Modelowanie i analiza systemów informatycznych

dokumentacja projektu

Oliwia Gowor, Hubert Kwiatkowski, Miłosz Prządka

18 grudnia 2024

Część I

Opis programu

Program symuluje dynamiczną grę strategiczną opartą na planszy, w której gracze (zarówno użytkownicy, jak i agenci) rywalizują ze sobą. poprzez przemieszczanie się po planszy w celu osiągnięcia celu lub eliminacji przeciwników. Każda rozgrywka odbywa się na siatce reprezentującej planszę, a uczestnicy podejmują decyzje o ruchach na podstawie danych o otoczeniu. Gra może być używana do celów rozrywkowych, edukacyjnych (np. nauka programowania sztucznej inteligencji) lub jako platforma testowa do rozwijania i optymalizacji algorytmów AI.

Instrukcja wdrożenia

Program można uruchomić lokalnie, korzystając z terminala. Wdrożenie jest szybkie i wymaga spełnienia wymagań systemowych: środowiska Python w wersji 3.9 lub nowszej oraz biblioteki ‘curses‘, która jest standardowo dostępna w systemach Linux.

Kroki:

1. Pobranie kodu ze źródła

   Pierwszym krokiem jest skopiowanie projektu na lokalny komputer, np. używając ‘git clone‘ lub innej metody dystrybucji kodu.

2. Instalacja Python 3.9+

   Należy upewnić się, że na komputerze zainstalowana jest wersja Pythona 3.9 lub nowsza:

   - W systemie Linux:

   (a) **sudo apt update**

   (b) **sudo apt install python3**

   - W systemie Windows: Należy pobrać instalator Pythona i wykonać instalację.

3. Uruchomienie programu

   Ostatnim krokiem jest przejście do katalogu projektu w terminalu i uruchomienie aplikacji poleceniem:

   python main.py

4. Dodatkowe informacje

   - Jeśli wykorzystywany jest system Windows, konieczne może być zainstalowanie alternatywnej biblioteki obsługującej ‘curses‘, np. ‘windows-curses‘, za pomocą polecenia:

   pip install windows-curses

- W systemach Linux i macOS ‘curses‘ jest instalowane domyślnie.

Program jest gotowy do użycia natychmiast po uruchomieniu.

0.1 Instrukcja obsługi

Plansza gry oraz instrukcje dotyczące sterowania są wyświetlane bezpośrednio w terminalu. Nawigacja po programie odbywa się za pomocą wpisywania odpowiednich znaków w terminalu. Gracze sterują ruchem za pomocą klawiszy (`W`, `S`, `A`, `D`), wybierając jeden z czterech kierunków (odpowiednio: góra, dół, lewo, prawo). Agenci podejmują decyzje w oparciu o algorytmy oceny sytuacji na planszy, takie jak analiza bliskości przeciwników czy unikanie kolizji. Agenci są autonomiczni i nie są im przekazywane żadne obiekty, wywoływane są tylko publiczne funkcje. Agentom losowo zostaje przypisana jedna z dostępnych taktyk gry (standardowa lub agresywna).

W grze dostępne są dwa tryby:

- **Gracz vs. Agenci:** Użytkownik wciela się w rolę jednego z graczy i rywalizuje z agentami kontrolowanymi przez algorytmy.
- **Agenci vs. Agenci:** Rozgrywka odbywa się wyłącznie pomiędzy agentami, a użytkownik może obserwować ich zachowanie w celach analizy i porównań.

Gra kończy się, gdy jeden z graczy spełni warunki zwycięstwa poprzez eliminację wszystkich przeciwników.

Część II

Opis działania

System gry opiera się na matematycznym modelu siatki, gdzie plansza to macierz o wymiarach 20x20. Każdy gracz, będący obiektem na planszy, posiada pozycję , kierunek ruchu oraz stan (aktywny, zniszczony).

Model matematyczny systemu i jego komponentów

Model matematyczny gry jest zbudowany w oparciu o siatkę (planszę) oraz zestaw reguł opisujących interakcje między graczami i elementami planszy.

1. Plansza
   Plansza jest reprezentowana jako macierz dwuwymiarowa o wymiarach , gdzie:

   M: liczba wierszy (wysokość planszy),

   N: liczba kolumn (szerokość planszy).
   Każda komórka planszy może przyjąć jedną z następujących wartości:

   - "..": pusta przestrzeń,
   - P_k: pozycja gracza,
   - A_k: pozycja agenta.

2. Pozycja gracza/agenta
   Pozycja każdego gracza jest definiowana jako wektor współrzędnych , gdzie x_k to kolumna, a y_k to wiersz na planszy. Ruch gracza polega na zmianie jego pozycji zgodnie z wektorem przesunięcia d, tj.:

   $\mathbf{p}_k^{nowa} = \mathbf{p}_k + \mathbf{d}$, gdzie d:

   - UP$\rightarrow$(0,1),
   - DOWN$\rightarrow$(0,+1)
   - LEFT$\rightarrow$(1,0),
   - RIGHT$\rightarrow$(+1,0).

3. Decyzje agentów:
   Funkcje decyzyjne, takie jak:

   - Funkcja bazowa opiera się na ocenie dostępnych kierunków D w postaci:
     $D = \{(x', y')x' = x + \delta x, y' = y + \cdot i \cdot M[x'][y'] =' ..'\}$.
   - Funkcja oceny punktowej dla kierunku d: $Score(d) = Reward(d) - Penalty(d)$.

4. Zasady kolizji:
   Gracz koliduje, jeśli:

   - Wyjdzie poza granice planszy: $x < 0xWy < 0yH$.
   - Wykona ruch na zajęte pole: $M[x][y]'..'$.

Technika rozwiązania i projektowanie systemu
Program został zaprojektowany w oparciu o modularną architekturę, co ułatwia rozwój, testowanie i modyfikację komponentów. System wykorzystuje także wzorce projektowe.

1. **Wzorce projektowe**:
   - **Strategia**: Decyzje agentów implementowane jako różne strategie (`Agent`, `AggressiveAgent`).
   - **Fabryka**: Metoda `create_game` generuje graczy w zależności od trybu gry.
   - **MVC (Model-View-Controller)**: Plansza i mechanika gry stanowią Model, interfejs terminalowy to Widok, a sterowanie graczami (w tym decyzje) to Kontroler.
2. **Podział na komponenty**:
   - **Gracz**: Klasa bazowa, wspierająca ruch i kolizje.
   - **Agent**: Klasa potomna z różnymi strategiami decyzyjnymi.
   - **Watchtower**: Klasa odpowiedzialna za analizę planszy i rekomendacje ruchów.
   - **Game**: Klasa centralna, integrująca planszę, graczy i logikę gry.

0.1.1

Techniki implementacyjne
Algorytmy sterowania agentami opierają się na przeszukiwaniu lokalnym, tj. analizie możliwych ruchów i ich oceny w danej turze. Gra działa w trybie turowym, gdzie w każdej turze każdy gracz (lub agent) wykonuje jeden ruch.

Zabezpieczenia i odporność na ataki

1. **Walidacja danych wejściowych**:
   - Funkcje sprawdzają, czy ruch jest w granicach planszy (`is_within_bounds`).
   - Gracze są umieszczani na pustych polach przy użyciu `random_empty_position`.
2. **Zapobieganie błędom logicznym**: Kolizje i błędy ruchu są obsługiwane przez flagę `crashed`.
3. **Odporność na złośliwe zachowania**:
   - Gra jest izolowana, bez możliwości wprowadzenia kodu zewnętrznego.
   - Mechanizmy wykluczają ruchy poza planszę i zapobiegają konfliktom pozycji.

Algorytmy

Standardowy wybór kierunku (get_suggestion) Metoda ta sugeruje ruch agenta, który jest oparty na analizie dostępnych kierunków w oparciu o ich ocenę. W przypadku naruszenia reguł gry, wybierany jest bezpieczny kierunek.

Ocena kierunków (_evaluate_directions):

- Dla każdej możliwej pozycji (np. ruch "UP", "DOWN", "LEFT", "RIGHT") obliczana jest punktacja.
- Kierunki są oceniane według:
  - Liczby wolnych pól w otoczeniu (im więcej wolnych pól, tym lepiej).
  - Kar za obecność innych agentów w bliskiej odległości (karana jest odległość 1 lub 2 pola).
- Wynikiem jest kierunek o najwyższej punktacji. W razie braku jednoznacznego wyniku wybierany jest losowy ruch.

Data: agent, game
Result: Najlepszy kierunek ruchu
$x, y \leftarrow$ pozycja agenta (agent.position);
directions
$\leftarrow \{\text{“}UP\text{”} : (0, -1), \text{“}DOWN\text{”} : (0, 1), \text{“}LEFT\text{”} : (-1, 0), \text{“}RIGHT\text{”} : (1, 0)\}$;
best_direction $\leftarrow$ *null*;
best_score $\leftarrow -\infty$;
foreach kierunek, (dx, dy) *w directions* **do**
    $new\_x \leftarrow x + dx$;
    $new\_y \leftarrow y + dy$;
    if $0 \leq new\_x < game.width$ *i* $0 \leq new\_y < game.height$ **then**
        if game.board[new_y][new_x] == “..” **then**
            $score \leftarrow evaluate\_move(new\_x, new\_y, agent, game)$;
            if $score > best\_score$ **then**
                $best\_score \leftarrow score$;
                $best\_direction \leftarrow$ kierunek;
            end
        end
    end
end
return *best_ direction, jeśli istnieje, w przeciwnym razie losowy kierunek z directions*;

Algorithm 1: Algorytm wyboru najlepszego kierunku ruchu dla agenta.

Sprawdzenie zgodności z zasadami (is_within_bounds): Upewnia się, że ruch nie wychodzi poza granice planszy.

Data: direction, x, y, game
Result: Czy nowa pozycja mieści się w granicach planszy?
$dx, dy \leftarrow directions[direction]$;
$new\_x \leftarrow x + dx$;
$new\_y \leftarrow y + dy$;
return $0 \leq new\_x < game.width$ *oraz* $0 \leq new\_y < game.height$;

Algorithm 2: Sprawdzenie, czy nowa pozycja mieści się w granicach planszy.

Bezpieczny kierunek (get_safe_direction): Wybierany jest taki kierunek, który nie tylko mieści się w granicach planszy, ale także prowadzi na wolne pole.

Data: directions, x, y, game
Result: Pierwszy kierunek prowadzący do wolnego pola w granicach planszy
foreach kierunek, (dx, dy) *w directions* **do**
    $new\_x \leftarrow x + dx$;
    $new\_y \leftarrow y + dy$;
    if $0 \leq new\_x < game.width$ *oraz* $0 \leq new\_y < game.height$ *oraz*
     game.board[new_y][new_x] == ".." **then**
        return *kierunek*;
    end
end

Algorithm 3: Znajdowanie pierwszego wolnego kierunku w granicach planszy.

Agresywny wybór kierunku (get_aggressive_suggestion) Metoda ta również sugeruje ruch agenta, ale skupia się na bardziej ofensywnej strategii, nagradzając zbliżanie się do przeciwników oraz próbę ich ograniczenia.

Proces analizy Ocena kierunków (_evaluate_directions_aggressive): Działa podobnie jak w przypadku standardowej analizy, ale ocenia ruchy na podstawie innych kryteriów:

- Nagroda za bliskość do przeciwników (większa za bezpośrednią bliskość, mniejsza za odległość do 3 pól).
- Ocena potencjalnych ruchów, które mogą ograniczyć manewry przeciwnika (mechanizm "trap potential").
- Kara za wybór pól blisko krawędzi planszy (mniejsza możliwość manewrowania).
- Kara za zwiększanie dystansu od przeciwnika.

Data: directions, x, y, game, agent
Result: Najlepszy kierunek na podstawie oceny agresywnej
best_direction ← *null*;
best_score ← $-\infty$;
foreach kierunek, (dx, dy) *w directions* **do**
  new_x ← $x + dx$;
  new_y ← $y + dy$;
  if $0 \leq new\_x < game.width$ *oraz* $0 \leq new\_y < game.height$ **then**
    if game.board[new_y][new_x] == ".." **then**
      score ← *evaluate_aggressive_move*(*new_x*, *new_y*, *agent*, *game*);
      if *score* > *best_score* **then**
        best_score ← *score*;
        best_direction ← kierunek;
      end
    end
  end
end
return *best_ direction, jeśli istnieje, w przeciwnym razie losowy kierunek z directions*;

Algorithm 4: Algorytm wyboru najlepszego kierunku ruchu z oceną agresywną.

Sprawdzenie zgodności z zasadami: Analogiczne do standardowego wyboru, ale z priorytetem dla strategii agresywnej.

Ruch maksymalizujący ofensywę (get_best_aggressive_move): Szuka najlepszego ruchu na podstawie najwyższej punktacji w ocenie agresywnej.

Data: directions, x, y, game, agent
Result: Najlepszy kierunek na podstawie agresywnej strategii
best_direction ← *null*;
best_score ← $-\infty$;
foreach kierunek, (dx, dy) *w directions* **do**
  new_x ← $x + dx$;
  new_y ← $y + dy$;
  if $0 \leq new\_x < game.width$ *oraz* $0 \leq new\_y < game.height$ **then**
    if game.board[new_y][new_x] == ".." **then**
      score ← *evaluate_aggressive_move*(*new_x*, *new_y*, *agent*, *game*);
      if *score* > *best_score* **then**
        best_score ← *score*;
        best_direction ← kierunek;
      end
    end
  end
end
return *best_ direction, jeśli istnieje, w przeciwnym razie losowy kierunek z directions*;

Algorithm 5: Algorytm wyboru najlepszego kierunku ruchu w oparciu o agresywną strategię.

Implementacja systemu

Podział na pliki

System został zaprojektowany z podziałem na cztery główne moduły: `agent.py`, `watchtower.py`, `game.py` oraz `main.py`. Każdy z plików pełni odrębną funkcję w systemie, co ułatwia jego rozwój i utrzymanie.

`agent.py`

Ten moduł zawiera klasy definiujące zachowanie graczy, zarówno kontrolowanych przez użytkownika, jak i przez komputer. W szczególności:

- Klasa `Player`: Reprezentuje podstawowego gracza w grze. Utrzymuje informacje o stanie gracza, takie jak pozycja, symbol na planszy, aktualny kierunek ruchu i status kolizji. Obsługuje funkcję move, która aktualizuje pozycję gracza na planszy w oparciu o obecny kierunek.
- Klasa `Agent`: Dziedziczy po klasie Player i implementuje algorytmy decydujące o kierunku ruchu agenta. Funkcja decide_move pozwala agentowi na podjęcie decyzji o ruchu, opierając się na sugestiach z Watchtower.
- Klasa `AggressiveAgent`: Rozszerza klasę Agent, modyfikując jej logikę w kierunku bardziej agresywnego stylu gry. Agresywny agent stara się podejmować decyzje, które zwiększają presję na przeciwników, np. zbliżanie się do nich lub próby ich blokowania.

`watchtower.py`

Moduł ten pełni funkcję "wieży strażniczej", która ocenia sytuację na planszy i generuje sugestie ruchu dla agentów.

- Funkcja `get_suggestion` sugeruje ruch w sposób defensywny, unikając kolizji i starając się utrzymać agenta w bezpiecznej pozycji.
- Funkcja `get_aggressive_suggestion` ocenia najlepszy możliwy ruch w sposób bardziej ofensywny, np. poprzez zbliżanie się do przeciwników.
- Wbudowane metody, takie jak `is_within_bounds` czy `get_safe_direction`, zapewniają, że ruchy są zgodne z zasadami gry i nie wychodzą poza granice planszy.

`game.py`

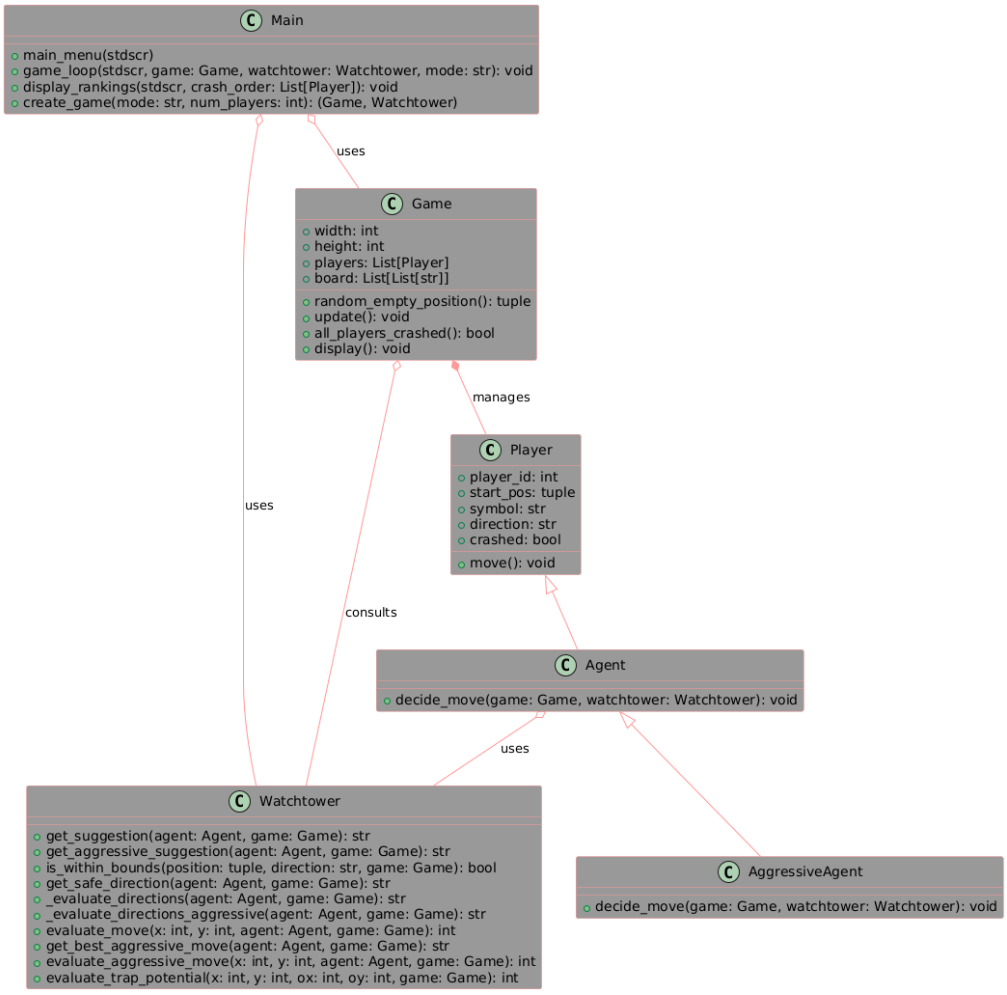
Ten moduł definiuje logikę gry oraz operacje na planszy.

- Klasa `Game`: Tworzy planszę gry o określonych wymiarach i inicjuje graczy na losowych, pustych pozycjach. Odpowiada za aktualizację stanu planszy w każdej turze oraz za sprawdzanie, czy gra się zakończyła. Funkcja update przesuwa graczy, sprawdza kolizje i aktualizuje ich statusy na planszy.

`main.py`

Plik `main.py` pełni rolę punktu wejścia do aplikacji. Obsługuje interfejs użytkownika przy użyciu biblioteki curses. Umożliwia wybór trybu gry: "User vs Computer"lub "Computer Only". Implementuje główną pętlę gry (`game_loop`), która steruje interakcjami gracza, agentami i aktualizacjami stanu gry.

Diagram klas



Rysunek 1: Diagram klas programu

Zasada działania programu

Działanie systemu można opisać jako symulację gry wieloosobowej na siatce dwuwymiarowej. System dzieli się na trzy główne fazy:

- Inicjalizacja gry: Plansza i gracze są tworzeni na podstawie parametrów określonych w głównym menu. Pozycje początkowe graczy są losowane tak, aby nie kolidowały z innymi elementami planszy.
- Rozgrywka: W każdej turze gracze (zarówno użytkownik, jak i agenci) podejmują decyzje dotyczące swoich ruchów: Użytkownik steruje swoim graczem za pomocą klawiszy

kierunkowych (W, A, S, D). Agenci otrzymują sugestie ruchów z `Watchtower` i decydują, czy je zastosować. Po dokonaniu ruchów stan planszy jest aktualizowany, a gracze, którzy kolidują z przeszkodami lub innymi graczami, są oznaczani jako "rozbici".

- Zakończenie gry: Gra kończy się, gdy wszyscy gracze są rozbici lub na planszy pozostaje tylko jeden aktywny gracz. Na koniec prezentowana jest tabela wyników z kolejnością graczy, w jakiej odpadli z gry.

Funkcje programu

Główne komponenty

Inicjalizacja planszy: Kod odpowiada za stworzenie planszy (board) o zadanych wymiarach oraz inicjalizację pozycji graczy (`random_empty_position` w `Game`).

Pseudokod:

```
FOR each player IN game.players:
    WHILE True:
        random_position = generate_random_position()
        IF position_empty(random_position):
            player.position = random_position
            break
```

Decyzja o ruchu gracza:

Gracze sterowani przez komputer (agenci) korzystają z funkcji `decide_move` w `Agent` lub `AggressiveAgent`. Ruchy są sugerowane przez `Watchtower` na podstawie: Otaczających pól (`evaluate_move`). Bliskości przeciwników (`evaluate_aggressive_move`).

Odwołania do równania:

Funkcja oceniająca pole (`evaluate\_move`) przypisuje punkty na podstawie: score=$\sum$(wolne pola - kary za bliskość przeciwników) score=$\sum$(wolne pola - kary za bliskość przeciwników) W wersji agresywnej nagradzana jest także bliskość przeciwników oraz możliwość ich blokowania: aggressive_score = odległość do przeciwników + potencjał zablokowania przeciwnika

Aktualizacja stanu gry:

Plansza jest aktualizowana w każdej turze. Gracze przesuwają się na nowe pozycje, a status kolizji jest weryfikowany.

Pseudokod:

```
    FOR each player IN players:
        player.move()
        IF player.position OUTSIDE bounds OR COLLIDES with another
           player:
            player.crashed = True
```

Wyświetlanie wyników: Po zakończeniu gry wyświetlana jest tabela z wynikami.

Dzięki modularnej budowie systemu każda część kodu może być rozwijana lub modyfikowana niezależnie. Przykładowo, można łatwo dodać nowe typy agentów lub zmodyfikować logikę ruchów w Watchtower.

Testy

Przeprowadzono trzy rodzaje testów:

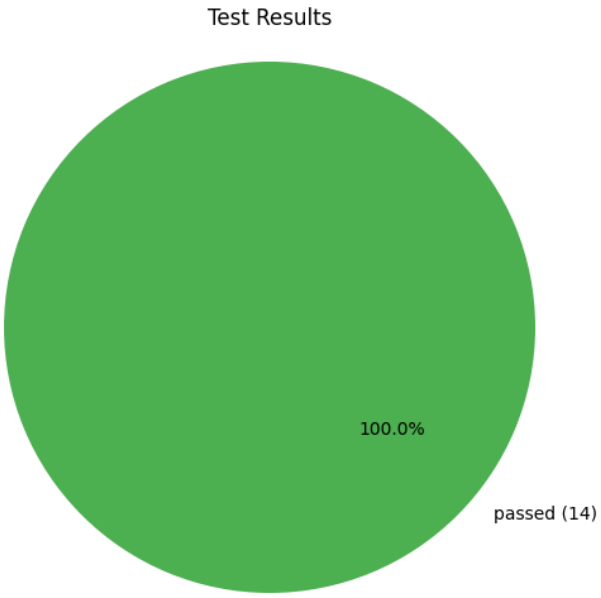
- testy jednostkowe,
- testy funkcjonalne,
- testy mutacyjne.

Poniżej przedstawiono szczegółowe wyniki i interpretacje każdego z tych rodzajów testów.

1. Testy jednostkowe:

   Testowane klasy:

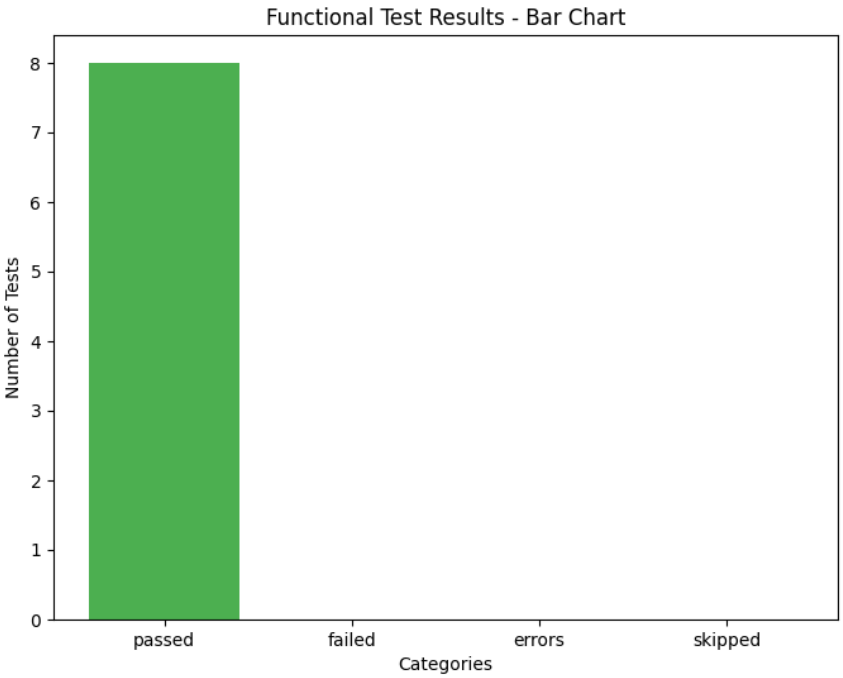
   - Player — podczas testów tworzony jest obiekt Player oraz sprawdzane jest, czy jego atrybutu zostały poprawnie zainicjalizowane. Następnie sprawdzane jest, czy obiekt poprawnie zmienia swoją pozycję w zależności od kierunku ruchu.
   - Agent — podczas testów tworzone są obiekty Game, Watchtower oraz Agent. Pierwszy test sprawdza, czy agent poprawnie przyjmuje sugestię wieży i zmienia kierunek na „Up". Drugi test sprawdza, czy agent wybiera losowy kierunek z możliwych opcji.
   - AggressiveAgent — inicjalizowane są obiekty Game, Watchtower oraz AggressiveAgent. Test sprawdza, czy agent przyjmuje agresywną sugestię wieży i zmienia kierunek na „DOWN".
   - Watchtower — inicjalizowane są obiekty Game, Agent oraz Watchtower. Pierwszy test sprawdza, czy wieża zwraca bezpieczny kierunek „UP", gdy pozycja na którą zmierza jest wolna. Drugi test sprawdza, czy wynik oceny ruchu jest typu *int*.
   - Game — inicjalizowany jest obiekt Game. Pierwszy test sprawdza poprawność inicjalizacji planszy gry. Drugi test sprawdza, czy metoda $random_empty_position$ zwraca losową pustą pozycję na planszy. Trzeci test sprawdza, czy po wykonaniu ruchu gracza, jego pozycja oraz pozycja na planszy są poprawnie aktualizowane. Ostatni test sprawdza, czy metoda $all_players_crashed$ poprawnie wykrywa, gdy wszyscy gracze ulegli zderzeniu.



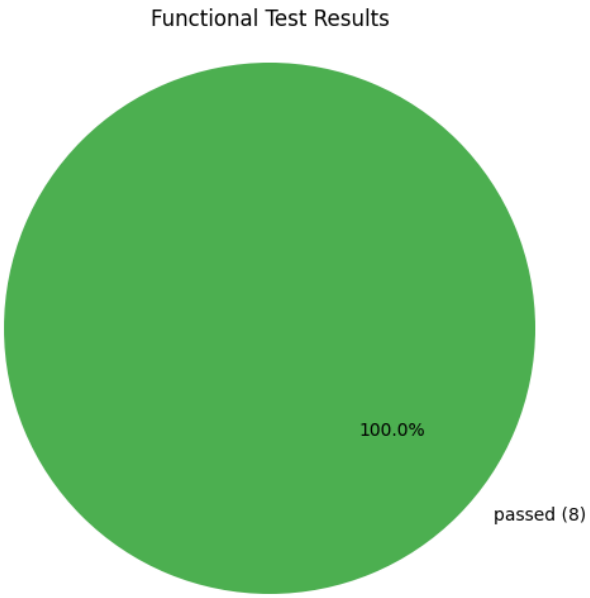
Rysunek 2: Wykres kołowy statusu testów jednostkowych

2. Testy funkcjonalne:

   - Inicjalizacja gry — sprawdzanie, czy gra poprawnie uruchamia planszę i jej stan początkowy (wszystkie komórki są puste).
   - Inicjalizacja gracza — sprawdza proces umieszczania graczy na planszy. Upewnia się, czy gracze są rozmieszczeni na różnych pozycjach.
   - Ruch gracza — sprawdza podstawowe mechanizmy ruchu gracza.
   - Podejmowanie decyzji przez agenta — sprawdza proces podejmowania decyzji przez agenta, upewniając się, że każdy agent podejmuje decyzję o kierunku ruchu i że kierunek jest prawidłowy.
   - Kolizja — test sprawdza zachowanie na granicach planszy, upewniając się, że gracz oznaczony jest jako *crashed* po próbie ruchu poza granice planszy.
   - Logowanie — test sprawdza funkcjonalność logowania: usuwa istniejący plik logu, tworzy nowy oraz zapisuje zdarzenia i sprawdza, czy plik logu został utworzony i zawiera odpowiednie zdarzenie.
   - Sugestia wieży — sprawdza, czy mechanizm sugerowania ruchów przez wieżę działa poprawnie i zawsze zwraca jeden z czterech możliwych kierunków.
   - Strategia agresywnego agenta — sprawdza, czy agresywny agent podejmuje decyzje zgodnie z agresywną strategią sugerowaną przez wieżę.



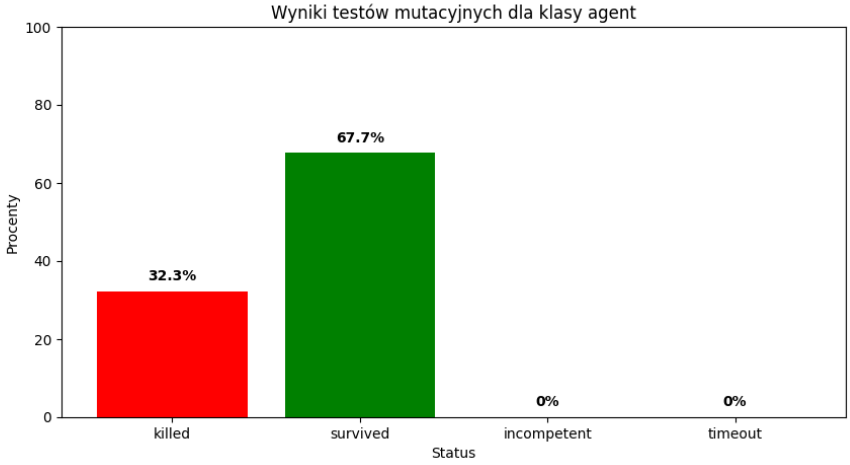
Rysunek 3: Wykres słupkowy statusu testów funkcjonalnych



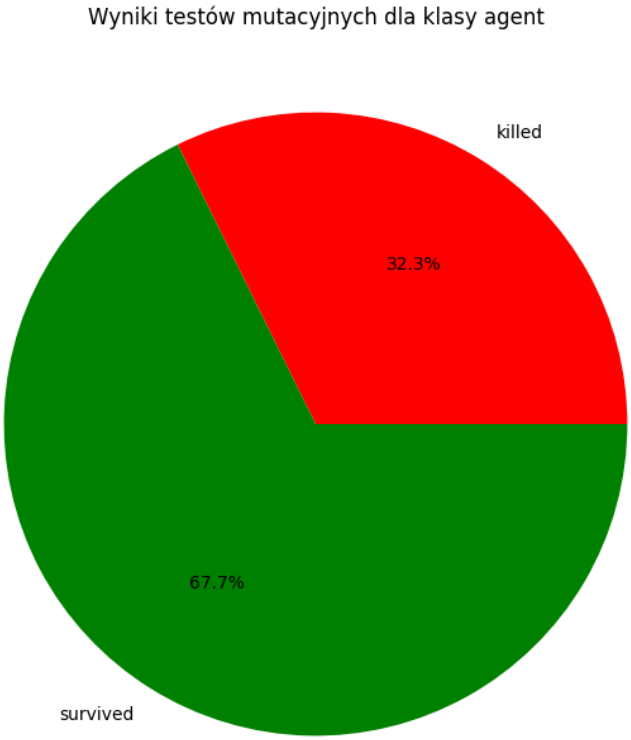
Rysunek 4: Wykres kołowy statusu testów funkcjonalnych

3. Testy mutacyjne: Testy mutacyjne dały następujące wyniki:

    - Test mutacyjny ukierunkowany na klasę Agent:

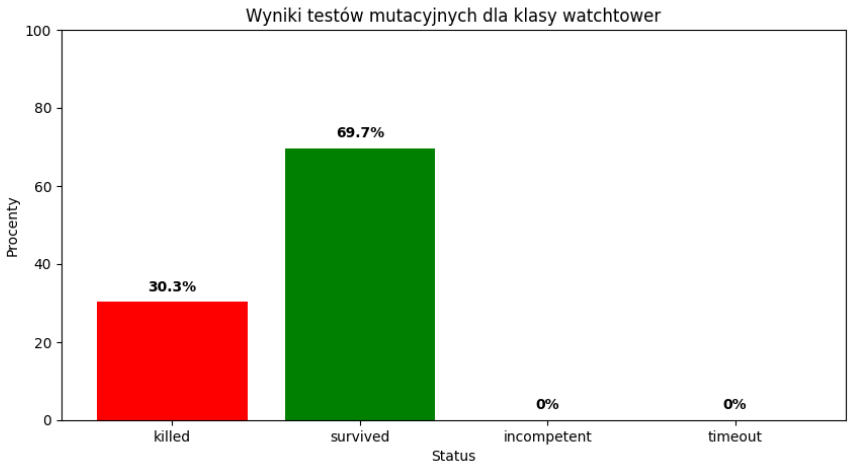


Rysunek 5: Wyniki testów mutacyjnych dla klasy Agent

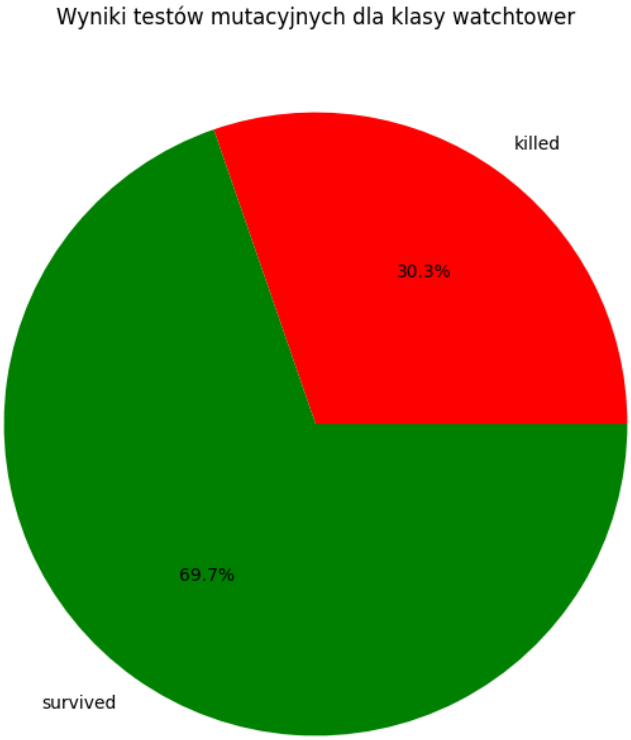


Rysunek 6: Wyniki testów mutacyjnych dla klasy Agent

- Test mutacyjny ukierunkowany na klasę Watchtower:

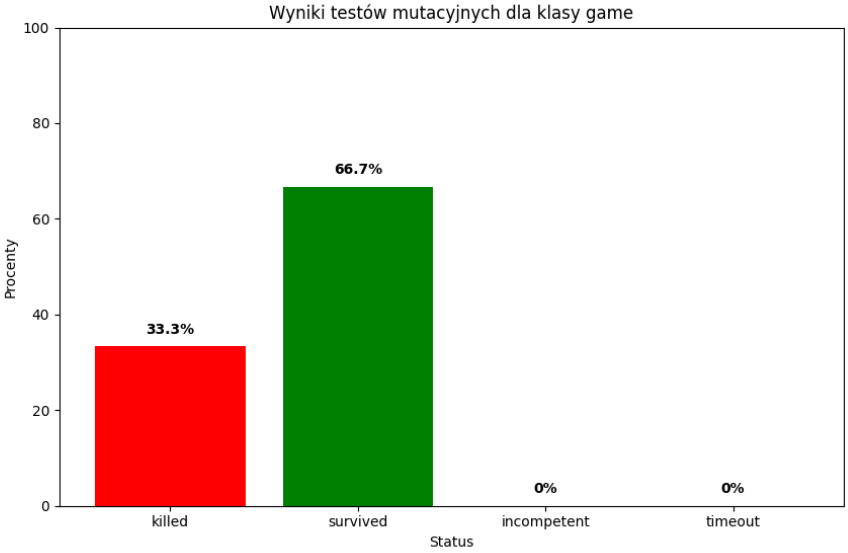


Rysunek 7: Wyniki testów mutacyjnych dla klasy Watchtower

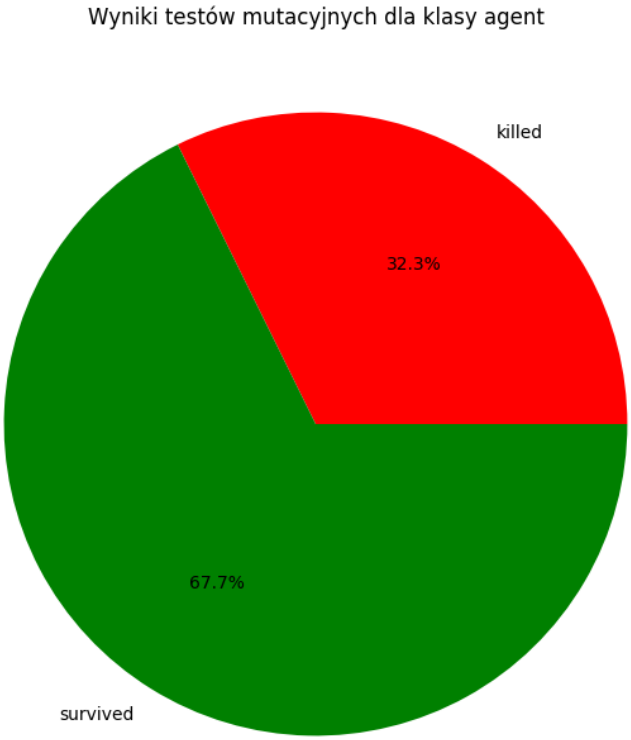


Rysunek 8: Wyniki testów mutacyjnych dla klasy Watchtower

- Test mutacyjny ukierunkowany na klasę Game:



Rysunek 9: Wyniki testów mutacyjnych dla klasy Game



Rysunek 10: Wyniki testów mutacyjnych dla klasy Game

Słabe wyniki testów mutacyjnych mogą wskazywać na:

- Niewystarczającą liczbę testów jednostkowych.
- Niską jakość testów jednostkowych.
- Słabe pokrycie kodu.
- Niewystarczającą różnorodność testów.

Złożoność oprogramowania

Przeprowadźmy kompletną, szczegółową analizę metodą Punktów Funkcyjnych (FPA) dla dostarczonego programu, uwzględniając wagi dla złożoności funkcji.

I. Identyfikacja i Klasyfikacja Funkcji:

Zaczynamy od zidentyfikowania External Inputs (EI), External Outputs (EO) i Logical Internal Files (LIF) oraz przypisania im odpowiedniej złożoności (prosta, średnia, złożona).

1. Zewnętrzne Wejścia (EI):

- Wybór trybu gry w menu głównym: Proste (jeden wybór z kilku opcji). Waga: 3

- Sterowanie gracza w grze (tryb gracz vs. komputer): Średnie (zmiana kierunku ruchu, wpływ na stan gry). Waga: 4 (Opcjonalne, występuje tylko w trybie gracz vs. komputer).

2. Zewnętrzne Wyjścia (EO):

- Wyświetlanie menu gry: Proste (statyczny tekst, kilka opcji). Waga: 4
- Wyświetlanie planszy gry: Średnie (dynamiczna aktualizacja, reprezentacja stanu gry, pozycja graczy). Waga: 5
- Wyświetlanie rankingu końcowego: Proste (formatowany tekst z wynikami). Waga: 4
- Logowanie do pliku ‘game_logs.txt‘: Proste (zapisywanie tekstu, informacje o zdarzeniach i decyzjach agentów). Waga: 4

3. Logiczne Wewnętrzne Typy Plików (LIF):

- Reprezentacja planszy gry (‘game.board‘): Złożona (dwuwymiarowa tablica, przechowywanie stanu każdego pola, detekcja kolizji). Waga: 15
- Dane graczy (obiekty klasy ‘Player‘, ‘Agent‘, ‘AggressiveAgent‘): Średnie (przechowywanie pozycji, kierunku, symbolu, stanu ‘crashed‘). Waga: 10

II. Obliczenie Unadjusted Function Points (UFP):
Teraz obliczamy UFP, sumując ważone wartości dla EI, EO i LIF.
Tryb "gracz vs. komputer"(EI = 3 + 4 = 7): UFP = (7 1) + (17 1) + (25 1) = 7 + 17 + 25 = 49
Tryb "tylko agenci"(EI = 3): UFP = (3 1) + (17 1) + (25 1) = 3 + 17 + 25 = 45
III. Określenie Wpływu Czynników Projektowych (TDI):
Przypisujemy wagi od 0 do 5 dla każdego z 14 czynników, oceniając ich wpływ na projekt. Poniższe wartości są przykładem i powinny być dostosowane do konkretnego kontekstu.

- Komunikacja (między komponentami programu): 3 (istotna, ale nie krytyczna)
- Rozproszone przetwarzanie danych: 2 (brak rozproszonego przetwarzania w ścisłym tego słowa znaczeniu)
- Wydajność (czas reakcji, płynność gry): 5 (krytyczna dla interaktywnej gry)
- Obciążenie (liczba graczy, intensywność aktualizacji): 4 (potencjalnie wysokie obciążenie w grze wieloosobowej)
- Częstość transakcji (aktualizacje stanu gry): 3 (umiarkowana, aktualizacje co pewien czas)
- Wprowadzanie danych online (interakcja z użytkownikiem): 2 (proste interakcje, wybór menu, sterowanie)
- Efektywność (wykorzystanie zasobów): 4 (ważna, szczególnie w kontekście potencjalnego działania w przeglądarce)

- Aktualizacja online (aktualizacje oprogramowania): 1 (brak wymogu aktualizacji online w trakcie gry)
- Złożoność przetwarzania (algorytmy AI, detekcja kolizji): 3 (umiarkowana złożoność algorytmów)
- Wielokrotne wykorzystanie (komponentów kodu): 1 (ograniczony potencjał ponownego użycia)
- Łatwość instalacji: 2 (stosunkowo prosta instalacja, uruchomienie skryptu)
- Łatwość utrzymania (czytelność kodu, dokumentacja): 3 (kod w miarę modularny, ale wymaga dokumentacji)
- Wielostanowiskowość (obsługa wielu graczy): 4 (istotny czynnik, gra wieloosobowa)
- Łatwość konfiguracji (parametry gry): 2 (ograniczone opcje konfiguracji)

Obliczenie TDI (Total Degree of Influence):
TDI = 3 + 2 + 5 + 4 + 3 + 2 + 4 + 1 + 3 + 1 + 2 + 3 + 4 + 2 = 39
IV. Obliczenie Adjusted Function Points (AFP):
AFP = UFP (0.01 TDI + 0.65)
Dla UFP = 49 (tryb z graczem): AFP = 49 (0.01 39 + 0.65) AFP = 49 (0.39 + 0.65) AFP = 49 1.04 AFP = 50.96 $\approx$ 51
Dla UFP = 45 (tryb tylko agenci): AFP = 45 (0.01 39 + 0.65) AFP = 45 (0.39 + 0.65) AFP = 45 1.04 AFP = 46.8 $\approx$ 47
Podsumowanie:
Ostateczne oszacowanie punktów funkcyjnych (AFP) wynosi około 51 dla trybu z graczem i około 47 dla trybu tylko agenci. Dokładna analiza z uwzględnieniem wag dla złożoności funkcji pozwoliła na uzyskanie bardziej precyzyjnego wyniku. Widać, że czynniki projektowe mają istotny wpływ na ostateczną wartość AFP. Pamiętaj, że kluczowe jest dokładne zrozumienie wymagań i specyfiki projektu przy szacowaniu złożoności funkcji i wpływu czynników projektowych.