WOJSKOWA AKADEMIA TECHNICZNA

**im. Jarosława Dąbrowskiego**

**WYDZIAŁ CYBERNETYKI**



PRACA DYPLOMOWA

STUDIA I°

Temat pracy: **pAKIET Programowy wspomagający konstruowanie symulacji wieloagentowych**

**Informatyka**

…………………………………………………………………………

(kierunek studiów)

**INŻYNIERIA SYSTEMÓW**

…………………………………………………………………………

(specjalność)

|  |  |
| --- | --- |
| Dyplomant: | Promotor: |
| **Paweł Karczewski** | **Dr inż. Dariusz Pierzchała** |

**Warszawa 2025**

**OŚWIADCZENIE**

*Wyrażam zgodę ~~/ nie wyrażam~~~~\*~~ ~~zgody~~   
na udostępnianie mojej pracy w czytelni Archiwum WAT.*

Dnia .......................... .................................

(podpis)

\**Niepotrzebne skreślić*

Spis treści

[Spis treści 3](#_Toc188569546)

[Wstęp 4](#_Toc188569547)

[Rozdział I. Wstęp teoretyczny 5](#_Toc188569548)

[I.1. Czym jest agent? 5](#_Toc188569549)

[I.2. Skomplikowana logika agentów 6](#_Toc188569550)

[I.3. Środowiska zadaniowe agentów 11](#_Toc188569551)

[I.4. Systemy wieloagentowe 13](#_Toc188569552)

[I.5. Uzasadnienie zastosowania symulacji w projektowaniu systemów wieloagentowych 14](#_Toc188569553)

[Rozdział II. Analiza istniejących rozwiązań 15](#_Toc188569554)

[II.1. NetLogo 15](#_Toc188569555)

[II.2. JADE (Java Agent Development Framework) 15](#_Toc188569556)

[II.3. AnyLogic 16](#_Toc188569557)

[II.4. Repast 16](#_Toc188569558)

[II.5. Mesa 17](#_Toc188569559)

[II.6. Podsumowanie 17](#_Toc188569560)

[Rozdział III. Architektura rozwiązania 18](#_Toc188569561)

[III.1. Wymagania funkcjonalne i niefunkcjonalne 19](#_Toc188569562)

[III.2. Najważniejsze elementy frameworka 20](#_Toc188569563)

[Rozdział IV. Przykłady symulacji i testy frameworka 32](#_Toc188569564)

[IV.1. Przykłady symulacji 32](#_Toc188569565)

[IV.2. Testy wydajnościowe na różnych wersjach .Net 42](#_Toc188569566)

[IV.3. Testy jednostkowe 43](#_Toc188569567)

[Podsumowanie 44](#_Toc188569568)

[Bibliografia 45](#_Toc188569569)

[Spis rysunków 45](#_Toc188569570)

[Załączniki 45](#_Toc188569571)

Wstęp

Współczesny rozwój technologii, takich jak sztuczna inteligencja, Internet Rzeczy (IoT) czy analiza dużych zbiorów danych (big data), oraz rosnąca złożoność procesów społecznych i ekonomicznych wymagają coraz bardziej zaawansowanych narzędzi do modelowania i analizy. Systemy wieloagentowe (ang. Multi-Agent Systems, MAS) stanowią jedno z kluczowych narzędzi w tej dziedzinie, umożliwiając symulowanie i badanie interakcji pomiędzy niezależnie działającymi jednostkami, zwanymi agentami. Niniejsza praca koncentruje się na stworzeniu uniwersalnego i wydajnego frameworka do budowy takich symulacji, który odpowiada na współczesne wyzwania badawcze oraz praktyczne.

Celem pracy jest zaprojektowanie i implementacja frameworka, który łączy w sobie przejrzystość, wydajność oraz uniwersalność, wyznaczając nowe standardy w tej dziedzinie. Wyniki pracy mogą znaleźć zastosowanie zarówno w badaniach naukowych, jak i w praktyce, np. w analizie procesów logistycznych, modelowaniu zjawisk społecznych czy zarządzaniu kryzysowym. Tworząc elastyczne i skalowalne narzędzie, praca przyczynia się do rozwoju metodologii w obszarze systemów wieloagentowych.

Struktura pracy została podzielona na kilka rozdziałów:

* Rozdział pierwszy opisuje agenta jako byt odbierający dane ze środowiska, analizujący je i działający na ich podstawie, wyróżniając różne typy agentów i środowisk. Wprowadza koncepcję systemów wieloagentowych oraz podkreśla znaczenie symulacji w ich projektowaniu i optymalizacji.
* Rozdział drugi porównuje istniejące rozwiązania. Podkreśla ich zalety i wady.
* Rozdział trzeci przedstawia architekturę rozwiązania, opisując kluczowe elementy frameworka do symulacji wieloagentowych. Skupia się na krokach czasowych, agentach, środowisku oraz mechanizmach takich jak snapshoty, losowość i weryfikacja symulacji. Podkreśla także elastyczność, intuicyjność i wydajność zaprojektowanego systemu.
* Rozdział czwarty opisuje pięć symulacji, takich jak ruch lotniska, kolonia mrówek czy gra w życie, pokazując zastosowanie frameworka w modelowaniu różnych zjawisk. Przedstawia szczegółowe zasady działania oraz wyniki każdej symulacji.

1. Wstęp teoretyczny
   1. Czym jest agent?

Agent to byt zanurzony w środowisku, zdolny do odbierania informacji o jego stanie za pomocą czujników, które dostarczają percepty – dane wejściowe agenta. Chronologiczny zapis takich perceptów nazywamy ciągiem percepcyjnym. Agent posiada mechanizm przetwarzania tych danych – logikę decyzyjną – której celem jest realizacja określonych celów. Wywieranie wpływu na środowisko odbywa się za pomocą aktuatorów, które generują akcje modyfikujące stan otoczenia.

Co istotne, agent nie tylko reaguje na percepty, ale także posiada zdolność refleksji. Oznacza to, że jest w stanie analizować swoje działania, oceniać ich skuteczność, przewidywać przyszłe stany środowiska oraz planować działania z wyprzedzeniem. Refleksja pozwala agentowi na proaktywność, dzięki czemu może samodzielnie podejmować inicjatywy oraz adaptować swoje strategie w odpowiedzi na zmieniające się warunki. W trakcie działania agent może również zdobywać doświadczenie, które wykorzystuje do doskonalenia swoich działań. Taki byt nazywamy agentem.

A diagram of a company

Description automatically generated

1. Schemat działania agenta
   1. Skomplikowana logika agentów

W celu uporządkowania sposobu działania agentów oraz ich zdolności przetwarzania informacji i podejmowania decyzji. Stuart Russell i Peter Norvig w swojej książce [1] wyróżniają pięć podstawowych kategorii agentów. Każda z tych kategorii różni się poziomem złożoności logiki oraz zakresem możliwości. Od agentów prostych, działających wyłącznie na podstawie bieżących perceptów, po zaawansowane systemy zdolne do uczenia się, przewidywania i podejmowania złożonych decyzji. Poniżej opisano każdą z tych kategorii.

* + 1. Agent odruchowy

Agent odruchowy jest najprostszym typem agenta, który reaguje wyłącznie na bieżące percepty. Jego logika działania jest oparta na zestawie reguł „jeśli – to” (ang. condition-action rules). Agent tego typu nie posiada pamięci ani zdolności przewidywania skutków swoich działań. Funkcjonuje w sposób reaktywny, bazując jedynie na aktualnych informacjach ze środowiska.

**Przykład:**

* Termostat, który włącza ogrzewanie, gdy temperatura spada poniżej określonego progu.
* Robot sprzątający, który omija przeszkody w czasie rzeczywistym.

**Zalety:**

* Prosta implementacja.
* Niskie wymagania obliczeniowe.

**Wady:**

* Brak zdolności adaptacji i przewidywania.
* Ograniczona skuteczność w złożonych środowiskach.
  + 1. Agent odruchowy oparty o modele

Agent odruchowy oparty o modele wprowadza zdolność do przechowywania wewnętrznej reprezentacji środowiska, nazywanej modelem. Dzięki temu agent może analizować stan otoczenia nie tylko na podstawie bieżących perceptów, ale również uwzględniając wcześniejsze doświadczenia. Model środowiska pozwala agentowi przewidywać skutki swoich działań oraz rozwiązywać bardziej skomplikowane problemy.

**Przykład:**

* Nawigacja GPS, która korzysta z mapy jako modelu środowiska, aby zaproponować optymalną trasę.
* Robot przemysłowy, który śledzi pozycję obiektów na linii produkcyjnej.

**Zalety:**

* Możliwość przewidywania skutków działań.
* Większa efektywność w dynamicznych środowiskach.

**Wady:**

* Wyższa złożoność obliczeniowa.
* Wymóg utrzymania dokładnego modelu środowiska.
  + 1. Agent ukierunkowany na cel

Agent ukierunkowany na cel podejmuje decyzje w celu osiągnięcia określonych celów. Oprócz modelu środowiska, agent ten posiada mechanizm pozwalający mu określić, czy dany stan jest pożądany (czyli czy cel został osiągnięty). Agent potrafi wybrać sekwencję działań, które przybliżą go do realizacji celu.

**Przykład:**

* System nawigacji autonomicznego pojazdu, który planuje trasę do wyznaczonego celu.
* Algorytm grający w gry, który stara się wygrać poprzez osiągnięcie zwycięskiego stanu.

**Zalety:**

* Skuteczność w osiąganiu zdefiniowanych celów.
* Możliwość zastosowania w złożonych środowiskach.

**Wady:**

* Wymaga dokładnej definicji celu.
* Może mieć trudności z adaptacją w przypadku zmiany warunków lub priorytetów.
  + 1. Agent z funkcją użyteczności

Agent z funkcją użyteczności jest rozszerzeniem agenta ukierunkowanego na cel, ale zamiast dążyć jedynie do osiągnięcia celu, optymalizuje swoje działania w taki sposób, aby maksymalizować funkcję użyteczności. Funkcja użyteczności pozwala agentowi oceniać różne stany środowiska pod kątem ich „jakości” lub „korzystności” i wybierać takie działania, które prowadzą do najbardziej pożądanych stanów.

**Przykład:**

* Systemy rekomendacyjne, które oceniają, jakie treści (np. filmy czy produkty) będą najbardziej atrakcyjne dla użytkownika.
* Zarządzanie zasobami w chmurze obliczeniowej, gdzie optymalizuje się koszty i wydajność.

**Zalety:**

* Zdolność do podejmowania bardziej złożonych decyzji.
* Możliwość priorytetyzacji różnych celów.

**Wady:**

* Trudność w zdefiniowaniu funkcji użyteczności dla złożonych systemów.
* Wyższe wymagania obliczeniowe.
  + 1. Agent uczący się

Agent uczący się to najbardziej zaawansowany typ agenta, który jest zdolny do modyfikowania swojej logiki działania na podstawie doświadczenia. Taki agent wykorzystuje różne techniki uczenia maszynowego, aby adaptować się do zmieniających się warunków, identyfikować wzorce w danych i doskonalić swoje decyzje. Wyróżnia się tu różne podejścia, takie jak uczenie nadzorowane, nienadzorowane czy ze wzmocnieniem.

**Przykład:**

* System rozpoznawania mowy, który poprawia swoje działanie na podstawie nowych danych.
* Autonomiczny pojazd, który uczy się na podstawie symulacji i rzeczywistych doświadczeń.

**Zalety:**

* Wysoka adaptacyjność.
* Możliwość działania w nieznanych lub dynamicznych środowiskach.

**Wady:**

* Duże wymagania obliczeniowe.
* Złożoność implementacji i potencjalne ryzyko błędów wynikających z procesu uczenia.
  1. Środowiska zadaniowe agentów

Agent jest zawsze osadzony w pewnym środowisku. W zależności od problematyki jest ono definiowane na różne sposoby. Stuart Russell i Peter Norvig w swojej książce[1] wyróżniają kilka takich cechy.

* + 1. Obserwowalność

Otoczenie, do którego ma dostęp agent może ograniczać się tylko do określonych aspektów bądź zakresu zależnego od położenia agenta.

Przykłady:

* Szachista ma wiedzę o wszystkich elementach gry.
* Autonomiczny robot sprzątający widzi tylko w zakresie swoich czujników. Dowie się, że drzwi są zamknięte dopiero kiedy się do nich zbliży.
* Pokerzysta nie widzi kart w dłoni przeciwnika.
  + 1. Mnogość agentów

System może zawierać zarówno wielu jak i tylko jednego agenta. Mogą one działać niezależnie, kooperować bądź rywalizować.

Przykłady:

* Autonomiczny robot sprzątający raczej będzie to robił sam.
* Mrówki współpracują w celu znalezienia drogi do pożywienia.
  + 1. Determinizm

Mając stan obecnego środowiska i agentów czasem możemy jednoznacznie przewidzieć przyszłość. Są jednak takie sytuacje, w których musimy zakładać stochastyczną naturę środowiska.

Przykłady:

* Przed autonomicznym samochodem może wyskoczyć jakiś niespodziewany uczestnik ruchu.
* Przeciwnik w grze, którego zachowanie jest determinowane przez logikę niezawierającą przypadkowości.
  + 1. Sekwencyjność

Agent podejmuje wiele działań w ciągu cyklu swojej pracy. Mówimy, że jest to zbiór pewnych sekwencji, prowadzących do osiągnięcia wybranego celu. Po drugiej stronie jest pojedyncza decyzja, która determinuje jaki będzie wynik pracy takiego agenta. Jest to epizod, który można powiedzieć trwa nieprzerwanie.

Przykłady:

* Sekwencyjny - Jadący samochód musi podczas manewrów aktualizować informacje czy nie przestać wykonywać swojego zadania i momentalnie zmienić kierunek ruchu.
* Epizodyczny - Program rozwiązujący sudoku.
  + 1. Dynamika

Agent podczas podejmowania decyzji musi się liczyć, że w tym czasie środowisko może się zmienić. Mówimy wtedy o dynamicznym charakterze. Po drugiej stronie są statyczne środowiska. Gdy jednak agent podejmuje decyzje na środowisku, który się nie zmienia, ale czas decyzji ma znaczenie w kontekście oceny jego wydajności to mówimy półdynamicznej naturze.

Przykłady:

* Dynamiczny - System naprowadzający pocisk lecący w kierunku ruchomego celu.
* Statyczny - Gra turowa bez ograniczeń czasowych.
* Półdynamiczny - System namierzania nieruchomego celu, nie zakładający zmian położenia ani zakłóceń. Czas ważny z punktu widzenia wybranej strategii wojennej.
  + 1. Ciągłość

Mając sytuację, że pomiędzy 2 punktami w czasie mamy nieskończenie wiele możliwych stanów to mówimy tu o ciągłym charakterze środowiska. W przeciwnym wypadku jest dyskretne.

* + 1. Rozpoznawalność

Jeżeli potrzeba na początku poznać środowisko zanim podejmie się decyzję możemy mówić, że jest nieznane z punktu widzenia agenta.

* 1. Systemy wieloagentowe

Systemy agentowe są podejściem do projektowania i implementacji oprogramowania, w którym centralnym elementem są agenci.

Systemy wieloagentowe (Multi-Agent Systems, MAS) to rozszerzenie tej koncepcji, gdzie wiele agentów współpracuje lub konkuruje, by osiągnąć zdefiniowane cele. Każdy agent w systemie posiada własne cele, zasoby i wiedzie, co czyni interakcje między nimi niezwykle złożonymi.

Systemy agentowe mogą być klasyfikowane według różnych kryteriów, takich jak:

1. Rodzaj interakcji:
   * Kooperacyjne – agenci współpracują, by osiągnąć wspólny cel.
   * Konkurencyjne – agenci rywalizują ze sobą, np. w grach czy symulacjach rynkowych.
2. Stopień autonomii:
   * Autonomiczne – agenci działają niezależnie, podejmując decyzje na podstawie własnych celów i stanu otoczenia.
   * Zcentralizowane – działania agentów są sterowane przez centralny mechanizm.
3. Rodzaj komunikacji:
   * Bezpośrednia – agenci komunikują się ze sobą za pomocą wiadomości.
   * Pośrednia – interakcje zachodzą poprzez zmiany w otoczeniu (np. feromony w systemach inspirowanych mrowiskami).
4. Zastosowania:
   * Symulacyjne – np. modelowanie zjawisk społecznych, rynkowych czy ekologicznych.
   * Operacyjne – np. zarządzanie ruchem drogowym czy systemy handlu elektronicznego.
   1. Uzasadnienie zastosowania symulacji w projektowaniu systemów wieloagentowych

Symulacje są kluczowym narzędziem w projektowaniu i analizie systemów wieloagentowych, gdyż umożliwiają:

1. Eksplorację złożonych interakcji:
   1. Wieloagentowe systemy cechuje wysoka złożoność wynikająca z interakcji między agentami oraz między agentami a środowiskiem. Symulacje pozwalają na badanie tych interakcji w kontrolowanych warunkach.
2. Testowanie scenariuszy:
   1. Symulacje umożliwiają sprawdzenie, jak system zachowa się w różnych scenariuszach, np. w sytuacjach kryzysowych lub przy zmianach parametrów otoczenia.
3. Optymalizację projektów:
   1. Dzięki symulacjom możliwe jest optymalizowanie działania systemu jeszcze przed jego wdrożeniem. Pozwala to zaoszczędzić czas i zasoby.
4. Weryfikację hipotez:
   1. Symulacje stanowią podstawę do weryfikacji hipotez dotyczących działania systemu. Dzięki nim można przewidzieć efekty wdrożenia określonych strategii lub zasad.

Podsumowując, zastosowanie symulacji w projektowaniu systemów wieloagentowych przyczynia się do lepszego zrozumienia ich działania, minimalizacji ryzyka oraz efektywnego wdrożenia rozwiązań w rzeczywistych zastosowaniach.

1. Analiza istniejących rozwiązań

Rozwój technologii i potrzeba modelowania złożonych systemów spowodowały, że symulacje systemów wieloagentowych (MAS – Multi-Agent Systems) zyskały na popularności. W literaturze oraz na rynku istnieje wiele frameworków i narzędzi wspierających budowanie takich symulacji. W celu zaprojektowania nowoczesnego i wydajnego narzędzia konieczna jest analiza istnieących rozwiązań, ich mocnych i słabych stron, a także określenie obszarów, które wymagają usprawnień.

* 1. NetLogo

NetLogo należy do języków bardziej rozpoznawalnych w środowiskach naukowych. Powstał w 1999 roku i opiera się głównie na prostocie i łatwością obsługi dla osób nie zaznajomionymi z żadnym konkretnym językiem oprogramowania.

**Zalety:**

* Intuicyjny interfejs użytkownika.
* Wbudowane wsparcie dla wizualizacji wyników.
* Obszerna biblioteka przykładowych modeli.

**Wady:**

* Ograniczona wydajność w przypadku dużej liczby agentów.
* Brak wsparcia dla programowania w bardziej zaawansowanych językach.
* Trudności w integracji z innymi systemami.
  1. JADE (Java Agent Development Framework)

JADE to framework napisany w Javie, zaprojektowany do budowy systemów wieloagentowych, szczególnie tych zgodnych z platformą FIPA (Foundation for Intelligent Physical Agents).

**Zalety:**

* Wszechstronność i zgodność z międzynarodowymi standardami.
* Silne wsparcie dla zaawansowanych symulacji w oparciu o logikę i planowanie.

**Wady:**

* Przestarzały
* Obecnie mała aktywność społeczności
  1. AnyLogic

AnyLogic to narzędzie komercyjne oferujące wsparcie dla różnych typów symulacji, w tym systemów agentowych, dyskretnego zdarzeniowego i dynamicznego.

**Zalety:**

* Wszechstronność: obsługuje wiele paradygmatów symulacji.
* Zaawansowane opcje wizualizacji i raportowania.
* Silne wsparcie techniczne i dokumentacja.

**Wady:**

* Wysoki koszt licencji.
* Zamknięta natura systemu ogranicza możliwość dostosowania.
  1. Repast

Repast (Recursive Porous Agent Simulation Toolkit) to zaawansowany, darmowy i otwartoźródłowy framework do budowy symulacji agentowych, rozwijany od ponad 20 lat.

**Zalety:**

* Wsparcie dla różnych języków programowania (Java, Python).
* Bogata dokumentacja.
* Darmowe i open-source.
* Integracja z GIS (Geographical Information Science).

**Wady:**

* Mniej intuicyjny interfejs.
* Trudniejsza nauka dla początkujących.
* Złożoność konfiguracji.
  1. Mesa

Mesa to framework oparty na Pythonie, zaprojektowany z myślą o prostocie i elastyczności.

**Zalety:**

* Łatwa integracja z narzędziami data science i machine learning.
* Prostota implementacji modeli.
* Aktywna społeczność i rosnąca popularność.
* Darmowe i open-source.

**Wady:**

* Ograniczona wydajność dla bardzo dużych symulacji.
  1. Podsumowanie

Każde z analizowanych rozwiązań posiada swoje mocne strony i ograniczenia. Wybór konkretnego frameworka zależy od wymagań projektu, takich jak skalowalność, wydajność, prostota użycia czy potrzeba wizualizacji wyników. Dla projektowanego frameworka kluczowe będzie zaadresowanie następujących problemów:

* Wydajność: Ograniczenie problemów z obsługą dużej liczby agentów.
* Elastyczność: Możliwość integracji z różnego rodzaju narzędziami oraz algorytmami.
* Prostota: Łatwość napisania kodu modularnego o niskim poziomie sprzężenia (ang. Loose coupling).
* Współczesne technologie: Wsparcie dla nowoczesnych paradygmatów programowania i infrastruktury chmurowej.

Analiza tych potrzeb stanowi podstawę do zaprojektowania rozwiązania, które będzie jednocześnie uniwersalne, wydajne i intuicyjne.

1. Architektura rozwiązania

Framework, który został stworzony w ramach tej pracy inżynierskiej, jest narzędziem umożliwiającym projektowanie i realizację symulacji wieloagentowych o charakterze dyskretnym, opartych o zdefiniowany krok czasowy. Jego podstawowym celem jest dostarczenie programistom środowiska do budowy symulacji, w których jednostki (zwane agentami) działają w wyodrębnionych krokach czasowych, realizując określone cele oraz wchodząc w interakcje z otoczeniem.

Framework został w całości napisany w języku C#, co umożliwiło skorzystanie z jego zaawansowanych funkcji programowania obiektowego. Taka architektura jest szczególnie bliska naturze agentów, którzy w systemie obiektowym mogą być reprezentowani jako niezależne obiekty z własnymi stanami i metodami. Programowanie obiektowe pozwala na intuicyjne modelowanie agentów jako autonomicznych jednostek działających według jasno zdefiniowanych reguł.

Rozwiązanie kładzie nacisk na trzy kluczowe aspekty: przejrzystość, uniwersalność oraz wydajność. Przejrzystość uzyskano dzięki użyciu zrozumiałej struktury kodu i wykorzystaniu nowoczesnych funkcji języka C#. Uniwersalność frameworka umożliwia zastosowanie go w szerokim spektrum scenariuszy symulacyjnych, od prostych systemów po bardziej złożone środowiska, gdzie występują złożone interakcje pomiędzy agentami. Z kolei wydajność osiągnięto poprzez zastosowanie mechanizmów optymalizacyjnych, takich jak efektywne zarządzanie pamięcią czy wsparcie dla operacji wielowątkowych.

Framework został zaprojektowany z myślą o elastyczności, aby użytkownicy mogli łatwo definiować agentów, ich zachowania oraz reguły rządzące całym środowiskiem. Dzięki temu narzędziu, proces tworzenia symulacji staje się bardziej intuicyjny, co pozwala skupić się na samych założeniach badawczych, zamiast na złożonościach implementacyjnych.

* 1. Wymagania funkcjonalne i niefunkcjonalne

Przed opracowaniem frameworka istotne było zdefiniowanie wymagań, które będą determinować jego funkcjonalność oraz sposób implementacji. Wymagania te zostały podzielone na dwie główne kategorie: funkcjonalne i niefunkcjonalne. Wymagania funkcjonalne opisują konkretne możliwości, jakie framework ma oferować użytkownikom, podczas gdy wymagania niefunkcjonalne określają cechy jakościowe i techniczne, które zapewniają wydajność, skalowalność i łatwość obsługi narzędzia.

**Wymagania funkcjonalne**

* Możliwość definiowania agentów z różnorodnymi stanami i zachowaniami.
* Obsługa środowiska symulacyjnego, w którym agenci mogą się poruszać i wchodzić w interakcje.
* Implementacja mechanizmu kroków czasowych, umożliwiającego wykonywanie symulacji w precyzyjnie zdefiniowanych interwałach czasowych.
* Obsługa różnorodnych scenariuszy symulacyjnych, w tym symulacji o charakterze deterministycznym i stochastycznym.
* Możliwość wizualizacji wyników symulacji w czasie rzeczywistym lub po jej zakończeniu.
* Elastyczne API umożliwiające łatwą integrację z innymi narzędziami i bibliotekami.

**Wymagania niefunkcjonalne**

* Wysoka wydajność umożliwiająca obsługę dużej liczby agentów jednocześnie.
* Skalowalność frameworka, umożliwiająca łatwe rozszerzanie jego funkcjonalności.
* Intuicyjność i prostota użytkowania, aby framework był dostępny zarówno dla zaawansowanych programistów, jak i mniej doświadczonych użytkowników.
* Wykorzystanie nowoczesnych technologii i wzorców projektowych zapewniających czytelność i łatwość utrzymania kodu.
* Wsparcie dla wielowątkowości i równoległego przetwarzania w celu optymalizacji czasu wykonywania symulacji.
  1. Najważniejsze elementy frameworka

Framework ma przed sobą jeszcze wiele zmian, ale są pewne elementy, które będą tworzyły fundamenty oprogramowania.

* + 1. Symulacja

Symulacja jest pewnym zbiorem etapów, które następują po sobie w każdej iteracji pętli. Oznacza to, że jest pewien porządek, który trzeba zdefiniować. Przed wykonaniem ciągu faz następuje przesunięcie w czasie.

1. Przetwarzanie symulacji

A diagram of a work flow

Description automatically generated

1. Główna pętla symulacji

A screen shot of a computer program

Description automatically generated

* + 1. Kontekst symulacji

Każda symulacja osadzona jest w pewnym środowisku oraz może zawierać nieujemną liczbę agentów.

Środowisko jest zawsze wymagane. Reprezentuje stan symulacji na zewnątrz agenta.

Agenci nie są wymagani. Posiadają swój własny stan.

A diagram of a company

Description automatically generated

1. Kompozycja kontekstu symulacji

Wszystko co ma dostęp do kontekstu symulacji może ją aktualizować przez dodawanie i usuwanie agentów oraz aktualizować przez interfejs udostępniony przez środowisko i agentów.

1. Interfejs kontekstu symulacji

A screen shot of a computer code

Description automatically generated

* + 1. Agent

Agenci to abstrakcyjny byt. Posiadają własną logikę i kompozycję. Rozróżnić ich można po identyfikatorze zbudowanym na podstawie GUID (Globally Unique Identifier).

A screenshot of a computer

Description automatically generated

1. Schemat implementacji agenta

Dla wygody została dodana klasa, implementująca domyślnie podstawowe elementy agenta SimpleSimulationAgent

1. Implementacja SimpleSimulationAgent

A screen shot of a computer program

Description automatically generated

* + 1. Faza

Przy każdej iteracji symulacji uruchamiane są w zdefiniowanej kolejności etapy. Etapy te będą używały kontekstu symulacji, aby zaktualizować stan środowiska i agentów. Proces jest podzielony na osobne etapy, aby ułatwić zarządzanie symulacją i mieć dowolność w poprawie wydajności.

A diagram of a stage

Description automatically generated

1. Schemat implementacji fazy
2. Przykładowa implementacja fazy

A screen shot of a computer program

Description automatically generated

* + 1. Czas

Chwila jest reprezentowana jako czas i tura od rozpoczęcia symulacji. Z racji tego, że nie każda symulacja opiera się o czas to nie musi być on aktualizowany za to tura zawsze. Aby zwiększyć bezpieczeństwo poprawności symulacji klasa została stworzona jak niemutowalna czyli zmiany nie można zrobić wewnątrz już utworzonej instancji.

1. Interfejs czasu w postaci tylko do odczytu

A screen shot of a computer code

Description automatically generated

1. Implementacja czasu

A computer screen shot of text

Description automatically generated

* + 1. Wydajność

W celu badania wydajności symulacji została dodana dodatkowa klasa, która pozwala na osobne mierzenie specyficznych kawałków kodu. Z racji tego, że niektóre fragmenty kodu to wnętrze pewnych pętli to w takich przypadkach liczy średni czas. Tworzenie obiektu przeliczeń dla bloku using zapewni domknięcie każdego liczenia z racji zastosowania wzorca Disposable.

1. Przykładowe zbieranie danych wydajnościowych

A screen shot of a computer program

Description automatically generated

Przykładowe statystyki:

* FPS: 28.06
* Move ball(avg): 263.43 ns
  + 1. Losowość

Podczas tworzenia symulacji chcemy zobaczyć, jak może przebiegać symulacja z różnymi warunkami początkowymi oraz móc odtworzyć specyficzny przypadek. W tym celu wprowadzony został mechanizm generowania liczb losowych. Podczas budowania symulacji można ustawić wartość ziarna.

W tym momencie interfejs posiada ograniczoną liczbę funkcji.

1. Interfejs do generowania liczb losowych

A screen shot of a computer code

Description automatically generated

Chcąc mieć dostęp do interfejsu wystarczy wstrzyknąć go za pomocą konstruktora.

1. Przykładowe zastosowanie generatora liczb losowych

A screen shot of a computer program

Description automatically generated

* + 1. Budowanie symulacji

Budowanie symulacji jest procesem skomplikowanym. Przydatne jest wydzielenie pewnych poziomów abstrakcji. Wiedząc, że symulacja składa się z faz można to w widoczny sposób pokazać w procesie jej budowania. Ułatwia to specjalnie do tego stworzona klasa budowniczego oparta o wzorzec fluent builder.

1. Fluent builder dla budowania symulacji

A screen shot of a computer program

Description automatically generated

1. Przykład budowy symulacji

A computer screen with text

Description automatically generated

* + 1. Snapshot

W celu prześledzenia przebiegu symulacji został stworzony system budowania snapshotów. Symulacja przy każdej rozpoczynającej się iteracji zapisuje stan w postaci pliku w formacie json, w którym są:

* Obecny czas symulacji
* Stan środowiska
* Stan wszystkich agentów.

Elementy, które mają zostać zapisane muszą zostać wymienione w konfiguracji środowiska oraz agentów.

Przy budowaniu symulacji należy dodać funkcję robienia snapshotów.

1. Przykładowa klasa tworząca snapshoty

A screen shot of a computer

Description automatically generated

1. Przykładowy snapshot w pliku json

A computer screen with white text

Description automatically generated

* + 1. Weryfikacja poprawności symulacji

Przy każdej symulacji mogą się zdarzyć błędy i to często takie, które niekoniecznie spowodują, że symulacja się zatrzyma. W celu zdiagnozowania takich sytuacji dodany został oddzielny mechanizm sprawdzania. Wystarczy dodać logikę i na końcu każdej symulacji następuje weryfikacja. Przy niepomyślnym przejściu weryfikacji symulacja się zatrzymuje.

1. Przykład dodawnia weryfikacji

A screen shot of a computer code

Description automatically generated

* + 1. Callback

Poza logiką samej symulacji czasem chcemy dodać jakieś dodatkowe rzeczy, jak informacje o zakończeniu iteracji bądź wyrenderowaniu obecnego stanu. W tym celu dodany jest mechanizm, gdzie możemy wykorzystać obecny kontekst symulacji. Zbiór wszystkich akcji uruchamia się na końcu symulacji.

* + 1. Wstrzykiwanie zależności jako podstawa frameworka

Podstawą projektowanego frameworka jest wykorzystanie mechanizmu wstrzykiwania zależności (Dependency Injection, DI), który pozwala na dynamiczne zarządzanie zależnościami pomiędzy różnymi komponentami systemu. Dzięki temu każde kluczowe elementy symulacji, takie jak klasy reprezentujące aktorów, środowisko czy fazy, mają możliwość korzystania z dowolnych, wcześniej zarejestrowanych zależności w kontenerze DI. Takie podejście znacznie zwiększa elastyczność, przejrzystość oraz łatwość rozszerzania frameworka.

Zastosowanie DI jako fundamentu frameworka przynosi szereg korzyści:

1. Modularność i elastyczność
   1. Komponenty frameworka nie są na sztywno związane. Tworzy to oprogramowanie łatwe do modyfikacji przez podmianę zależności albo dodanie logiki używając wzorca dekorator.
2. Testowalność
   1. Do każdego komponentu w bardzo prosty sposób można dodać testy jednostkowe. Wstrzykiwane zależności mogą być problemu zmienione przez dowolne klasy symulujące wybrane działanie kodu.
3. Czytelność
   1. Wyraźne zdefiniowanie zależności jest sposobem na uproszczenie kodu bez zmiany jego złożoności logicznej. Wydzielenie osobnych serwisów do ograniczonych skomplikowanych funkcjonalności ułatwia utrzymanie rozwiązania.
4. Reużywalność
   1. Każdy z wygenerowanych komponentów jak oraz dodatkowe serwisy mogą zostać wykorzystane w wielu miejscach tej samej bądź innej symulacji. Pozwala to również na dopisanie dodatkowych modułów.

Inicjalizacja symulacji przy użyciu tego frameworka ogranicza się jedynie do wykorzystania zdefiniowanej metody rozszerzającej na kontenerze zależności. Jedyne co trzeba podać w argumentach to assemblies, w których ma szukać komponentów jak środowisko, aktorzy czy fazy. Jeżeli podczas tworzenia oprogramowania programista zdecyduje dodać osobny serwis to jego trzeba dodać już ręcznie.

1. Rejestracja frameworka przez metodę rozszerzającą

A computer screen shot of a program code

Description automatically generated

1. Przykłady symulacji i testy frameworka

W ramach niniejszego rozdziału zostaną przedstawione przykłady symulacji opracowanych przy użyciu stworzonego frameworka oraz testy weryfikujące jego działanie. Celem tego rozdziału jest ukazanie praktycznych zastosowań narzędzia oraz ocena jego wydajności, skalowalności i uniwersalności.

* 1. Przykłady symulacji
     1. Symulacja ruchu lotniska

Pierwszym przykładem jest symulacja ruchu lotniska. Dla uproszczenia założono, że samolot po przydzieleniu mu pasa startowego okupuje go aż do odlotu.

**Celem symulacji:**

Sprawdzenie, czy zmniejszona ilość pasów startowych spowoduje powstanie korka w powietrzu.

**Środowisko:**

Lotnisko z ograniczoną liczbą pasów startowych.

**Agenci:**

* Samoloty oczekujące na przydzielenie pasa startowego oraz samoloty lądujące.
* Pasażerowie wychodzący z samolotu.

**Zasady:**

* Nowe samoloty przylatują z rozkładem wykładniczym i posiadają ilość pasażerów z rozkładem równomiernym.
* Kolejne samoloty muszą czekać na zwolnienie pasa.
* Każdy samolot po otrzymaniu pasa startowego korzysta z niego do momentu odlotu.
* Każdy samolot ląduje i odlatuje w takim samym czasie.
* Po wylądowaniu samolot czeka na pozbycie się pasażerów.
* Gdy jakiś samolot już odleciał to zwalnia pas startowy.

**Rezultaty:**

Symulacja pokazała, że przy:

* 12 pasach startowych praktycznie nie tworzą się korki w powietrzu.
* 10 pasach stale jest jakaś kolejka, ale ma ona ograniczoną długość
* 8 pasach kolejka się cały czas powiększa

A sheet of music with lines and dots

Description automatically generated

1. Wizualizacja lotniska przy 12 pasach startowych

A screenshot of a music score

Description automatically generated

1. Wizualizacja lotniska przy 10 pasach startowych

A screenshot of a music tab

Description automatically generated

1. Wizualizacja lotniska przy 8 pasach startowych
   * 1. Symulacja kolonii mrówek

W kolejnym przykładzie framework wykorzystano do symulacji kolonii mrówek. Symulacja ukazuje, jak mrówki tworzą ścieżki pomiędzy mrowiskiem a źródłami jedzenia, wykorzystując feromony. Możliwość modyfikacji parametrów utrzymywania się feromonów pozwala na badanie wpływu tych zmian na zachowanie kolonii.  
  
**Cel symulacji:**Zbadanie funkcjonowania mrówek w środowiskach o różnej utrzymywalności się feromonów.  
  
**Środowisko:**Przestrzeń dwuwymiarowa, po której poruszają się mrówki. Posiada mrowiska oraz źródła pożywienia.   
  
**Agenci:**  
Mrówki, które poszukują jedzenia. Nie posiadają wiedzy o drodze.  
 **Zasady:**

* Mrówki na początku są rozmieszczone w losowych miejscach
* Mrówki po znalezieniu jedzenia szukają mrowiska, a po znalezieniu mrowiska znowu szukają jedzenia.
* Mrówki zostawiają po sobie 2 typy feromonów wskazujące skąd idą (jedzenie, mrowisko).
* Im dalej jest mrówka od punktu, z którego startuje tym zostawia słabsze fereomony.
* Feromony się ulatniają w postępie geometrycznym.
* Wybór drogi przez mrówkę głównie polega na feromonach które czuje. Mimo to w losowych momentach używa losowego ruchu, a wskaźnik tej losowości zwany temperaturą jest ustawiony taki sam na czas całej symulacji.

**Rezultaty:**

* Przy dostatecznie dużym czynniku utrzymywania się feromonów mrówki znajdują ścieżkę pomiędzy pożywieniem a mrowiskiem.
* W przypadku zbyt szybkiego ulatniania się feromonów nie daje się ukształtować, żadnej ścieżki.

**A group of colored dots

Description automatically generated with medium confidence**

1. Wizualizacja mrowiska przy utrzymujących się feromonach

**A group of dots with different colored circles

Description automatically generated**

1. Wizualizacja mrowiska przy szybko ulatniających się feromonach
   * 1. Symulacja efektu motyla

Symulacja efektu motyla polega na wizualizacji ruchu piłek odbijających się wewnątrz okręgu. Każda piłka porusza się zgodnie z ustalonymi zasadami, a ich trajektorie nie oddziałują ze sobą.

**Cel symulacji:**Ukazanie, jak niewielkie różnice w punktach początkowych mogą prowadzić do znacznie odmiennych wyników w końcowych efektach symulacji.  
  
**Środowisko:**Zamknięty okrąg, w którym piłki odbijają się od jego ścian.  
  
**Agenci:**  
Piłki, które spadają i odbijają się pod ścian.  
 **Zasady:**

* Początkowo wszystkie piłki spadają ze środka okręgu. Są nieznacznie przesunięte.
* Podczas odbicia zmieniają swój kierunek poruszania się na lustrzany względem punktu odbicia.
* Na piłki działa grawitacja przyśpieszająca ich ruch w dół.

**Rezultaty:**

* Początkowo nie widać, aby piłki mocno zmieniały swój punkt położenia.
* W pewnym momencie widać, że piłki zaczynają tworzyć łańcuch ze względu na to jak bardzo były wysunięte względem środka okręgu.
* W pewnym momencie widać wpływ błędów numerycznych na położenie piłek.
* Ostatecznie prowadzi to do pełnego chaosu położenia.

|  |  |
| --- | --- |
| Przy początku:  A black circle with red dot  Description automatically generated | Po rozczepieniu:  A rainbow colored line in a circle  Description automatically generated |
| Widoczny efekt błędów numerycznych:  A rainbow colored lines in a circle  Description automatically generated | Ostateczny rezultat:  A colorful circle with black background  Description automatically generated |

1. Wizualizacja efektu motyla
   * 1. Symulacja gry w życie

Symulacja „Gra w życie” Johna Conwaya jest klasycznym przykładem automatu komórkowego. Symulacja ta pokazuje, jak proste reguły mogą prowadzić do powstawania złożonych struktur i zachowań.

**Cel symulacji:**Pokazanie jak proste reguły mogą prowadzić do złożonych struktur i zachowań.  
  
**Środowisko:**Dwuwymiarowa przestrzeń komórek.  
  
**Agenci:**  
Komórka, która w zależności od warunków może być albo „żywa” albo „martwa”.  
 **Zasady:**

* Początkowo każda komórka ma stan albo „żywy”, albo „martwy”.
* „Żywa” komórka z mniej niż dwoma lub więcej niż trzema sąsiadami „umiera” (samotność lub przeludnienie).
* „Żywa” komórka z dwoma lub trzema sąsiadami pozostaje „żywa”.
* „Martwa” komórka z dokładnie trzema sąsiadami staje się „żywa” (narodziny).

**Rezultaty:**

* Tworzą się struktury statyczne, które z czasem nie zmieniają swojego kształtu.
* Powstają oscylatory, które cyklicznie powracają do swoich poprzednich ułożeń.
* Generują się struktury przemieszczające po planszy.

A black and white background

Description automatically generated

1. Wizualizacja symulacji „Gra w życie”
   * 1. Ofiary i drapieżnicy

W tej symulacji przedstawiono zależność między populacjami drapieżników i ich ofiar.

**Cel symulacji:**Analiza dynamiki obu grup, z uwzględnieniem czynników takich jak głód drapieżników i szybkość rozmnażania ofiar.  
  
**Środowisko:**Dwuwymiarowa plansza.  
  
**Agenci:**

* Drapieżnicy polujący na ofiary.
* Ofiary rozmnażające się w określonej szybkości.

**Zasady:**

* Na początku losowo ustawiane są miejsca, w których przebywają osobniki.
* Drapieżniki mają określony czas. przed którym muszą coś zjeść aby przeżyć.
* Po zjedzeniu ofiary drapieżniki regenerują czas oraz tworzą nowego potomka.
* Ofiary, gdy są same zwiększają swój współczynnik narodzin, który po przekroczeniu krytycznej wartości pozawala im stworzyć potomka.
* Wszystkie osobniki poruszają się w losowym kierunku.

**Rezultaty:**

* Dla optymalnych parametrów można wydzielić 2 fazy symulacji:
  + Zwiększanie populacji ofiar, gdzie cały możliwy teren jest zajmowany przez ofiary a nieliczne wysepki przez drapieżników.
  + Pościgu drapieżników za ofiarami.
* Przy zbyt małym współczynniku rozmnażalności ofiar łatwiej o to, że zbyt mało pożywienia mają drapieżnicy i w ten sposób giną. Kończy się to tym, że drapieżnicy kumulują się przy jakimś skupisku ofiar, a ofiary spoza tego skupiska żyją i po śmierci drapieżników odbudowują swoją populację.
* Podobny efekt daje zbyt krótki czas wytrzymałości na brak pożywienia o drapieżników.
* Zbyt krótki czas wytrzymałości drapieżników prowadzi do zjedzenia wszystkich ofiar, a następnie do śmierci drapieżników, czyli śmierci wszelkiego życia.

|  |  |
| --- | --- |
| Faza rozwoju ofiar:  **A green and blue dots  Description automatically generated** | Faza zrównoważonego pościgu:  **A close-up of a black background  Description automatically generated** |
| Widoczna apokalipsa drapieżników:  A green and blue gradient on a black background  Description automatically generated | Odrodzenie ofiar:  A green and black background  Description automatically generated |
| Zbyt długie życie drapieżników początek:  A blue and black speckled background  Description automatically generated | Po zjedzeniu wszystkich ofiar:  A black square with white lines  Description automatically generated |

1. Wizualizacja symulacji „Ofiary i drapieżnicy”

* 1. Testy wydajnościowe na różnych wersjach .Net

Język C# został wybrany jako główny język programowania dla stworzonego frameworka ze względu na jego przejrzystość, uniwersalność oraz dynamiczny rozwój, który w ostatnich latach skupia się również na optymalizacji wydajności. Testy przeprowadzone w ramach projektu wykazały znaczące różnice w wydajności aplikacji w zależności od użytej wersji platformy .NET. Program wykorzystujący stworzony framework, uruchomiony na wersji .NET 9.0, osiągał niemal trzykrotnie lepsze wyniki w porównaniu do tej samej aplikacji uruchomionej na .NET 7.0. Wyniki te podkreślają, że .Net, jako platforma, stale podąża za wymaganiami współczesnych systemów informatycznych, oferując coraz lepsze możliwości optymalizacji i skalowalności.

A screenshot of a computer program

Description automatically generated

1. Wynik testów wydajnościowych dla różnych wersji .Net
   1. Testy jednostkowe

Jednym z kluczowych elementów tworzenia wysokiej jakości oprogramowania jest możliwość łatwego pisania testów jednostkowych. Dobrze zaprojektowany kod powinien być zrozumiały, modularny oraz wolny od nadmiernych zależności, co przekłada się na prostotę weryfikacji jego poprawności przy użyciu testów. Dzięki testom jednostkowym programiści mogą szybko sprawdzić działanie poszczególnych modułów, zidentyfikować błędy i upewnić się, że zmiany wprowadzone w kodzie nie wpłynęły negatywnie na istniejącą funkcjonalność.

1. Przykładowe testy jednostkowe frameworka

A computer screen shot of a program

Description automatically generated

Podsumowanie

W ramach niniejszej pracy inżynierskiej udało mi się stworzyć framework do symulacji wieloagentowych, który spełnia założenia przejrzystości, uniwersalności oraz wydajności. Jestem zadowolony z osiągniętych rezultatów, a powstałe oprogramowanie jest narzędziem, którego sam chętnie używałbym w przyszłości do realizacji podobnych projektów.

W początkowej fazie projektu miałem zupełnie inną koncepcję tworzenia oprogramowania. Dopiero podczas implementacji pierwszych symulacji oraz przeprowadzania testów wydajnościowych, zrewidowałem swoje podejście. Kluczowym aspektem okazało się podzielenie symulacji na fazy, które muszą być realizowane w każdym procesowanym momencie czasowym. Takie podejście uprościło tworzenie nowych symulacji, a jednocześnie ułatwiło zarządzanie kodem. Dodatkowo, zmiana podejścia pozwoliła na znaczące wpływy na wydajność frameworka, co jest jego istotnym atutem.

Pomimo osiągniętych rezultatów, framework posiada obecnie pewne ograniczenia. Jednym z nich jest wymóg stałego kroku czasowego w symulacji. Znaczącym usprawnieniem byłoby wprowadzenie możliwości uruchamiania iteracji symulacji w konkretnych chwilach czasowych za pomocą harmonogramu. Dodatkowo, wartość projektu mogłaby zostać podniesiona poprzez wprowadzenie koncepcji eventów, które mogłyby być zintegrowane z harmonogramem. Po wykonaniu akcji przypisanej do danego eventu, system mógłby automatycznie realizować wszystkie fazy iteracji.

W przyszłości planuję również wprowadzić zmiany w kodzie w celu zwiększenia jego czytelności, co wpłynie na łatwość dalszego rozwoju frameworka. Kolejnym kierunkiem rozwoju jest dodanie obsługi GPU do obliczeń, co pozwoli na jeszcze wydajniejsze przetwarzanie danych. Planuję także zaimplementować wbudowany moduł sieci neuronowej, co otworzy nowe możliwości zastosowań frameworka w bardziej zaawansowanych scenariuszach.

Stworzony framework znajduje zastosowanie nie tylko w obszarze symulacji. Może on być wykorzystany do tworzenia aplikacji orkiestrujących złożone środowiska wielokomponentowe. Elastyczność architektury pozwala na adaptację rozwiązania do różnorodnych scenariuszy, co czyni framework wszechstronnym narzędziem dla programistów.

Podsumowując, praca ta stanowi solidną podstawę do dalszego rozwoju i eksploracji możliwości zastosowań frameworka w różnych dziedzinach programowania.

Bibliografia

1. Russell S., Norvig P.: Artificial Intelligence: A Modern Approach, tom 1 i 2, wydanie IV
2. Perros H. Domputer Simulations Techniques: The definitive introduction!
3. https://learn.microsoft.com/pl-pl/dotnet/csharp/language-reference/ [22.01.2025]
4. https://devblogs.microsoft.com/dotnet/ [22.01.2025]

Spis rysunków

[Rys. 1. Schemat działania agenta 5](#_Toc188489066)

[Rys. 2. Kompozycja kontekstu symulacji 22](#_Toc188489067)

[Rys. 3. Schemat implementacji agenta 23](#_Toc188489068)

[Rys. 4. Schemat implementacji fazy 24](#_Toc188489069)

[Rys. 5. Wizualizacja lotniska przy 12 pasach startowych 32](#_Toc188489070)

[Rys. 6. Wizualizacja lotniska przy 10 pasach startowych 32](#_Toc188489071)

[Rys. 7. Wizualizacja lotniska przy 8 pasach startowych 32](#_Toc188489072)

[Rys. 8. Wizualizacja mrowiska przy utrzymujących się feromonach 34](#_Toc188489073)

[Rys. 9. Wizualizacja mrowiska przy szybko ulatniających się feromonach 34](#_Toc188489074)

[Rys. 10. Wizualizacja efektu motyla 36](#_Toc188489075)

[Rys. 11. Wizualizacja symulacji „Gra w życie” 38](#_Toc188489076)

[Rys. 12. Wizualizacja symulacji „Ofiary i drapieżnicy” 40](#_Toc188489077)

[Rys. 13. Wynik testów wydajnościowych dla różnych wersji .Net 41](#_Toc188489078)

Załączniki

1. Płyta CD/DVD zawierająca:
   1. Kody źródłowe oprogramowania