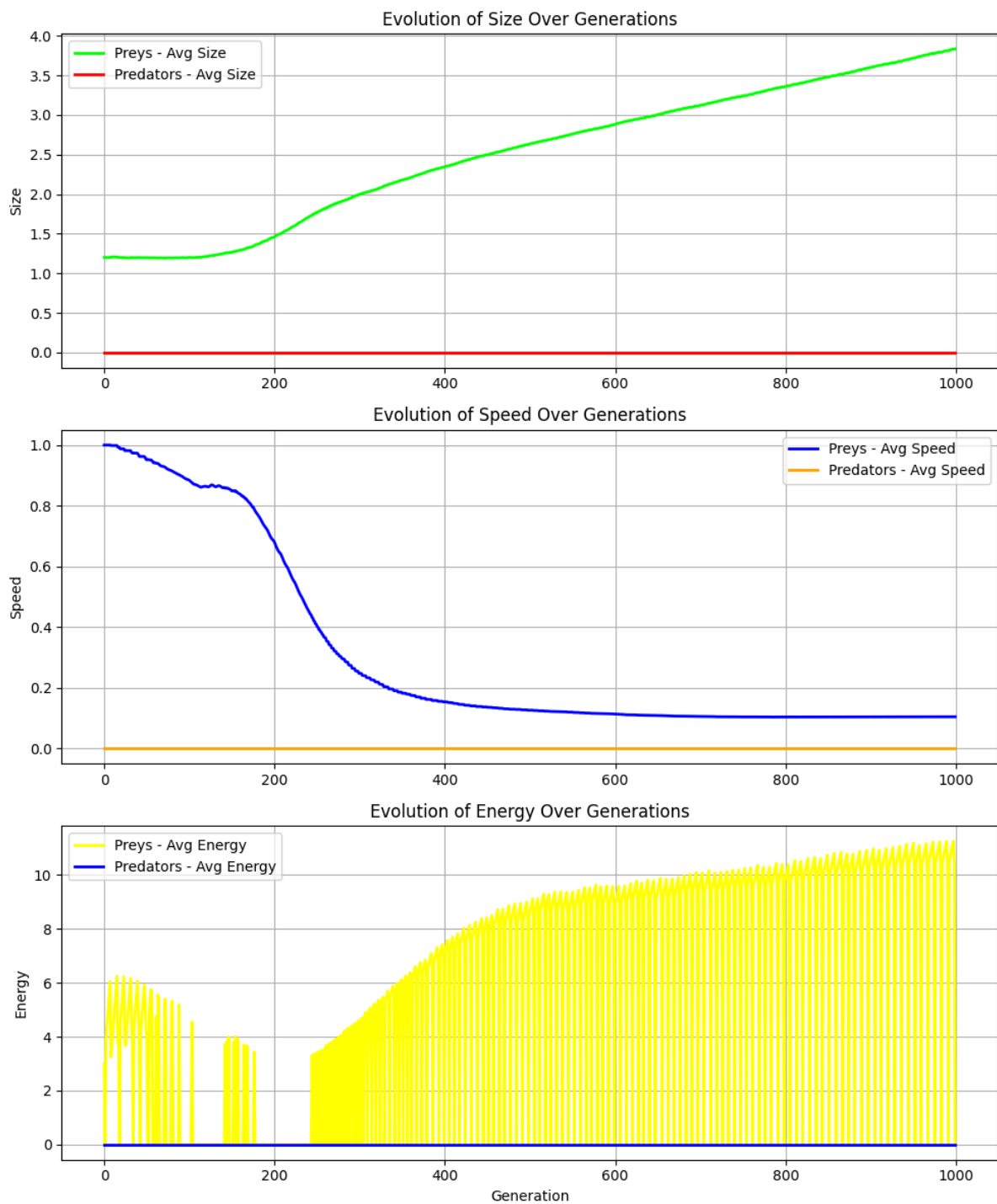
Rysunek 47: Ewolucja cech ofiar i drapieżników w czasie (jałowe środowisko)



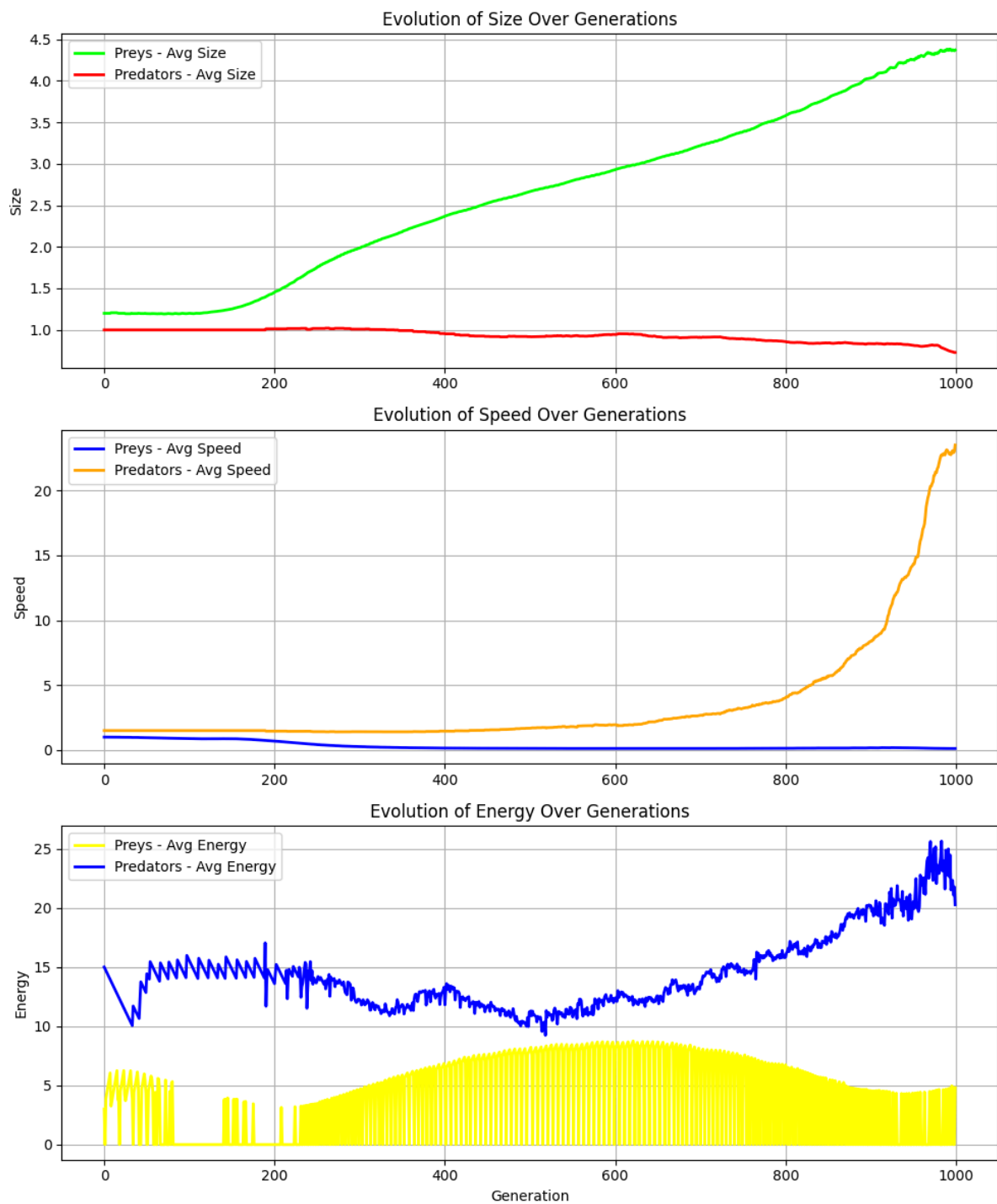
Rysunek 48: Ewolucja cech ofiar i drapieżników w czasie (normalne środowisko, brak drapieżników)



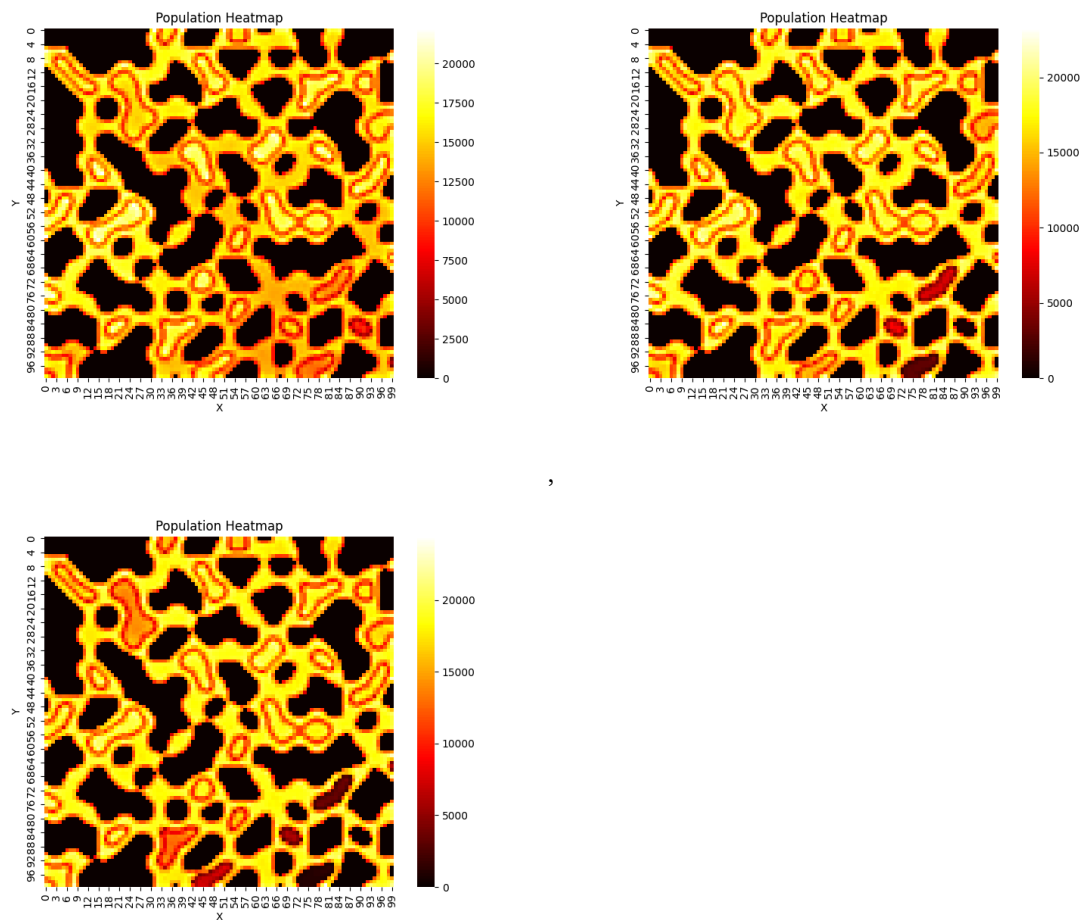
Rysunek 49: Ewolucja cech ofiar i drapieżników w czasie (normalne środowisko)



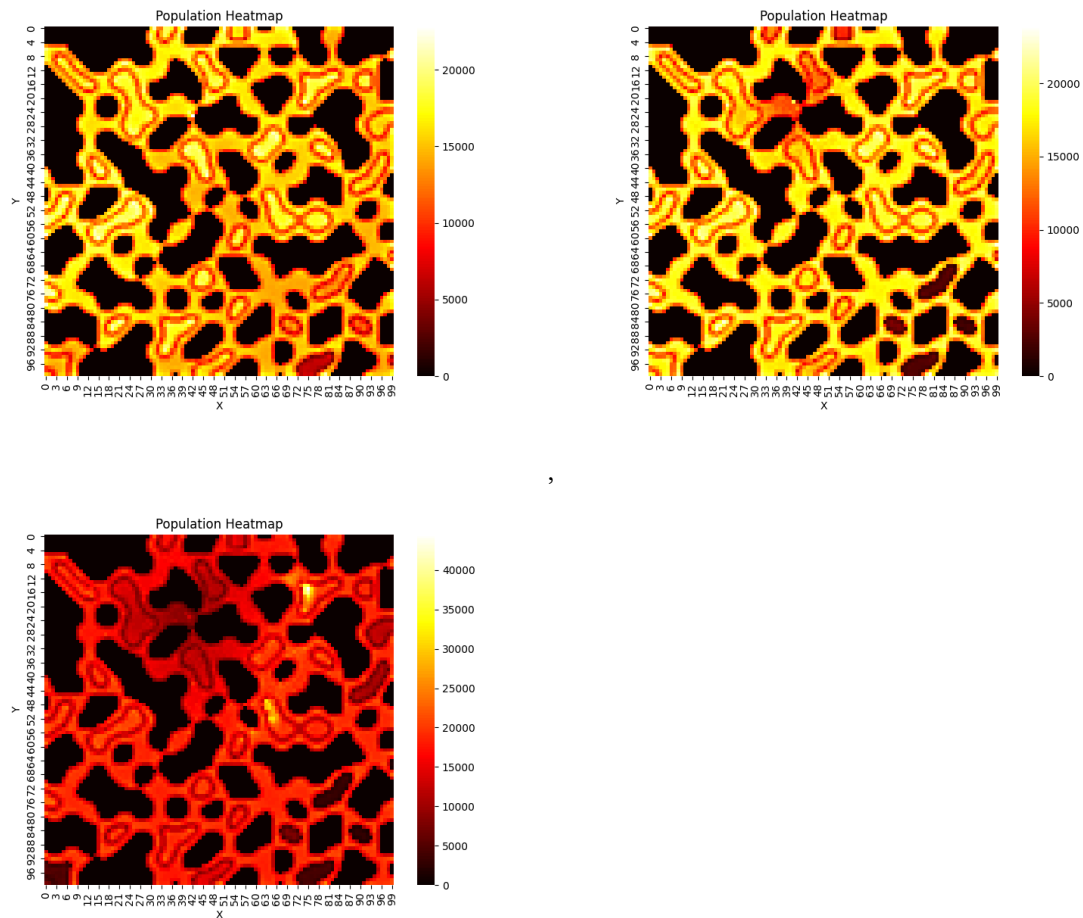
Rysunek 50: Ewolucja cech ofiar i drapieżników w czasie (bardzo bogate środowisko, brak drapieżników)



Rysunek 51: Ewolucja cech ofiar i drapieżników w czasie (bardzo bogate środowisko)

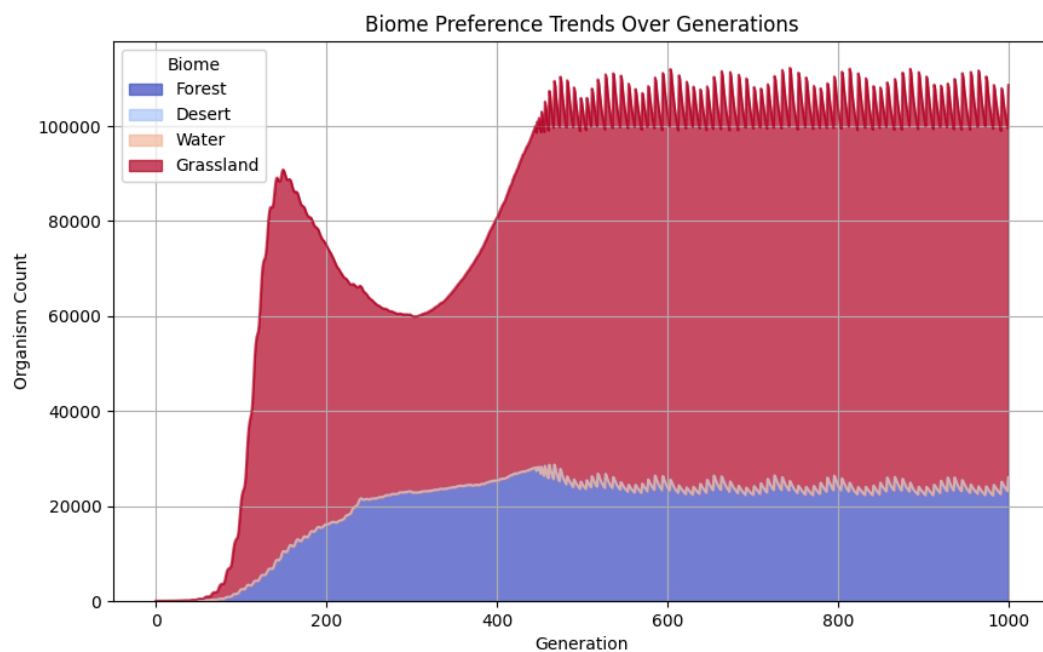


Rysunek 52: Mapa cieplna populacji ofiar w czasie (jałowe środowisko, normalne środowisko, bogate środowisko) bez drapieżników

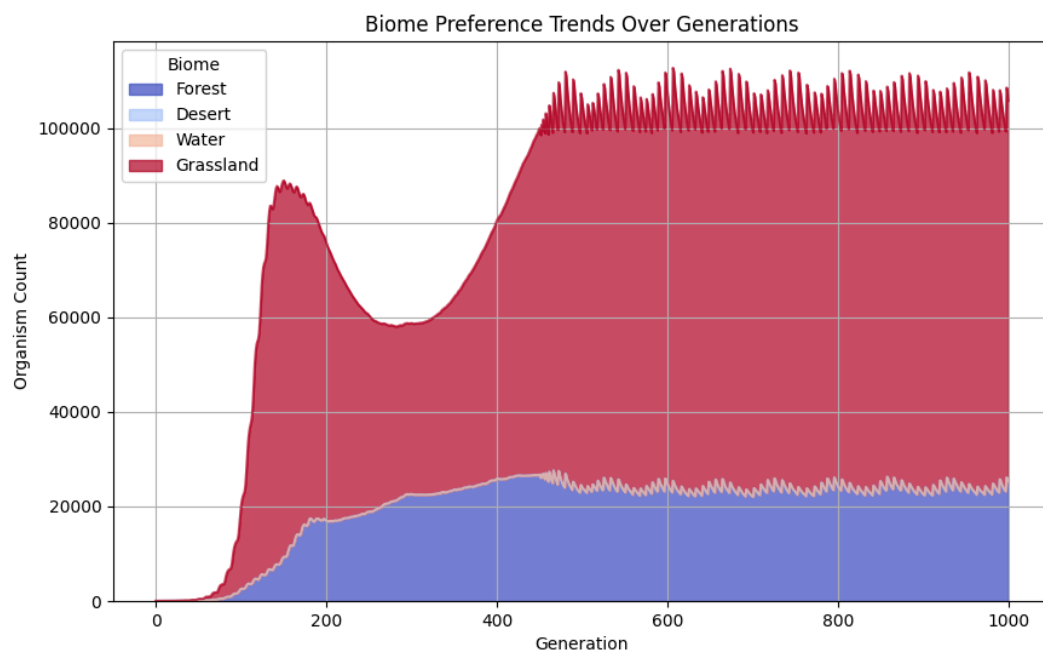


Rysunek 53: Mapa cieplna populacji ofiar i drapieżników w czasie (jałowe środowisko, normalne środowisko, bogate środowisko)

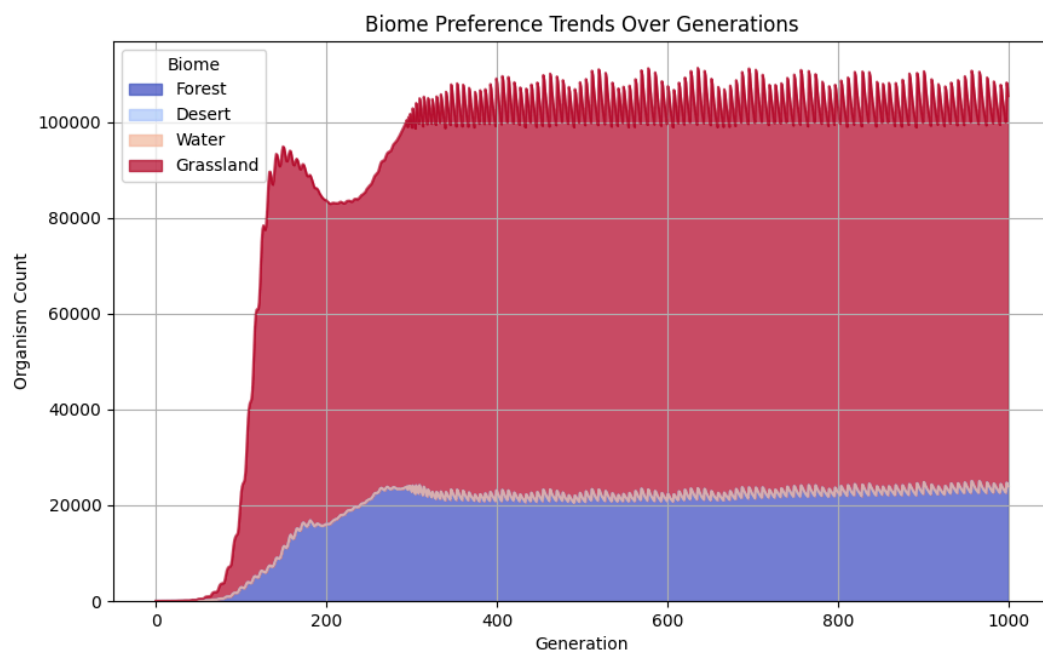
Na rys. 53 w wariancie bogatego środowiska są widoczne dwa skupiska organizmów. Wyższe skupisko jest na polanie, a drugie skupisko jest w lesie.



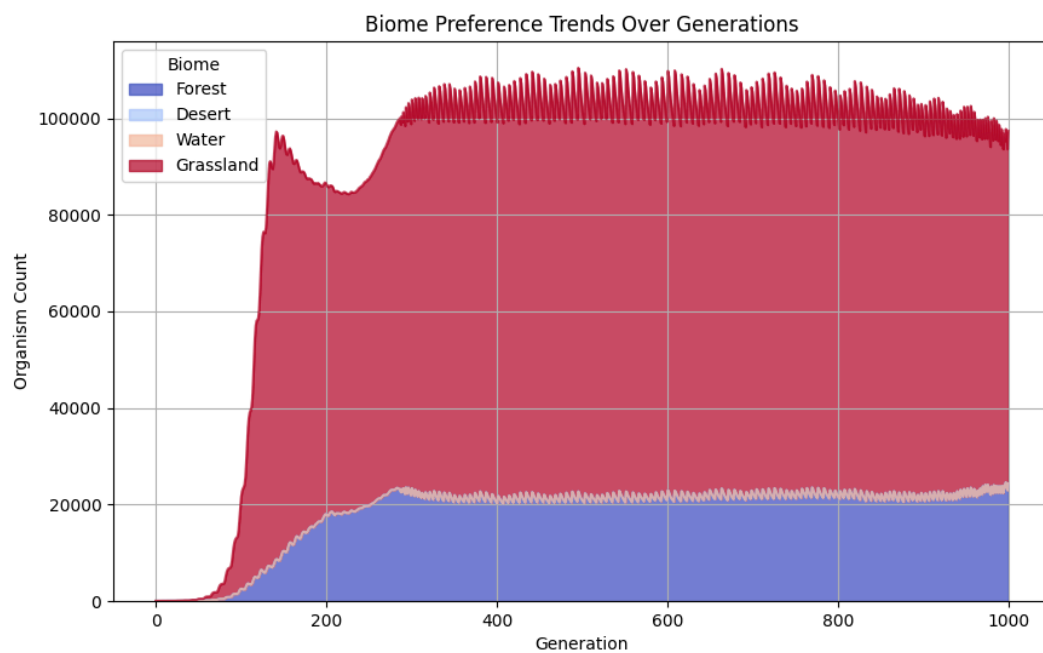
Rysunek 54: Trendy preferencji terenowych ofiar w czasie (jałowe środowisko, brak drapieżników)



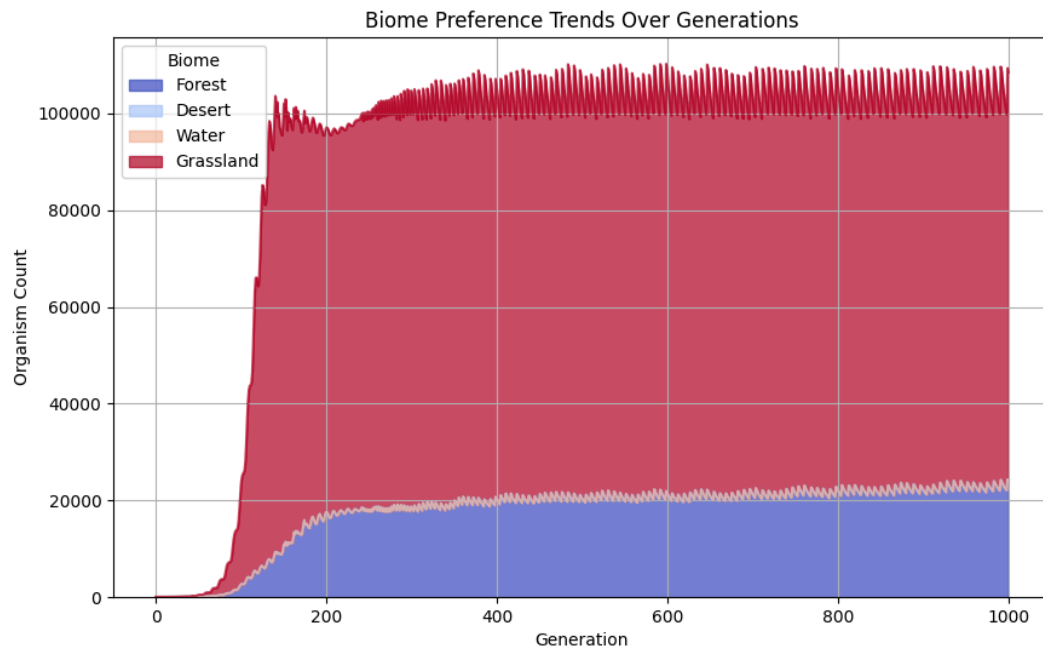
Rysunek 55: Trendy preferencji terenowych ofiar w czasie (jałowe środowisko)



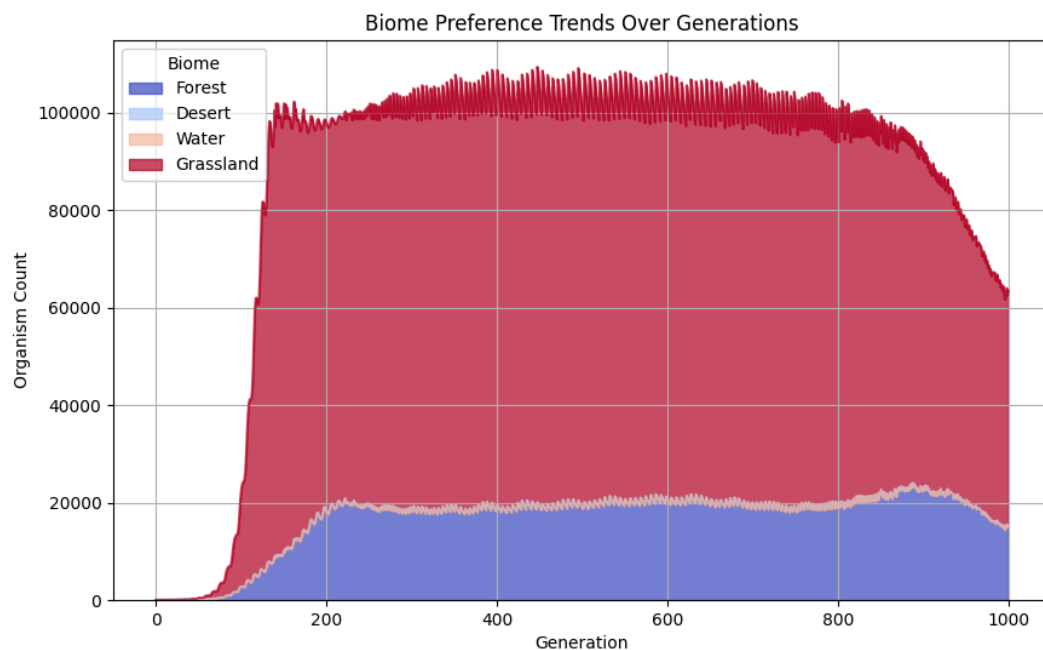
Rysunek 56: Trendy preferencji terenowych ofiar w czasie (normalne środowisko, brak drapieżników)



Rysunek 57: Trendy preferencji terenowych ofiar w czasie (normalne środowisko)



Rysunek 58: Trendy preferencji terenowych ofiar w czasie (bardzo bogate środowisko, brak drapieżników)



Rysunek 59: Trendy preferencji terenowych ofiar w czasie (bardzo bogate środowisko)

Na rys. 54-59 przedstawiono zmiany preferencji terenowych ofiar w czasie dla trzech scenariuszy: jałowe środowisko bez drapieżników (rys. 54), jałowe środowisko z drapieżnikami (rys. 55), normalne środowisko bez drapieżników (rys. 56), normalne środowisko z drapieżnikami (rys. 57), bardzo bogate środowisko bez drapieżników (rys. 58) oraz bardzo bogate środowisko z drapieżnikami (rys. 59). Można zauważyć, że generalnie nie ma widocznych różnic w preferencjach terenowych ofiar w zależności od dostępności jedzenia.

5.5. Podsumowanie wyników i wnioski

Przeprowadzona symulacja pozwoliła zaobserwować, jak różne czynniki środowiskowe i parametry ewolucyjne wpływają na kształtowanie się populacji oraz cech organizmów w czasie. W oparciu o wykonane scenariusze można sformułować następujące wnioski:

5.5.1. Wpływ poziomu mutacji

W tym scenariuszu można zauważyć, że zbyt niski poziom mutacji ogranicza zdolności adaptacyjne organizmów. W takich warunkach populacja drapieżników nie jest w stanie przetrwać i nie potrafi dostosować się do zmieniającego się środowiska. Z kolei zbyt wysoki poziom mutacji prowadził do niestabilności cech, co skutkowało chaotycznym rozkładem przystosowań i częstymi wahaniami populacji. Optymalne rezultaty adaptacyjne pojawiły się przy średnim poziomie mutacji, gdzie ewolucja zachodziła w sposób zrównoważony.

5.5.2. Rola drapieżników

Drugi scenariusz pokazał, że obecność drapieżników pełni istotną rolę w regulacji populacji ofiar. W warunkach bez drapieżników ofiary rozmnażały się w szybkim tempie, co prowadziło do przeludnienia i wyczerpania zasobów. Wprowadzenie umiarkowanej presji drapieżników pomagało ustabilizować populację oraz promowało selekcję korzystnych cech, takich jak szybkość czy niższy próg reprodukcji. Co ciekawe, spodziewałem się, że presja drapieżników wpłynie na ewolucję ofiar w taki sposób, że będą one coraz mniejsze, szybciej będą się przemieszczać i będą miały mniejszy próg reprodukcji. W rzeczywistości okazało się, że ofiary ewoluowały w kierunku większych rozmiarów i ich próg reprodukcji wzrastał. Silna presja drapieżników powodowała wyginięcie ofiar i przez to upadek populacji drapieżników.

5.5.3. Dostępność jedzenia i środowisko

W trzecim scenariuszu zauważyłem, że zróżnicowanie środowiska (jałowe, normalne i bogate) znacząco wpływało na strategię przetrwania ofiar. W bogatych środowiskach ofiary rozwijały się szybciej, ale również szybciej dochodziło do przeludnienia. W środowiskach jałowych przeżywały tylko najlepiej przystosowane jednostki, a tempo reprodukcji i wzrost populacji było znacznie wolniejsze.

6. Podsumowanie

Celem niniejszej pracy było stworzenie modelu symulującego procesy ewolucyjne w środowisku przy użyciu automatów komórkowych oraz architektury ECS. Zrealizowana symulacja pozwoliła zaobserwować, jak złożone zachowania mogą wyłaniać się z prostych reguł lokalnych, oraz jak zmienne środowiskowe i parametry organizmów wpływają na ich ewolucję.

W ramach eksperymentów przeanalizowałem szereg scenariuszy obejmujących m.in. różne poziomy mutacji, obecność drapieżników oraz zróżnicowanie środowiska. Analiza wykazała, że niektóre wyniki były zgodne z intuicją biologiczną (np. stabilizujący wpływ umiarkowanej presji drapieżników), jednak pojawiły się również efekty emergentne, które mnie zaskoczyły.

Największym zaskoczeniem była obserwacja, że pod wpływem silnej presji drapieżników ofiary nie stawały się mniejsze i bardziej efektywne energetycznie (jak można było przypuszczać), lecz wręcz przeciwnie, ewoluowały w stronę większych rozmiarów i wyższego progu reprodukcji. Może to sugerować, że większy rozmiar dawał im przewagę w konkurencji o zasoby. Jest to przykład tzw. wyścigu zbrojeń, gdzie adaptacje jednej grupy organizmów wywołują presję selekcyjną na drugą, prowadząc do nieoczywistych strategii przetrwania.

Ostatecznie praca ta pokazuje, że nawet bardzo uproszczone modele mogą doprowadzić do interesujących i złożonych wyników, które przypominają prawdziwe procesy biologiczne. Warto podkreślić, że zastosowanie języka Rust oraz architektury ECS pozwoliło mi na efektywne zarządzanie dużą liczbą jednostek w czasie rzeczywistym, co czyni stworzoną symulację dobrym punktem wyjścia do dalszych eksperymentów i potencjalnego rozwoju symulacji.

6.1. Kierunki dalszego rozwoju

Ten model można rozszerzyć na wiele sposobów, m. in.:

- Wprowadzenie mechanizmów uczenia lub pamięci (np. zachowanie preferencji terenowych).
- Symulacja więcej niż dwóch gatunków i zależność między nimi (np. pasożytnictwo, konkurencja).
- Wprowadzenie zmiennych klimatycznych (np. sezonowość, globalne ocieplenie).
- Rozszerzenie analiz o miary różnorodności genetycznej czy strategię kooperacyjną.
- Wprowadzenie genomów i genotypów, co pozwoliłoby na bardziej złożoną ewolucję cech.

Bibliografia

- [1] M. Gardner, „Mathematical Games”, *Scientific American*, t. 223, nr 4, s. 120–123, paź. 1970, doi: 10.1038/scientificamerican1070-120.
- [2] P. Bak, C. Tang, i K. Wiesenfeld, „Self-organized criticality: An explanation of the 1/f noise”, *Phys. Rev. Lett.*, t. 59, nr 4, s. 381–384, lip. 1987, doi: 10.1103/PhysRevLett.59.381.
- [3] Z. Olami, H. J. S. Feder, i K. Christensen, „Self-organized criticality in a continuous, nonconservative cellular automaton modeling earthquakes”, *Phys. Rev. Lett.*, t. 68, nr 8, s. 1244–1247, luty 1992, doi: 10.1103/PhysRevLett.68.1244.
- [4] A. Ilachinski, *Cellular Automata*, 0. wyd. WORLD SCIENTIFIC, 2001. doi: 10.1142/4702.
- [5] C. G. Langton, „Studying artificial life with cellular automata”, *Physica D: Nonlinear Phenomena*, t. 22, nr 1, s. 120–149, 1986, doi: [https://doi.org/10.1016/0167-2789\(86\)90237-X](https://doi.org/10.1016/0167-2789(86)90237-X).
- [6] P. Bak i K. Sneppen, „Punctuated equilibrium and criticality in a simple model of evolution”, *Phys. Rev. Lett.*, t. 71, nr 24, s. 4083–4086, grudz. 1993, doi: 10.1103/PhysRevLett.71.4083.
- [7] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993. doi: 10.1093/oso/9780195079517.001.0001.
- [8] K. Perlin, „An image synthesizer”, *SIGGRAPH Comput. Graph.*, t. 19, nr 3, s. 287–296, lip. 1985, doi: 10.1145/325165.325247.

7. Kod źródłowy

Repozytorium z kodem źródłowym dostępne jest pod adresem: https://github.com/GKaszewski/evolution_cellular_automata