

POLITECHNIKA WROCŁAWSKA  
WYDZIAŁ ELEKTRONIKI

---

KIERUNEK: Informatyka (INF)

SPECJALNOŚĆ: Systemy i Sieci Komputerowe (ISK)

PRACA DYPLOMOWA  
MAGISTERSKA

Analiza porównawcza wybranych metod  
sztucznej inteligencji w komputerowych grach  
strategicznych

Comparative analysis of artificial intelligence  
methods applied to strategy games

AUTOR:

Kamil Markuszewski

PROWADZĄCY PRACĘ:

Dr inż. Leszek Koszałka

OCENA PRACY:

---

WROCŁAW 2013

# Spis Treści

---

1. Wprowadzenie.....	5
1.1. Wstęp .....	5
1.2. Cel pracy .....	5
1.3. Zakres pracy.....	5
1.4. Użyte technologie .....	5
1.4.1. Unity 3D.....	6
2. Sztuczna inteligencja w grach komputerowych.....	7
2.1. Pojęcia.....	7
2.2. Zadanie sztucznej inteligencji w grach komputerowych .....	9
2.3. Historia sztucznej inteligencji w grach komputerowych .....	9
2.4. Automaty stanów skończonych .....	10
2.5. Logika rozmyta .....	10
2.6. Inteligencja stadna .....	11
2.7. Algorytmy grafowe i heurystyczne przeszukiwanie drogi.....	11
2.8. Drzewa decyzyjne.....	12
2.9. Drzewa gier.....	12
2.10. Sztuczne sieci neuronowe.....	13
3. Projekt komputerowej gry strategicznej .....	14
3.1. Plansza .....	14
3.2. Zasady rozgrywki .....	16
3.3. Warunki zwycięstwa.....	16
3.4. Funkcjonalności .....	16
4. Implementacja komputerowej gry strategicznej .....	17
4.1. Graficzny interfejs użytkownika .....	17
4.2. Struktura programu .....	19

5. Zastosowane metody sztucznej inteligencji.....	22
5.1. Reprezentacja planszy .....	22
5.2 Metody zapisu grafu .....	25
5.1.1. Macierz sąsiedztwa .....	25
5.1.2. Macierz incydencji .....	26
5.1.3. Lista incydencji .....	26
5.1.4. Lista sąsiedztwa .....	27
5.3. Algorytmy wyszukiwania drogi .....	28
5.3.1. Algorytm Dijkstry .....	28
5.3.2. Algorytm Forda-Bellmana .....	29
5.3.3. Algorytm A* .....	30
5.3.4. Porównanie algorytmów .....	33
5.4. Modyfikacje algorytmu A* .....	39
5.4.1. Proponowane modyfikacje.....	40
5.5. Logika rozmyta i automaty stanów skończonych .....	42
6. Podsumowanie .....	46
6.1. Efekty implementacji.....	46
6.2. Wady i zalety aplikacji .....	48
6.3. Możliwość rozwoju .....	48
6.4. Podsumowanie użytych metod sztucznej inteligencji .....	49
6.5. Zakończenie .....	50
7. Bibliografia .....	51
8. Załącznik: Instrukcja edycji kodu źródłowego gry.....	51

# Spis rysunków

---

Rys. 1.1 Unity Editor dostarczany wraz z silnikiem Unity 3D.	6
Rys. 2.1 Przykładowe drzewo decyzyjne dalszej strategii gry.	12
Rys. 3.1 Plansza gry.	14
Rys. 3.2 Legenda obiektów na planszy.	15
Rys. 3.3 Plansza gry zawierająca specyficzne przeszkody.	15
Rys. 4.1 Menu główne gry.	17
Rys. 4.2 Graficzny interfejs użytkownika dla rozgrywki.	17
Rys. 4.3 Informacje o agencji wyświetlane w obszarze menu rozgrywki.	18
Rys. 4.4 Menu główne rozgrywki.	18
Rys. 4.5 Wszystkie menu rozgrywki.	18
Rys. 4.6 Mini mapa.	18
Rys. 4.7 Struktura plików.	19
Rys. 4.8 Najważniejsze obiekty gry w środowisku Unity3d.	20
Rys. 4.9 Struktura projektu.	21
Rys. 5.1 Przykładowa plansza do gry.	22
Rys. 5.2 Proces przekształcania planszy w graf.	23
Rys. 5.3 Fragment grafu z rysunku 5.2. Posłuży za dalszy przykład.	24
Rys. 5.4 Macierz sąsiedztwa dla grafu z rysunku 5.3.	25
Rys. 5.5 Macierz incydencji dla grafu z rysunku 5.3.	26
Rys. 5.6 Lista incydencji dla grafu z rysunku 5.3	26
Rys. 5.7 Lista sąsiedztwa dla grafu z rysunku 5.3	27
Rys. 5.8 Algorytm Dijkstry w pseudokodzie.	28
Rys. 5.9 Algorytm Forda-Bellmana w pseudokodzie.	29
Rys. 5.10 Diagram aktywności opisujący algorytm A*.	31
Rys. 5.11 Wynik działania algorytmu A* wraz z analizowanymi polami.	32
Rys. 5.12 Wynik działania algorytmu A* wraz z analizowanymi polami.	32
Rys. 5.13 Tabela [ms] czas tworzenia grafu w zależności od liczby wierzchołków.	33
Rys. 5.14 Wykres czasu [ms] tworzenia grafu w zależności od liczby wierzchołków.	34
Rys. 5.15 Wykres czas [ms] tworzenia grafu w zależności od liczby wierzchołków.	35
Rys. 5.16 Tabela czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.	36
Rys. 5.17 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.	36
Rys. 5.18 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.	37
Rys. 5.19 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.	37
Rys. 5.20 Tabela czas [ms] wyszukiwania ścieżki w zależności od odległości na planszy.	37
Rys. 5.21 Wykres czas [ms] wyszukiwania ścieżki w zależności od odległości na planszy.	38
Rys. 5.22 Wykres czas [ms] wyszukiwania ścieżki w zależności od odległości dla algorytmu A*.	38
Rys. 5.23 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.	40
Rys. 5.24 Wynik działania modyfikacji algorytmu A*.	40
Rys. 5.25 Wykres czas [ms] wyszukiwania ścieżki w zależności od odległości dla modyfikacji algorytmu A*.	41
Rys. 5.26 Podział celów sztucznej inteligencji.	42
Rys. 5.27 Automat stanów agenta odpowiedzialny za zbieranie zasobu.	43
Rys. 5.28 Automat stanów gracza odpowiedzialny strategię gry.	43
Rys. 5.29 Opis stanu zasobu w magazynie z wykorzystaniem standardowej logiki.	44
Rys. 5.30 Wykresy ukazujące różnice między logiką rozmytą a standardową logiką.	44
Rys. 5.31 Klasa wspomagająca użycie logiki rozmytej.	45
Rys. 5.32 Przykład wykorzystania klasy FuzzyLogic.	45
Rys. 6.1 Rozgrywka.	46
Rys. 6.2 Rozgrywka.	46
Rys. 6.3 Wyniki ankiety.	47

# 1. Wprowadzenie

## 1.1. Wstęp

Sztuczna inteligencja to nauka zajmująca się między innymi logiką rozmytą, algorytmami ewolucyjnymi, sztucznymi sieciami neuronowymi oraz symulowaniem zachowania istot inteligentnych. Nauka ta traktuje zarówno o próbie stworzenia algorytmów, które potrafią zastąpić człowieka, a także o symulowaniu niektórych zachowań występujących w naturze. Wykorzystywana jest w bardzo wielu naukach potrzebujących automatyzacji lub rozwiązania skomplikowanych problemów.

Gry komputerowe są dziedziną, która często wykorzystuje sztuczną inteligencję. Dla różnych rodzajów gier komputerowych wykorzystywane są różne dziedziny tej nauki. W dzisiejszych czasach temat gier komputerowych to nie prosta rozrywka tworzona wyłącznie przez domowych amatorów. Tematyką tą zajmują się firmy zatrudniające tysiące pracowników, a tworzone aplikacje zajmują gigabajty pamięci i są swego rodzaju rozbudowanymi światami symulującymi dość dobrze rzeczywistość.

Niniejsza praca skupia się na grach strategicznych gdzie nie wykorzystuje się sieci neuronowych, lecz jedynie algorytmy wyszukujące drogę, logikę rozmytą oraz inteligencję stadną. Są to znacznie prostsze działy tej nauki jednak nie mniej ciekawe i użyteczne.

## 1.2. Cel pracy

Celem pracy jest implementacja komputerowej gry strategicznej wykorzystującej wybrane metody sztucznej inteligencji. Jest to typowo inżynierski aspekt pracy, który jest dość obszerny.

Aspektem badawczym pracy jest wybór, wieloaspektowa analiza oraz porównanie metod sztucznej inteligencji z użyciem autorskich wskaźników jakości. Metody te będą musiały być zaimplementowane, ocenione, a następnie niektóre z nich zostaną użyte w implementowanej grze. Zanim wybrane konkretne metody zostaną poddane analizie wykonany będzie przegląd literaturowy pod kątem praktycznego wykorzystania rozwiązań w grach.

Dodatkowo planowane jest przeprowadzenie oceny algorytmów w praktycznym wykorzystaniu przez kilku niezależnych użytkowników ukończonej aplikacji.

## 1.3. Zakres pracy

W celu stworzenia tej pracy przeprowadzono przegląd mechanizmów wykorzystywanych w komputerowych grach strategicznych. Zostały one omówione, lecz nie wszystkie z nich użyto czy zbadano. Został stworzony projekt logiczny gry komputerowej z naciskiem na stosunkowo proste zasady rozgrywki. Następnie projekt gry został zaimplementowany przy użyciu kilku mechanizmów sztucznej inteligencji, niektóre z nich zostały zaimplementowane w kilku wariantach. W trakcie implementacji przeprowadzono badanie wybranych algorytmów, polegające na testowaniu poprawności i wydajności.

## 1.4. Użyte technologie

Głównym aspektem inżynierskim pracy jest implementacja gry strategicznej. Do tego celu wybrane zostało narzędzie **Unity 3D**. Jest zarówno narzędziem graficznym specjalnie przygotowanym do tworzenia gier jak i środowiskiem programistycznym pozwalającym na używanie takich języków jak **JavaScript** oraz **C#**. Dzięki temu możliwe będzie wykorzystanie zalet bibliotek dostarczanych przez **.NET 3.5**.

Głównym aspektem badawczym pracy jest porównanie metod sztucznej inteligencji. Do tego celu zostanie wykorzystany głównie język **C#** wraz z **.NET 3.5**. Zrealizowanie wszystkich algorytmów w tej samej technologii ułatwi użycie ich w gotowym projekcie oraz pomoże obiektywnie je ocenić.

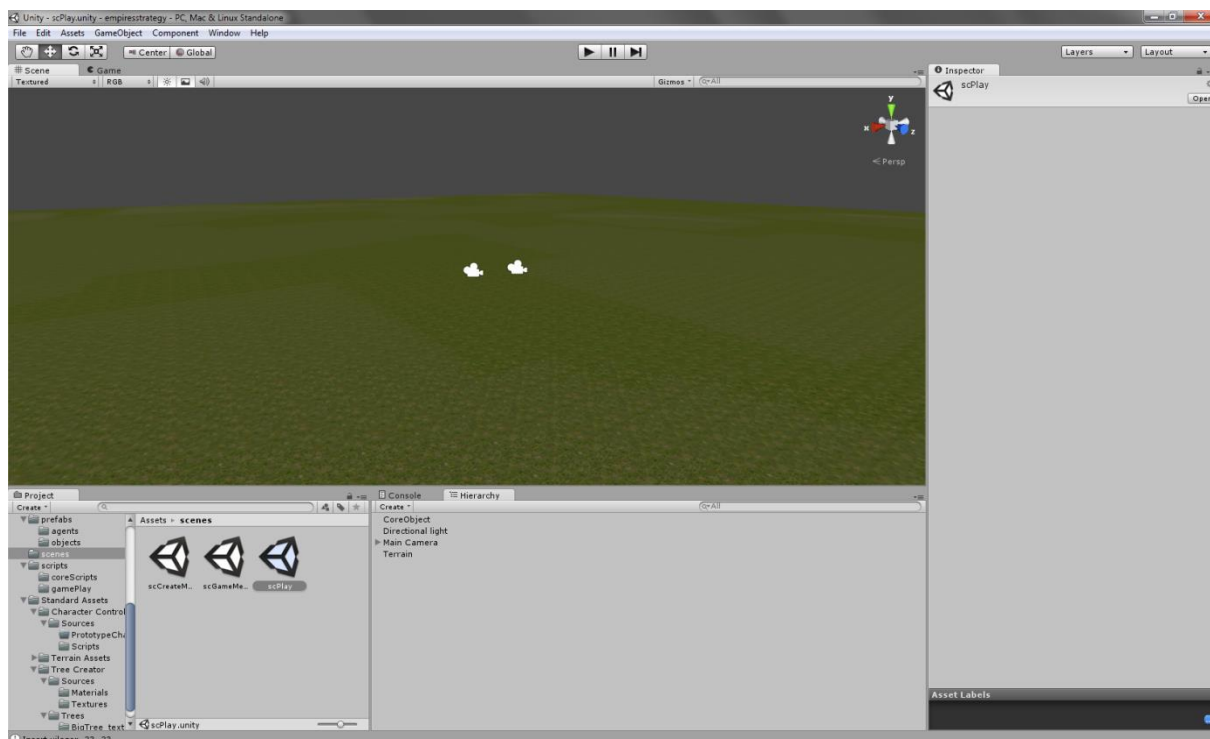
### 1.4.1. Unity 3D

**Unity 3D** [13] to program wraz z dostarczonymi interfejsami programowania aplikacji stworzony specjalnie do celu implementacji gier. W projekcie została użyta bezpłatna wersja silnika (licencja freeware), niepozwalająca na komercyjne użycie, a także nieposiadająca niektórych elementów środowiska.

W edytorze Unity Editor można tworzyć obiekty, przypisywać do nich skrypty czy trójwymiarowe modele, a następnie zbudować gotową już aplikację wybierając docelowy system operacyjny lub działanie w przeglądarce internetowej. Pozwala na tworzenie dwuwymiarowych lub trójwymiarowych scen, import grafiki, modeli 3d, dźwięków czy innych elementów wykorzystywanych w środowisku. Edytor radzi sobie z importowaniem do projektu trójwymiarowych modeli stworzonych w takich programach jak Blender, Maya czy 3D Studio Max wykorzystując ich formaty danych. Potrafi użyć również plików graficznych czy dźwiękowych. Obiekty scen tworzonych w edytorze mogą mieć przypisane skrypty tworzone w językach takich jak JavaScript, Boo oraz C#. W projekcie po stronie aplikacji klienckiej przeważa zdecydowanie **C#**, niektóre elementy oprogramowane są przy użyciu **JavaScript**.

Stworzenie prostej gry nie zawierającej zbyt wiele logiki jak na przykład wyścigi jest w tym środowisku możliwe nawet bez umiejętności programowania, wystarczy wykorzystanie istniejących skryptów oraz odpowiednia parametryzacja ich.

Unity Editor można rozszerzać na podstawie dostarczonego interfejsu i klas przeznaczonych specjalnie w tym celu. Każda właściwie zaimplementowana klasa, która jest odpowiednio przyporządkowana do obiektu w edytorze jest przez niego wyświetlana tak, że bez pomocy edytora tekstowego można zmieniać jej publiczne argumenty, pozwala to na konfigurację publicznych wartości obiektów bez ingerencji w kod źródłowy. Środowisko posiada także narzędzia pozwalające sprawdzić, jakie obciążenie generuje nasz program oraz inne przydatne udogodnienia.



Rys. 1.1 Unity Editor dostarczany wraz z silnikiem Unity 3D.

## 2. Sztuczna inteligencja w grach komputerowych

### 2.1. Pojęcia

Temat pracy operuje na pograniczu nauki, jaką jest informatyka wraz z dziedziną sztucznej inteligencji oraz rozrywki, jaką są gry komputerowe. Obie dziedziny wymagają znajomości zagadnień. Niniejszy rozdział został stworzony w celu przybliżenia pojęć. Pojęcia z jednej z dziedzin mogą być całkowicie obce dla osoby zajmującej się drugą z nich, dlatego ten rozdział jest tak ważny.

**Sztuczna inteligencja** – [8] Dział informatyki zajmujący się tworzeniem algorytmów oraz maszyn charakteryzujących się działaniem posiadającym elementy inteligencji. Oznacza to posiadanie umiejętności przystosowywania się do zadanych warunków, uczenia się, podejmowania złożonych decyzji oraz symulowanie zachowań istot inteligentnych.

**Rozproszona sztuczna inteligencja** - (ang. DAI: distributed artificial intelligence) Dział sztucznej inteligencji zajmujący się rozwiązywaniem problemów przy pomocy autonomicznych obiektów. Rozproszona sztuczna inteligencja wykorzystuje elementy współpracy poszczególnych agentów bez odgórnego zarządzania ich działaniami. Jedną z głównych dziedzin rozproszonej sztucznej inteligencji są systemy agentowe.

**Środowisko agentowe** – [8] Środowisko, w którym działają agenci i nie mogą wyjść poza nie. Agent odbiera od środowiska bodźce, ale również wpływa na nie. Dzięki temu działania jednego agenta wpływają na działanie pozostałych. Środowiskiem takim może być przestrzeń, na której operuje gra komputerowa.

**Agent** – Agentem nazywamy wirtualną jednostkę, która posiada autonomię i jest w stanie interpretować otaczające ją środowisko. Przykładem agenta dla gry w piłkę nożną będzie piłkarz, który potrafi dostosować swoje zachowanie do akcji dziejącej się na boisku.

Agenci dzielą się na cztery typy:

**Agent reakcyjny** – Jest to agent najprostszego typu, a jego jedyną funkcją jest wykonanie akcji w przypadku wystąpienia określonej sytuacji.

**Agent pamiętający stan środowiska** – Ten typ agenta wykonuje akcje w zależności od kilku sytuacji dziejących się w określonej kolejności na przestrzeni czasu. Jest w stanie zapamiętywać wydarzenia z przeszłości.

**Agent z określonym celem** – Agent tego typu posiada wiedzę o swoim strategicznym celu. Przed wykonaniem działania przewiduje, czy zbliży go to do strategicznego celu.

**Agent użytkowy** – Najbardziej rozbudowany typ agenta. Łączy on w sobie cechy wszystkich typów agentów i dodatkowo wprowadza funkcję skuteczności. Funkcja ta pozwala mu określić, jakie działania prowadzą do celu najszybciej, uwzględniając również pewność tego wyboru, koszty czy inne czynniki.

**Graf** – [2] Zbiór wierzchołków, które mogą być połączone krawędziami, przy czym każda krawędź łączy dwa wierzchołki grafu. Zarówno wierzchołki jak i krawędzie grafu mogą być numerowane. Krawędzie grafu mogą posiadać wagi, oznacza to, że przejście z jednego do drugiego wierzchołka jest warte tyle, co waga krawędzi.

**Drzewo decyzyjne** – Metoda wspomagająca proces decyzyjny oparta o drzewa grafowe (grafy nie zawierające cykli) oraz algorytmy grafowe. Drzewa decyzyjne wykorzystywane są w uczeniu maszynowym oraz systemach ekspertowych.

**Heurystyka** – Idea przeszukiwania zbioru rozwiązań metodami nie algorytmicznymi. Jest używana w przypadku, gdy znalezienie optymalnego rozwiązania danego problemu jest zbyt złożone czasowo lub wydajnościowo. Metoda pozwala na znalezienie przybliżonego rozwiązania problemu w znacznie krótszym czasie. Wadą heurystyki jest fakt, iż rozwiązanie może być bliskie optymalnemu, lecz nie koniecznie optymalne.

**Komputerowa gra strategiczna** – Program komputerowy służący, jako jedno lub wieloosobowa gra. Charakteryzuje się faktem, iż głównym czynnikiem wpływającym na zwycięstwo nie jest losowość czy umiejętności manualne, a przyjęta taktyka i strategia długotrwałych działań. Gra taka może polegać na dowodzeniu wojskiem (np. *Red Alert*), sterowaniu ekonomią państwa (np. *Pharaoh*) czy zarządzaniu firmą (np. *Transport Tycoon*). Najczęściej rozgrywa się na dwuwymiarowej płaszczyźnie podzielonej na kwadraty lub sześciiany. Rozgrywka nastawiona jest na dalekosiężne planowanie i odpowiednie zarządzanie gromadzonymi zasobami.

**TBS** - Strategiczne gry turowe (ang. turn-based strategy) – podgatunek komputerowych gier strategicznych, którego główną cechą jest podział czasu gry na tury. Każdy z graczy podczas swojej tury może wykonywać akcje takie jak sterowanie agentami czy rozbudowa budynków, następnie po zakończeniu tury kolejny gracz wykonuje swoje akcje. Jest to gatunek, który powstał, jako pierwszy, czego powodem jest mała moc obliczeniowa pierwszych komputerów. Przykładem turowej gry strategicznej jest seria *Heroes of Might & Magic*.

**RTS** – Strategiczna gra czasu rzeczywistego (ang. real-time strategy) – podgatunek komputerowych gier strategicznych, w którym nie ma podziału na tury, a wszyscy gracze wykonują swoje działania jednocześnie i natychmiastowo. W gatunku tym ważna jest szybkość podejmowania decyzji oraz umiejętność panowania nad działaniami wszystkich agentów. Podchodząc do sprawy ze strony algorytmicznej w tego typu programach nie ma miejsca na skomplikowane, czasochłonne obliczenia podczas rozgrywki. Przykładem gry strategicznej czasu rzeczywistego jest *Red Alert*.

**CRPG** – Komputerowa gra fabularna (ang. Computer Role Playing Game) – gatunek gier komputerowych, w którym gracz wciela się w jedną postać, rozwija ją i kieruje jej działaniami. Cechy charakterystyczne gatunku to rozwijanie licznych współczynników cechujących postać, widok z góry, sterowanie myszą. Często rozgrywka odbywa się za pomocą sieci internetowej. Wtedy w rozgrywce uczestniczy wielu graczy, nierzadko tysiące.

**FPS** – (ang. First-person shooter) – Gatunek gier komputerowych, w których gracz wciela się w jedną postać, a grę widzi oczami tej postaci. Rozgrywka polega najczęściej na zastrzeleniu przeciwnika. Głównymi czynnikami wpływającymi na zwycięstwo jest zręczność, ale również taktyka polegająca na przykład na wykorzystywaniu przeszkód.



## 2.2. Zadanie sztucznej inteligencji w grach komputerowych

W zależności od rodzaju gry komputerowej sztuczna inteligencja może pełnić różne zadania [10]. Niektóre z nich są specyficzne i występują tylko w określonych gatunkach gier. Inne uruchamiane są przed rozgrywką, by przygotować dane do gry.

Najczęstsze wykorzystanie to między innymi:

- Wyszukiwanie drogi na planszy z przeszkodami. Polega na znalezieniu najkrótszej drogi między dwoma punktami na płaszczyźnie lub w przestrzeni. Powinno być możliwie szybkie. Często stosuje się algorytmy zwiększające płynność znajdowania drogi lub wygładzające tę drogę. Używane głównie w grach strategicznych oraz CRPG.
- Sterowanie „agentami”. Polega na sterowaniu przeciwnikami gracza w taki sposób, by symulować racjonalne zachowania oraz zapewnić odpowiednio trudne wyzwanie dla gracza. Występuje w niemal wszystkich gatunkach gier. Czasami stosuje się mechanizmy pozwalające na przygotowanie danych do sterowania agentami. Przykładem takiego działania mogą być gry polegające na wyścigach samochodów, gdzie sztuczna sieć neuronowa jeszcze przed rozgrywką oblicza optymalne trasy przejazdu. Inny przykład to wyliczenie na planszy gry FPS strategicznych punktów, w których ustawiać się będą „agenci”.
- Wspomaganie decyzji gracza. Polega na częściowym sterowaniu obiektami należącymi do gracza lub informowaniu go o optymalnych strategiach.
- Komentowanie akcji. Wykorzystywane głównie w grach sportowych. Na podstawie różnych czynników sztuczna inteligencja określa jak przebiega rozgrywka.
- Symulowanie realnych zachowań. Wykorzystywane w niemal wszystkich najnowszych grach. Sztuczna inteligencja ma za zadanie symulować zachowania agentów: ich ruchy, uczucia widoczne na twarzy a także nastawienie w stosunku do gracza.

## 2.3. Historia sztucznej inteligencji w grach komputerowych

Temat sztucznej inteligencji towarzyszył grom komputerowym od samego początku ich istnienia, kiedy to w latach 50. XX wieku *Arthur Samuel* [16] stworzył algorytm gry w warcaby. Algorytm ten był w stanie pokonać najlepszych graczy w USA. Gry dla jednego gracza z przeciwnikami zaczęły pojawiać się w 1970 roku. Jedne z pierwszych to gry na komputerze atari: *Speed race* (wyścigi) i *Pursuit* (symulator samolotu bojowego). W tamtych czasach powstały najbardziej znane gry takie jak *Space invaders* (1978), *Pac man* (1980). Najciekawszą jest pierwsza gra RPG *First Queen* (1988), w której postaci były kontrolowane przez sztuczną inteligencję.

W latach 90. XX [10] wieku nastąpiło ogromne zwiększenie udziału sztucznej inteligencji w grach komputerowych. Powstały wtedy pierwsze gry strategiczne czasu rzeczywistego wymagające obsługi automatów stanów skończonych i algorytmów wyszukiwania drogi. Mechanizmy te były bardzo proste, przez co łatwe do oszukania, a wyszukiwanie drogi często działało błędnie. Wtedy okazało się, że należy wprowadzić odpowiedni poziom trudności w grach oraz sprawić, że sztuczna inteligencja nie będzie tak przewidywalna.

Ogromnym przełomem okazały się gry *Warcraft* (1994), *Unreal* (1999), *Black&White* (2001) oraz *Colin McRae Rally* (1998). W grach tych na szeroką skalę wykorzystano algorytmy wyszukiwania drogi, agentów naśladowujących zachowania graczy oraz uczenie się agentów przed i podczas rozgrywki.

Obecnie gry komputerowe są ogromnymi programami, w których sztuczna inteligencja pełni bardzo wiele ról, na bardzo różnych poziomach. Steruje zarówno otoczeniem, graczem jak i agentami w grze.

## 2.4. Automaty stanów skończonych

Automaty stanów skończonych to mechanizm pozwalający definiować, w jakim stanie znajduje się dany obiekt oraz jaka akcja powoduje przejście do innego stanu. Jest to abstrakcyjny matematyczny model zachowania dynamicznego obiektu. Mechanizmy te wykorzystywane są w grach komputerowych od początku lat 90. Jedną z pierwszych gier wykorzystujących automaty stanów była gra RPG *First Queen* (1988), której automat stanów zajmował się zachowaniem agentów będących przeciwnikami.

Automaty stanów dzielą się na deterministyczne i niedeterministyczne. W sztucznej inteligencji w grach wykorzystuje się automaty deterministyczne. Czasami wykorzystuje się tak zwane abstrakcyjne automaty stanów, charakteryzujące się faktem, iż stanem nie jest wartość logiczna, lecz cała struktura danych.

Zastosowanie sztucznej inteligencji w grach można podzielić na dwa obszary [1][7]. Obszar odpowiedzialny za sytuację systemu, czyli bezpośrednio związany z rozgrywką, oraz obszar odpowiedzialny za zachowanie systemu, a więc wszystkie elementy niewpływające na rozgrywkę, ale dające graczowi więcej wrażeń, symulujące realizm. Obiema tymi obszarami zajmują się właśnie automaty stanów skończonych. Pozwalają na obsługę stanu gry i podejmowanie w oparciu o nie decyzji. Obsługują także sferę zachowania, a w szczególnych przypadkach obsługują komentowanie wydarzeń oraz zarządzanie dialogami.

## 2.5. Logika rozmyta

Logika rozmyta [7] jest ściśle powiązana z automatami stanów. Logika rozmyta jest elementem teorii zbiorów rozmytych i teorii prawdopodobieństwa. Pozwala podejmować decyzje o wykonaniu akcji na podstawie bardzo wielu liniowych czynników. Ten dział sztucznej inteligencji zajmuje się przypisaniem wartościom ciągłym kilku wartości dyskretnych, na których będzie można operować. Pozwala jednak na rozmycie granicy między wartościami wprowadzając pośrednie stany logiczne. W ten sposób niektóre z analizowanych wartości mogą należeć do dwóch pozornie wykluczających się zbiorów.

Przykładem może być przypisanie stanom produktu na magazynie wartości: stan niski, stan średni, stan wysoki. Klasyczna logika pozwala na określenie, iż granicą między tymi stanami jest określona wartość liczbowa. Na przykład stan magazynowy niski byłby określony poprzez mniej niż 40% produktu, a stan wysoki poprzez więcej niż 60% produktu, wartości 40%-60% określane byłyby poprzez stan średni. Logika rozmyta pozwala na rozmycie tej granicy liczbowej, dzięki czemu graniczne wartości będą zawierały się w obydwu z grup. Opisywane obiekty prócz samej przynależności będą miały również przypisany stopień przynależności do danej grupy. Logikę przynależności można rozmywać w różnym stopniu, tak by dopasować ją do oczekiwanych rezultatów. Dzięki temu stanowi o wartości 45% można przypisać zarówno zbiór stan niski, jak i zbiór stan średni. Stopień przynależności do obu grup wynosiłby 50%. Daje to możliwość podjęcia działań przypisania obu zbiorom oraz skorygowania ich o współczynnik przynależności.

Korzyści, jakie może nadać to mechanizmowi wykorzystywanym w grach są bardzo widoczne dla gracza. Nadaje to grze więcej realizmu i eliminuje sytuacje, gdzie zmiana jakiegoś współczynnika w grze o jeden zmienia całkowicie zachowanie środowiska.

## 2.6. Inteligencja stadna

Inteligencja stadna jest jednym z elementów nadających grze realizmu. Steruje ona grupą agentów sprawiając wrażenie jedności. Pierwszy algorytm stadny został przedstawiony przez Craiga Reynoldsa w 1987 roku. Opisał on 4 cechy takich algorytmów: [16] rozdzielczość (zachowywanie odległości), wyrównanie (uzależnienie prędkości i kierunku od sąsiednich agentów), spójność (zbieranie agentów w grupy) oraz unikanie (zapobieganie zderzeniom z przeszkodami). Algorytmy takie cyklicznie aktualizują cele poszczególnych agentów. Polega to na wzajemnej współpracy wielu agentów sterowanych poprzez sztuczną inteligencję, nie istnieje jednak żaden globalny obiekt sterujący ogółem agentów. Bardzo dobrym przykładem użycia takiego mechanizmu jest gra *Age of empires 2* (1999), w której jednostki można zebrać w grupę tworząc 4 różne formacje bojowe. Podczas poruszania się wpływ na ich drogę mają przeszkody, przeciwnicy oraz skład poruszającej się grupy. Podobne algorytmy prócz wykorzystania w grach strategicznych używane są również do symulowania zachowania grup obiektów. Jednym z najbardziej spektakularnych sposobów użycia tego mechanizmu jest wykorzystanie go w filmach do symulowania przemaszchu wojsk, czego przykładem może być seria *Władca pierścieni* (2001 - 2003). W filmie tym występowały sceny wojny z tysiącami żołnierzy sterowanymi przez różnego typu algorytmy stadne, jedna ze scen zawierała 250 000 sztucznych agentów symulujących zachowania stadne.

Zachowania zwierząt stały się dodatkowo inspiracją do stworzenia algorytmów ogólnego przeznaczenia pozwalających rozwiązywać różne problemy decyzyjne czy optymalizacyjne. Istnieją trzy podstawowe algorytmy stadne zbudowane na podstawie świata natury i działające w oparciu o nie, są to: PSO (ang. Particle swarm optimization) - optymalizacja stadna cząstek, systemy mrówkowe oraz systemy rojowe. Klasycznym przykładem wykorzystania takiego algorytmu jest implementacja algorytmu mrówkowego dla problemu komiwojażera. Problem ten polega na znalezieniu takiej ścieżki w grafie, która odwiedza wszystkie wierzchołki przy możliwie najmniejszej wadze krawędzi. Algorytm ten nie zawsze znajduje jednak rozwiązanie optymalne, ale działa w stosunkowo krótkim czasie i daje rozwiązanie bliskie optymalnemu. Tego typu problemy, choć występują w grach, są dość rzadkie, dlatego nie zostaną szerzej poruszone.

## 2.7. Algorytmy grafowe i heurystyczne przeszukiwanie drogi

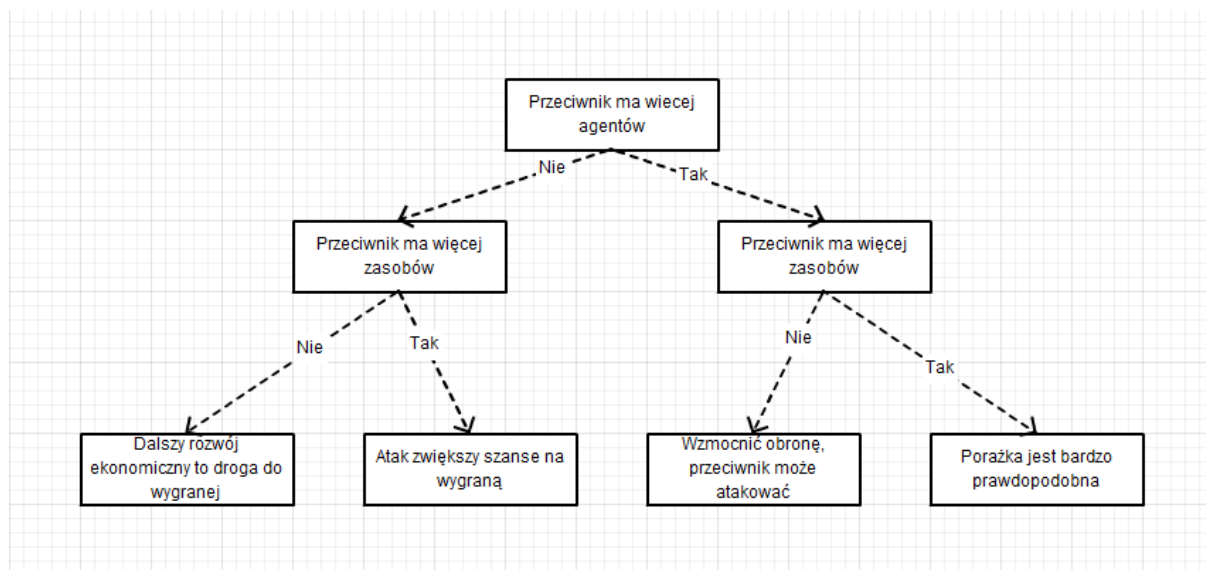
Głównym wykorzystaniem algorytmów grafowych w grach jest wyszukiwanie najkrótszej ścieżki między dwoma punktami na płaszczyźnie lub w przestrzeni. Przykładem takiego wykorzystania są gry RPG, RTS, FPS, gdzie rozgrywka ma miejsce na płaszczyźnie lub w przestrzeni. Istnieje kilka podstawowych algorytmów przeszukiwania grafu.

Najpopularniejsze to [2] algorytm Dijkstry oraz algorytm Forda-Bellmana. Jednak w przypadku gier komputerowych najczęściej używany jest algorytm A\*, mniej popularny dla innych dziedzin informatyki. Sprawdza się on szczególnie dobrze do wyszukiwania najkrótszej drogi na siatce. Graf, który będzie przekształcany można przedstawić, jako zbiór wierzchołków i krawędzi, jako macierz połączeń między wierzchołkami lub jako siatkę. Ostatni sposób znacznie ogranicza zbiór analizowanych grafów, jednak idealnie nadaje się do gier komputerowych. Wszystkie 3 algorytmy zostaną w niniejszym projekcie zaimplementowane i przetestowane w celu zweryfikowania tych informacji.

Algorytmy grafowe mogą być również używane przy innych funkcjonalnościach gry komputerowej jak na przykład drzewa decyzyjne, algorytm mini-maxu. Jednak te zagadnienia nie są tak ważne jak wyszukiwanie drogi.

## 2.8. Drzewa decyzyjne

Drzewa decyzyjne [6] są mechanizmem pozwalającym na podejmowanie decyzji w oparciu o różne czynniki, o których informacja jest zapisana w strukturze drzewa. Charakteryzują się faktem, iż mogą być stworzone zanim rozpocznie się rozgrywka i wykorzystywane podczas rozgrywki. Jest to ogromną zaletą, ponieważ tworzenie ich może być procesem długotrwałym, zaś wykorzystywanie nie jest skomplikowane. Mechanizm ten został stworzony głównie do rozwiązywania skomplikowanych problemów, takich jak rozpoznawanie obrazu, wtedy drzewo tworzone jest na podstawie procesu uczenia. Może być jednak wykorzystany dla nieskomplikowanych problemów decyzyjnych, a dobra, transparentna implementacja tego mechanizmu pozwala na łatwe, przejrzyste tworzenie logiki gry. Prosty przykład takiego użycia widać na rysunku (Rys. 2.1). Odizolowanie drzew decyzyjnych od pozostałego kodu wydaje się być bardzo dobrą praktyką.



Rys. 2.1 Przykładowe drzewo decyzyjne dalszej strategii gry.

## 2.9. Drzewa gier

Mechanizmy grafowe pozwalają na budowę drzewa gry [11], które analizowane algorytmem mini-maxu znajdzie optymalną strategię wygranej. W ten sposób sztuczna inteligencja jest w stanie wygrać grę w kółko i krzyżyk, szachy czy warcaby na podstawie znajomości drzewa gry i ruchów przeciwnika. Dzięki algorytmowi mini-maxu obiera optymalną strategię. Algorytm ten, polega na wykonywaniu takich ruchów, by przy założeniu, iż przeciwnik wykona najlepszy dla niego ruch maksymalizować swoje zyski. Stosowanie tego algorytmu niestety ma sens jedynie w przypadkach gier nazywanych grami o zerowej sumie. Jest to gra, w której zysk jednego gracza oznacza stratę drugiego, a przykładem takiej gry są właśnie szachy czy kółko i krzyżyk. Nie znaczy to jednak, że algorytmy nie bywają przydatne w większych projektach, ponieważ również tam zdarzają się przypadki, które da się rozwiązać w ten sposób.

Tego typu mechanizmy nie są potrzebne przy budowaniu mało zaawansowanej sztucznej inteligencji. Są jednak niezbędne przy tworzeniu sztucznej inteligencji mającej pokonać inteligencję człowieka. Ważnym czynnikiem jest konieczność budowy drzewa gry przez rozgrywką. Sama budowa może okazać się czasochłonna, jednak przeprowadzenie jej przed rozgrywką sprawia, iż wada ta jest marginalna.

## 2.10. Sztuczne sieci neuronowe

Sztuczne sieci neuronowe to algorytmy symulujące pracę mózgu. Algorytm składa się z prostych jednostek obliczeniowych, symulujących neurony, które przetwarzają dane pracując równolegle, komunikując się ze sobą. Przyjmują one pewne wartości wejściowe i według swoich parametrów wyliczają wartości wyjściowe. Parametry (wagi) sieci neuronowej zmieniają się podczas uczenia. Uczenie polega na podawaniu wartości wejściowych wraz z wartościami wyjściowymi, a sieć dostosowuje swoje parametry wewnętrzne. Każda sieć neuronowa może być zdefiniowana poprzez określenie modelu neuronu, topologii sieci oraz reguł uczenia sieci.

Sieci neuronowe można podzielić na jednokierunkowe oraz sieci ze sprzężeniami zwrotnymi. W pierwszej grupie sygnały przebiegają tylko w jedną stronę, mogą być to sieci warstwowe, które są najpopularniejsze. Uczenie sieci neuronowych w zdecydowanej większości wykorzystuje algorytm wstecznej propagacji błędów, który polega na lokalnej zmianie wagi neuronu oraz wysłaniu informacji do neuronów warstwy poprzedniej o potrzebie zmiany wagi neuronu w mniejszym stopniu. Dzięki temu zmiana wag następuje w wielu neuronach na zasadzie gradientu, zmniejszając intensywność zmiany w każdym kroku.

Mechanizm ten został użyty po raz pierwszy w dziedzinie gier komputerowych w roku 2000 w grze *Colin McRae Rally 2*. Posłużył wtedy do wyznaczenia optymalnej trasy przejazdu toru wyścigowego. Wyznaczał trasę na podstawie rodzaju gruntu, odległości do najbliższego zakrętu oraz krzywizny tego zakrętu. Był on uruchamiany przed rozgrywką i dostarczał danych wykorzystywanych przez agentów sterowanych sztuczną inteligencją podczas rozgrywki.

Największą zaletą tej metody jest możliwość dostosowania trudności rozgrywki do gracza, nauka nowych taktyk stosowanych przez rzeczywistych graczy. Największą wadą metody są skomplikowane, wymagające zasobów obliczenia. Zatem algorytmy te nie mogą działać w trakcie rozgrywki, lub działać w uproszczonej formie, a wyniki ich działania oraz dane wejściowe mogą zajmować sporo pamięci.

Sztuczne sieci neuronowe idealnie nadają się do tworzenia drzew decyzyjnych, które pozwolą podjąć decyzję na podstawie wielu zróżnicowanych czynników. Może być to zarówno wykorzystane do sterowania zachowaniem, komentowania wydarzeń czy sterowania skomplikowaną fabułą gry. Sterowanie zachowaniem wykorzystywane jest na przykład przy wspomnianych wyścigach samochodów. Komentowanie wydarzeń jest funkcjonalnością charakterystyczną dla gier sportowych. Skomplikowana fabuła gier to funkcjonalność występująca głównie w grach RPG. Jeśli przypadek jest rozbudowany wykorzystanie sztucznej sieci neuronowej na pewno okaże się wydajniejsze niż proste instrukcje warunkowe.

### 3. Projekt komputerowej gry strategicznej

#### 3.1. Plansza

W stworzonej grze plansza jest płaszczyzną podzieloną na kwadratowe pola. Układ taki jest bardzo zbliżony do idei szachownicy. Ze strony struktury gry podział ten jest bardzo widoczny, jednak graficzna reprezentacja powinna ten układ ukrywać, by gra nie wydawała się schematyczna.

Na planszy umieszczone są obiekty, każdy obiekt znajduje się w określonym sześciacie. Obiekty dzielą się na cztery podstawowe grupy: przeszkody, zasoby, agenci, budynki. Każde pole może zajmować maksymalnie jedna przeszkoda. Jeśli na danym polu nie ma przeszkód może zajmować je wiele jednostek i jeden zasób. Jednym z podstawowych wyzwań dla sztucznej inteligencji, jakie nasuwa się przy takich założeniach jest wyszukiwanie przez agentów drogi na płaszczyźnie. Należy znaleźć najkrótszą drogę między dwoma polami, a także sprawdzić czy taka istnieje.

Przeszkodą występującą na planszy są kamienie, jedyną ich funkcją jest blokowanie drogi. Są obiektami, których nie można usunąć, powodem ich istnienia jest utrudnienie w poruszaniu się po planszy. W rozgrywce istnieją trzy różne zasoby: *złoto*, *drewno* oraz *jedzenie*. Zasoby należy kolekcjonować za pomocą agentów i wykorzystać do uzyskania przewagi. Zasoby nie są przeszkodą, zatem nie blokują przejścia. Najważniejszymi obiektami na planszy są budynki, których przejęcie decyduje o wygranej. Po tak zdefiniowanej planszy poruszają się agenci graczy. Różnią się oni jedynie kolorem symbolizującym przynależność.



Rys. 3.1 Plansza gry.





Rys. 3.2 Legenda obiektów na planszy.



Rys. 3.3 Plansza gry zawierająca specyficzne przeszkody.

W zdecydowanej większości gier strategicznych przeszkody układają się w specyficzne kształty rozdzielające przestrzeń gry. Pozwala to na użycie bardziej skomplikowanych taktyk, takich jak zajęcie i obrona strategicznych punktów. Przykład takiej specyficznej planszy widać na rysunku (Rys. 3.3).

### 3.2. Zasady rozgrywki

Stworzona gra komputerowa odbywa się na płaszczyźnie o strukturze szachownicy. W rozgrywce uczestniczy dwóch graczy, z których jeden sterowany jest przez sztuczną inteligencję. W grze istnieją cztery typy obiektów, które mogą znaleźć się na polu, są to: budynek, agent, przeszkoda, zasób.

Budynki są obiektami tworzącymi jednostki, mogą one również magazynować zasoby. Stworzenie jednostki kosztuje określoną liczbę zasobów. Należy obrać odpowiednią strategię zbierania oraz zużywania konkretnych zasobów, aby zmaksymalizować swoje szanse zwycięstwa. Agenci zbierający zasoby mają określoną prędkość zbierania oraz mogą unieść ograniczoną ilość zasobów. Wartości te można ulepszać również za pomocą zebranych surowców, co daje kolejne możliwości strategiczne.

Agenci, prócz zbierania zasobów mogą zdobyć budynek przeciwnika. Gdy dwóch agentów przeciwnych drużyn stanie na sąsiednich polach obaj znikają. Pozwala to na utrudnianie przeciwnikowi kolekcjonowania surowców, co jest dodatkowym elementem taktycznym ubarwiającym rozgrywkę.

### 3.3. Warunki zwycięstwa

Warunkiem zwycięstwa jest zdobycie budynku przeciwnika, co można osiągnąć poprzez dotarcie określoną liczbą agentów do tego budynku. Przeciwna drużyna będzie się starać bronić swojego budynku własnymi agentami. Liczba agentów wymagana do przejścia budynku drużyny przeciwniej wynosi 10.

Podsumowując, na zwycięstwo składa się odpowiednia strategia gromadzenia oraz wydawania surowców, odpowiednia taktyka utrudniania gry przeciwnikowi, a także wybór odpowiedniego momentu do przejścia budynku przeciwnika. Czynnikiem, który wpłynie na wygraną będzie również umiejętne operowanie agentami w taki sposób, by żaden z nich nie stał bezczynnie. Oznacza to, iż gracz sterowany przez sztuczną inteligencję będzie miał w tej kwestii przewagę.

### 3.4. Funkcjonalności

W programie zostały zaimplementowane różne metody wyszukiwania drogi, oparte na algorytmach grafowych oraz na algorytmie A\*. W trakcie rozgrywki można zmienić wykorzystywany algorytm. Ważną rzeczą jest **możliwość zapisywania wyników działania algorytmów do plików tekstowych**. Zapisywane są tam czasy działania, a także w przypadku wykonywania pomiaru czasu działania zapisywane są wszystkie pośrednie wyniki działania w celu sprawdzenia poprawności danych. Przykładem takich wyników jest zapis tekstowy wszystkich rodzajów prezentowania planszy oraz zapis pól, po których porusza się wybrany algorytm.

Sztuczna inteligencja gracza sterowanego przez komputer opiera się na dwóch różnych automatach stanów opartych o logikę standardową lub logikę rozmytą. Gracz może zmienić wykorzystywany automat stanów, by zmienić trudność i jakość rozgrywki. Jeden z automatów charakteryzował się bardziej agresywnym sposobem gry (automat wojownik), drugi zaś początkowo pokojowym nastawieniem skupiającym się na zwiększeniu swojej przewagi poprzez ulepszenia swoich jednostek (automat budowniczy).

W celu przeprowadzenia testów mających praktyczny sens został stworzony mechanizm **zapisu planszy do pliku tekstowego** oraz wczytywania takich plików. Pozwoliło to na stworzenie planszy, która dla algorytmów wyszukujących drogę stanowiła większe wyzwanie. Dzięki temu lepiej przetestowano algorytm A\* oraz przeszukiwane przez niego pola.

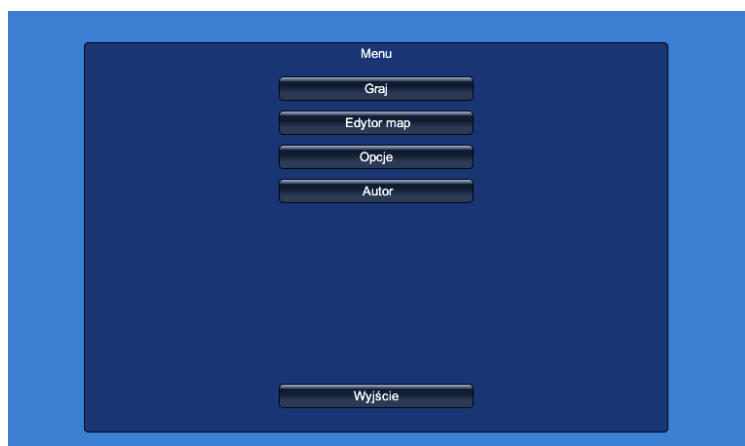
Program jest oczywiście wyposażony w podstawowe funkcje jak operowanie kamerą poprzez przesuwanie jej nad planszą, obsługa zaznaczania obiektów gry czy zaznaczanie grupowe. Pozostałe funkcjonalności, związane z graficznym interfejsem zostały opisane w dziale 4.1 Graficzny interfejs użytkownika.



## 4. Implementacja komputerowej gry strategicznej

### 4.1. Graficzny interfejs użytkownika

Interfejs użytkownika jest podzielony na dwa obszary: menu główne oraz interfejs rozgrywki. Menu główne widoczne jest na rysunku (Rys. 4.1), niestety ograniczony czas nie pozwolił na implementację wszystkich przygotowanych opcji. Nie są one jednak bezpośrednim tematem pracy. Graficzny interfejs rozgrywki widać na rysunku (Rys. 4.2). Jak widać dzieli się on na kilka obszarów. Są to: obszar rozgrywki, górny pasek, menu gry (po prawej stronie) oraz mapa (widoczna w prawym dolnym rogu). Obszar menu, gdy nie jest wybrane żadne menu wyświetla specyficzne informacje zaznaczonego agenta, co widać na rysunku (Rys. 4.3). Informacje te zawierają jego parametry, bezpośrednio dotyczące rozgrywki jak i informacje o stanie, w jakim się znajduje pod względem sztucznej inteligencji. Jeśli zostanie wybrane menu główne wyświetla się to, co widać na rysunku (Rys. 4.4), jego przyciski uruchamiają kolejne menu, wszystkie z nich są widoczne na rysunku (Rys. 4.5). Przycisk „Mapa” pozwala na wygenerowanie losowej planszy, zapisu planszy do pliku tekstowego, lub wczytaniu planszy z pliku tekstowego. Przycisk „Tester” pozwala na uruchomienie testów wykorzystanych algorytmów wraz z zapisaniem wyników do pliku. Czas trwania takich testów może wynieść kilkanaście godzin. Przycisk „Opcje” pozwala na włączanie i wyłączanie elementów rozgrywki, w tym zmianę algorytmów sztucznej inteligencji. Przycisk „Budynek” jest bezpośrednim elementem rozgrywki i pozwala na tworzenie oraz ulepszanie agentów. Widoczna na rysunku (Rys. 4.6) mini mapa, pozwala na oglądanie rozgrywki z szerszej perspektywy. W programie zostały wykorzystane trójwymiarowe modele, będące częścią środowiska Unity3d, oraz dwa trójwymiarowe modele niebędące częścią środowiska, pobrane ze strony udostępniającej darmowe modele [12].



Rys. 4.1 Menu główne gry.



Rys. 4.2 Graficzny interfejs użytkownika dla rozgrywki.



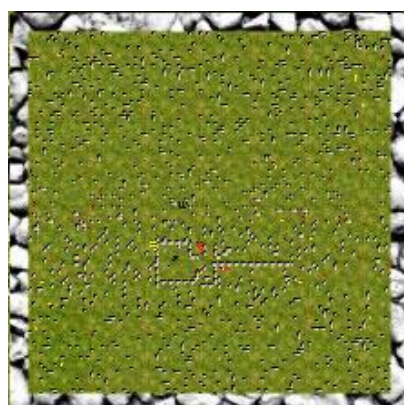
Rys. 4.3 Informacje o agencji wyświetlane w obszarze menu rozgrywki.



Rys. 4.4 Menu główne rozgrywki.



Rys. 4.5 Wszystkie menu rozgrywki.



Rys. 4.6 Mini mapa.





















## 4.2. Struktura programu

Wykorzystywane środowisko Unity3d podczas tworzenia aplikacji wymusza własną strukturę katalogów. Struktura ta została rozwinięta o dodatkowe foldery, co przedstawia rysunek (Rys. 4.7). Widać tu foldery i pliki stworzone przez środowisko Unity3d:

- Assets (modele, dźwięki, tekstury wykorzystywane w projekcie)
- Library (pliki potrzebne do kompilacji)
- Obj (pliki potrzebne do kompilacji)
- ProjectSettings (ustawienia projektu)
- Temp (pliki tymczasowe)
- Pliki projektu Visual Studio

Poza tym dodane zostały foldery i pliki:

- Logs (logi wyjątków)
- Pomiar (logi poprawności algorytmów oraz pomiary ich czasu trwania)
- Source (pliki źródłowe bibliotek zawierających kod źródłowy gry)
- Pliki map (zapis planszy gry w formacie tekstowym).

	Assets	2013-12-11 12:46	File folder	
	Library	2014-01-22 10:36	File folder	
	Logs	2014-01-22 10:28	File folder	
	obj	2013-09-06 22:46	File folder	
	pomiar	2014-01-11 15:25	File folder	
	ProjectSettings	2014-01-07 21:27	File folder	
	source	2013-09-06 22:50	File folder	
	Temp	2014-01-22 11:03	File folder	
	Assembly-CSharp.csproj	2013-09-06 23:06	Visual C# Project f...	4 KB
	Assembly-CSharp.csproj.vspssc	2013-09-06 22:49	Visual Studio Sour...	1 KB
	Assembly-CSharp.pdb	2013-09-15 17:40	PIDB File	7 KB
	Assembly-CSharp-vs.csproj	2013-12-14 22:27	Visual C# Project f...	5 KB
	empiresstrategy.sln	2013-09-06 22:49	Microsoft Visual S...	2 KB
	empiresstrategy.suo	2013-12-14 21:44	Visual Studio Solu...	86 KB
	empiresstrategy.userprefs	2013-09-15 17:40	USERPREFS File	1 KB
	empiresstrategy.vsscc	2013-09-06 22:49	Visual Studio Sour...	1 KB
	empiresstrategy-csharp.sln	2013-09-06 22:46	Microsoft Visual S...	2 KB
	empiresstrategy-csharp.suo	2014-01-11 17:39	Visual Studio Solu...	31 KB
	map.txt	2014-01-11 22:43	Text Document	314 KB
	map1.txt	2014-01-12 20:58	Text Document	314 KB

Rys. 4.7 Struktura plików.



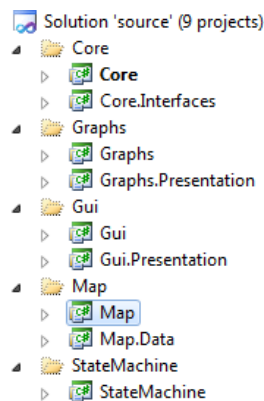
W środowisku Unity3d do każdego obiektu można dodawać skrypty obsługujące jego działanie. Możliwe jest obsłużenie między innymi zdarzenia pojawienia się obiektu lub zachowania obiektu podczas rysowania każdej klatki animacji. Skrypty podpięte do obiektów wykorzystują stworzone do tego celu osobne biblioteki stworzone w języku C#. Biblioteki te zawierają większość kodu źródłowego gry, a ich strukturę prezentuje rysunek (Rys. 4.9). Najważniejsze elementy gry takie jak obsługa graficznego interfejsu użytkownika, kamery czy zarządzania głównymi danymi gry obsługiwane są poprzez obiekt **CoreObject** widoczny na rysunku (Rys. 4.8). Prócz tego obiekty występujące w grze wielokrotnie tworzone są na podstawie bytu nazywanego **prefab** [14]. Byt ten może być zdefiniowany przed rozgrywką, a następnie wielokrotnie dodawany lub usuwany do sceny, idea jego istnienia polega właśnie na definiowaniu skomplikowanych konfiguracji bytu. Najważniejsze z tych bytów również przedstawione są na rysunku (Rys. 4.8).



Rys. 4.8 Najważniejsze obiekty gry w środowisku Unity3d.

Do omawianych obiektów przypisane są skrypty dziedziczące po klasie **MonoBehaviour** [15], co pozwala na obsługę zdarzeń dostarczanych przez środowisko. Wykorzystywane zdarzenia to głównie:

- Awake – uruchamiane po załadowaniu skryptu
- Update – uruchamiane z każdym przerysowaniem animacji
- OnGUI – uruchamiane z każdym przerysowaniem graficznego interfejsu
- Start – uruchamiane w pierwszej ramce animacji
- OnMouseOver – uruchamiane, gdy kursor znajdzie się w obrębie obiektu przy uwzględnieniu położenia kamery.



Rys. 4.9 Struktura projektu.

Program oparty jest o serwisy bazujące na idei singletonów. Występujące w programie serwisy to:

- SceneLoaderService (przeładowywanie scen gry)
- AgentService (zarządzanie agentami graczy)
- MatchDataService (przechowywanie danych rozgrywki)
- PlayerDataService (przechowywanie danych gracza)
- CameraService (sterowanie kamerą)
- PrefabService (zarządzanie predefiniowanymi obiektami graficznymi)
- TextureService (zarządzanie teksturami gry)
- PathFinderService (wyszukiwanie ścieżki)
- ServiceA (implementacja algorytmu A\*)
- ServiceDijkstra (implementacja algorytmu Dijkstry)
- ServiceFord (implementacja algorytmu Forda-Bellmana)
- GuiService (wyświetlanie przycisków graficznego interfejsu)

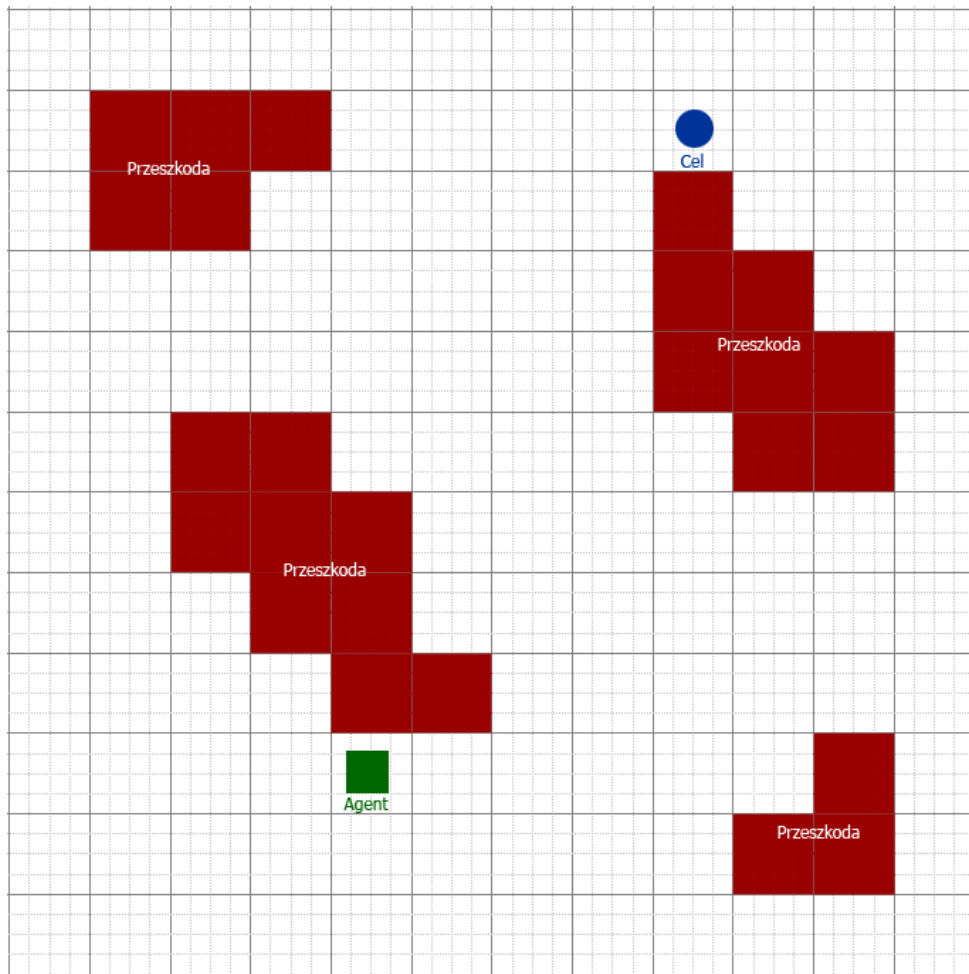
W programie występuje kilka głównych klas będących ważnymi elementami kodu, są to:

- Player (przechowuje dane konkretnego gracza)
- Agent (przechowuje informacje o agencie oraz odpowiada za jego sztuczną inteligencję)
- MyGameObject (jest obiektem bazowym dla wszystkich pozostałych obiektów występujących na planszy)

## 5. Zastosowane metody sztucznej inteligencji

### 5.1. Reprezentacja planszy

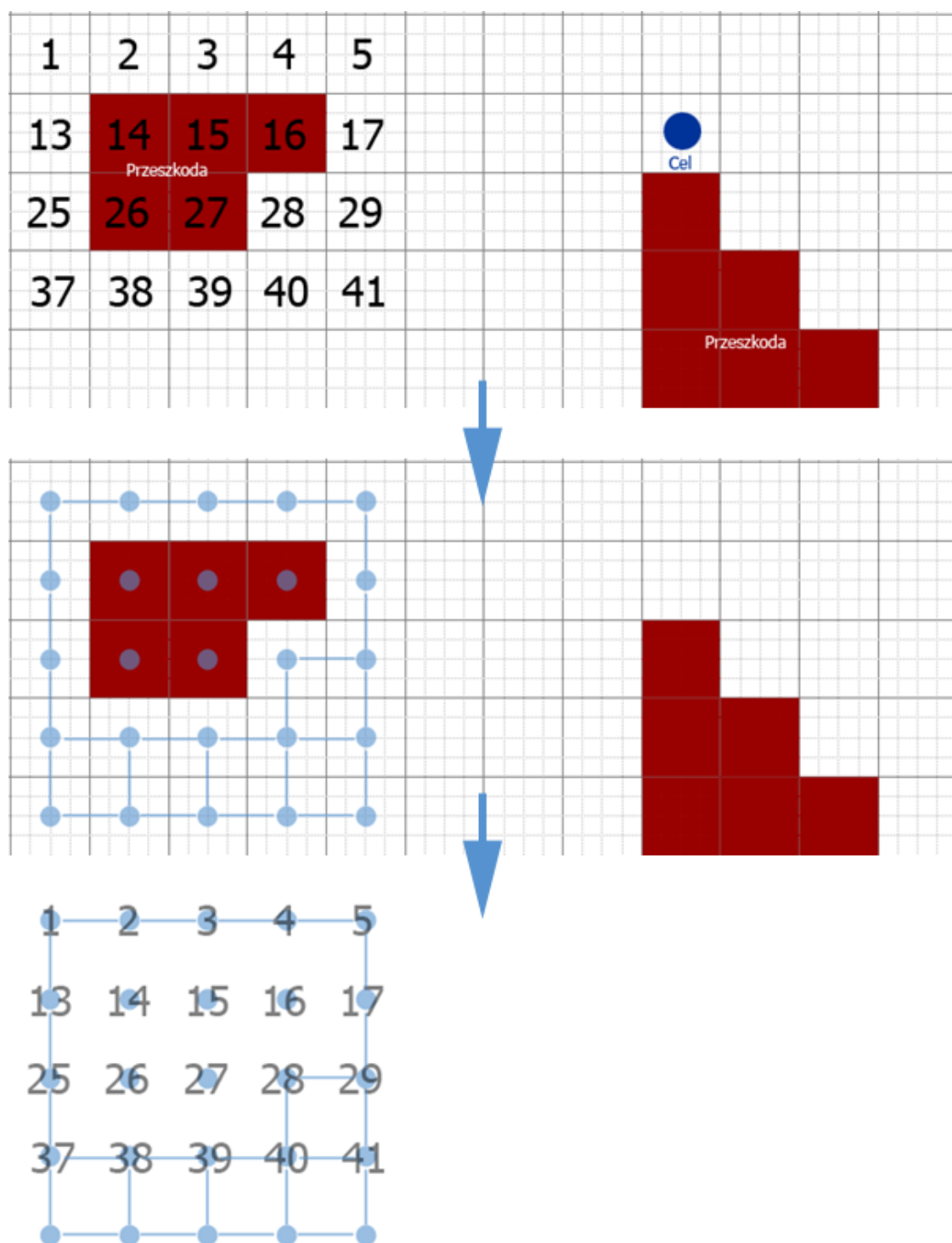
W przeważającej większości strategicznych gier komputerowych świat gry jest płaszczyzną podzieloną na kwadraty, co przypomina szachownicę. Najprostszym sposobem reprezentacji takiej planszy w programie jest **tablica dwuwymiarowa zawierająca obiekty**. Każdemu polu tablicy odpowiada pole w grze. Należy zaznaczyć, że pola mogą być puste, a obiekty mogą przemieszczać się tylko do sąsiednich pól.



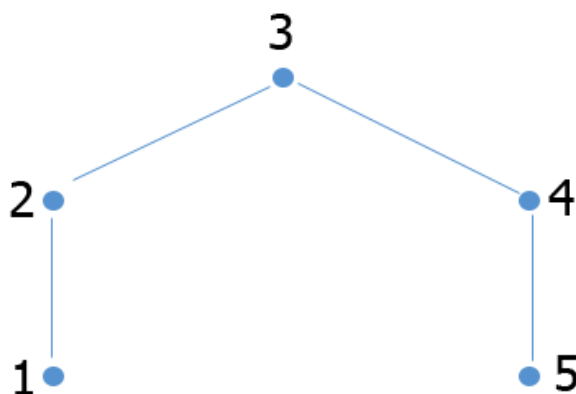
Rys. 5.1 Przykładowa plansza do gry.

Operowanie na takiej planszy może być mało wydajne, dlatego należy przetestować czy algorytmy grafowe nie dadzą lepszych rezultatów. W informatyce grafem jest struktura:  $G=(V, E)$ , gdzie V to wierzchołek (ang. Vertex), a E to krawędź (ang. Edge).

W celu wykorzystania algorytmów operujących na grafach niezbędny jest specyficzny zapis struktury planszy. Plansza musi być przekształcona w graf oraz zapisana w jednym z czterech sposobów: macierz sąsiedztwa, lista sąsiedztwa, macierz incydencji lub lista incydencji. Tworzony graf jest specyficzny, zatem należy określić kilka reguł.



Rys. 5.2 Proces przekształcania planszy w graf.



Rys. 5.3 Fragment grafu z rysunku 5.2. Posłuży za dalszy przykład.

Pierwszą operacją, jaką należy wykonać w celu przekształcenia planszy na graf jest ponumerowanie pól i przypisanie im wierzchołków. Następnie należy określić, czym jest dla tej planszy krawędź i które wierzchołki zostaną połączone krawędziami. Krawędzią jest połączenie między dwoma sąsiednimi polami (wierzchołkami). Należy rozważyć, czy każdy wierzchołek może być połączony z maksymalnie 4 czy 8 innymi. Połączenia z 8 wierzchołkami pozwala poruszać się po planszy na ukos. Dodaje to algorytmowi realizmu, ale również zwiększa liczbę obliczeń. W celu uproszczenia algorytmów generujących planszę w niniejszej pracy nie ma połączeń ukośnych.

Stworzony graf jest bardzo specyficzny, ponieważ każdy wierzchołek może być połączony jedynie z sąsiednimi wierzchołkami na siatce. W wygenerowanych zapisach grafu można zauważyć wiele prawidłowości. Plansza gry komputerowej będzie zawierać bardzo dużą liczbę wierzchołków, liczba krawędzi będzie zdecydowanie większa, ale nie może przekroczyć czterokrotności liczby wierzchołków.

Często grafy określa się mianem gęstych lub rzadkich. Daje to informację na temat stosunku liczby krawędzi do liczby wierzchołków. Grafy gęste to takie, gdzie liczba krawędzi jest większa niż  $V^2 / 2$ , oznacza to, że średnio każdy wierzchołek łączy się przynajmniej z połową pozostałych. Graf rozważany w niniejszej pracy jest zdecydowanie grafem rzadkim, a największa możliwa liczba krawędzi na pewno nie przekroczy wartości  $2V$ .

Graf wykorzystywany w programie jest grafem **nieskierowanym**. Stosowanie grafów skierowanych ma uzasadnienie w wyszukiwaniu najkrótszej ścieżki w grach komputerowych. Można wyobrazić sobie sytuację, gdy droga z danego miejsca wiedzie tylko w jedną stronę, przykładem może być skok z urwiska. W takim przypadku można utworzyć graf skierowany, by uwzględnić, iż w jedną stronę można się poruszać a w drugą jest to niemożliwe. Takie sytuacje jednak nie występują w grach strategicznych, lub są wyjątkami.

Graf może być ważony. Oznacza to, iż koszt przejścia dwiema różnymi krawędziami może być różny. Taka sytuacja w grach strategicznych ma częste uzasadnienie. Twórcy gier wzbogacają rozgrywkę o dodatkowy czynnik utrudniający poruszanie po danym terenie. Terenem takim mogą być bagna czy pustynia. Wagi krawędzi na takim terenie są większe.

Wagi grafu nie mogą być ujemne. W teorii grafów istnieje pojęcie ujemnej wagi, jednak w praktycznym wykorzystaniu grafów do wyszukiwania drogi na płaszczyźnie ujemne wagi kompletnie nie mają sensu.



## 5.2 Metody zapisu grafu

### 5.2.1. Macierz sąsiedztwa

Macierz sąsiedztwa [2] jest jednym ze sposobów reprezentacji grafów. Polega on na tym, że dla każdego wierzchołka grafu przechowywana jest kolumna oraz wiersz w macierzy. Zapisana tam jest relacja ze wszystkimi pozostałymi wierzchołkami w postaci wagi krawędzi łączącej dwa wierzchołki. Należy zbudować macierz o rozmiarach  $V \times V$ , złożoność pamięciowa takiego rozwiązania wynosi  $O(V^2)$ . Biorąc pod uwagę, że w przypadku płaszczyzny pojedynczy wierzchołek może być połączony jedynie z sąsiednimi wierzchołkami, rozwiązanie to wydaje się od początku niewydajne. Macierz taką wypełniamy zerami i jedynkami lub wagami krawędzi. Zero oznacza brak połączenia między wierzchołkami, jedynka oznacza istnienie krawędzi między wierzchołkami, inna liczba oznacza istnienie krawędzi między wierzchołkami o wadze równej danej liczbie. Czasami specjalnie dla wykorzystania w algorytmach wyszukujących najkrótszą drogę, zamiast zer macierz uzupełnia się jakąś bardzo dużą liczbą oznaczającą brak ścieżki. Ułatwia to późniejsze obliczenia.

	1	2	3	4	5
1	0	1	0	0	0
2	1	0	1	0	0
3	0	1	0	1	0
4	0	0	1	0	1
5	0	0	0	1	0

Rys. 5.4 Macierz sąsiedztwa dla grafu z rysunku 5.3.

Jak widać skonstruowana macierz jest symetryczna, ponieważ graf jest nieskierowany. Przy pomocy macierzy można zapisać również skierowane grafy, wtedy macierz jest niesymetryczna. Mając graf nieskierowany, niestety nie można zarezerwować mniejszej tablicy. Można jednak skrócić proces generowania macierzy poprzez wypełnianie tylko połowy macierzy, tworząc figurę w kształcie trójkąta.

Taki sposób zapisu grafu ma zdecydowanie najlepszy czas odczytu spośród wszystkich metod, ale jego generowanie jest czasochłonne i zajmuje dużo pamięci. Nadaje się dobrze do małych grafów, które trzeba szybko analizować. Dodatkowo taki graf można bardzo szybko modyfikować.

### 5.2.2. Macierz incydencji

Macierz incydencji [2] jest macierzą o rozmiarach  $V * E$ . Złożoność pamięciowa tego rozwiązania to  $O(V * E)$ . Kolumnami są wierzchołki grafu, a wierszami krawędzie. Wartości znajdujące się w macierzy to -1, 0 oraz 1. Wierzchołek, który jest początkiem krawędzi ma przypisaną wartość -1, a wierzchołek końcowy wartość 1. Zera przypisane są wierzchołkom, które nie są częścią danej krawędzi. Główną zaletą macierzy incydencji jest fakt, że pozwala na przejrzyste opisanie grafu skierowanego. Dla grafu nieskierowanego, używanego w aplikacji, sposób ten odpowiada zapisowi przy pomocy listy sąsiedztwa, zużywając dodatkowo więcej cennych zasobów. Z tego powodu zapis ten zostanie pominięty w badaniach.

	1	2	3	4	5
1 - 2	-1	1	0	0	0
2-3	0	-1	1	0	0
3-4	0	0	-1	1	0
4-5	0	0	0	-1	1

Rys. 5.5 Macierz incydencji dla grafu z rysunku 5.3.

### 5.2.3. Lista incydencji

Lista incydencji [2] jest strukturą złożoną z jednowymiarowej tablicy wierzchołków, w której każdy element zawiera listę wierzchołków, z którymi jest połączony. W tym przypadku zapisanie wag jest bardziej skomplikowane i wymaga przetrzymywania innego typu obiektów. Złożoność pamięciowa listy sąsiedztwa wynosi  $O(V+E)$ .

1	2	
2	1	3
3	2	4
4	3	5
5	4	

Rys. 5.6 Lista incydencji dla grafu z rysunku 5.3.

#### 5.2.4. Lista sąsiedztwa

Lista sąsiedztwa [2] zawiera zbiór krawędzi grafu, nie są one uporządkowane względem wierzchołków. Mogą być posortowane według wierzchołków lub wag. Ten sposób zapisu ma ogromną wadę w stosunku do listy incydencji - jest nieuporządkowany. Przeszukiwanie takiej listy zajmuje dużo czasu. A posortować można ją jedynie po jednej z dwóch wartości, co nie daje dobrych efektów. Zapis przy pomocy listy incydencji jest znacznie lepszą, choć bardzo podobną formą zapisu. Z tego powodu lista sąsiedztwa zostanie pominięta w badaniach. Złożoność pamięciowa tego rozwiązania wynosi  $O(E)$ . Rozwiązanie to jest mało wydajne, nie tylko dla problemu zapisu planszy gry. Jest ono mało wydajne ogólnie, ponieważ przeszukiwanie takiego nieuporządkowanego zbioru wymaga zbyt dużo czasu.

1	2
5	4
2	1
3	2
3	2
4	3
4	3
5	4

Rys. 5.7 Lista sąsiedztwa dla grafu z rysunku 5.3.

### 5.3. Algorytmy wyszukiwania drogi

#### 5.3.1. Algorytm Dijkstry

Algorytm Dijkstry [3] jest jednym z algorytmów wyszukujących najkrótszą ścieżkę w grafie, będąc jednocześnie najpopularniejszym z nich. Został opracowany w roku 1959 przez Edsgera Dijkstrę. Jego działanie polega na wyznaczeniu odległości rozumianej, jako suma wag krawędzi z jednego wierzchołka grafu do wszystkich pozostałych. Jednak algorytm można w prosty sposób przerwać, co pozwoli na wyszukanie odległości do konkretnego wierzchołka bez nadmiarowych obliczeń. W uproszczeniu algorytm polega na podziale zbioru wierzchołków na dwie grupy:

- Grupa wierzchołków z nieznaną odległością od punktu początkowego (**Q**)
- Grupa wierzchołków, u których już obliczono odległość od punktu początkowego (**S**)

```
Każdemu wierzchołkowi v znajdującemu się w zbiorze wszystkich wierzchołków ustaw:
{
    d[v] = MAX
    poprzednik[v] = null
}
d[start] = 0
Dodaj wszystkie wierzchołki do zbioru Q
dopóki Q zawiera wierzchołki wykonuj:
{
    u := Weź wierzchołek o najmniejszej drodze d[] ze zbioru wszystkich wierzchołków
    Jeśli d[u] = MAX
    {
        Zakończ algorytm - brak rozwiązań
    }
    Jeśli d = koniec
    {
        Zakończ algorytm - d[koniec] zawiera rozwiązanie
    }
    dla każdego wierzchołka v w zbiorze Q
    {
        jeżeli d[v] > d[u] + w(u, v) to
        {
            d[v] = d[u] + w(u, v)
            poprzednik[v] = u
        }
    }
}
```

Rys. 5.8 Algorytm Dijkstry w pseudokodzie.

Początkowo algorytm zawierał również trzecią grupę, zbiór wszystkich wierzchołków, jednak praktyka pokazała, iż na początku wywołania algorytmu można od razu całą zawartość tej grupy przenieść do zbioru **Q**.

Algorytm w uproszczeniu polega na cyklicznym wykonywaniu następujących kroków:

- Wybranie ze zbioru **Q** wierzchołka o znanej najmniejszej odległości do wierzchołka startowego.
- Dodanie wybranego wierzchołka do zbioru **S**.
- Przegląd wierzchołków sąsiadujących z wybranym i sprawdzenie czy nie znajdują się bliżej startowego.

Po wielokrotnym cyklicznym wywoływaniu zbiór **Q** stanie się pusty, a wszystkie wierzchołki zostaną przeniesione do zbioru **S**.

Metoda **relaksacji krawędzi** polega na sprawdzeniu, czy dana krawędź  $(s, v)$  nie prowadzi do wierzchołka  $v$  krótszą drogą, niż krawędź  $(u, v)$  aktualnie uważana za właściwą. Jeśli tak, najkrótsza znana odległość jest zmniejszana. Działanie takie wykonuje się dla wszystkich sąsiadujących z wierzchołkiem krawędzi. Właśnie na tej metodzie opiera się algorytm Dijkstry.

Należy zauważyć, iż podstawowa wersja algorytmu potrafi znaleźć odległość między dwoma wierzchołkami, ale nie zwraca drogi. W celu poznania drogi należy dodać do algorytmu zapamiętywanie poprzedniego wierzchołka oraz odtworzyć drogę. W przypadku algorytmu Dijkstry, polega to na odczytaniu kolejnych wierzchołków oznaczonych, jako poprzednie zaczynając od wierzchołka będącego celem.

Złożoność obliczeniowa algorytmu Dijkstry zależy od sposobu implementacji kolejki i może wynosić:

- $O(V^2)$  w przypadku oparcia implementacji o tabelę zamiast kolejki
- $O(E * \log(V))$  w przypadku implementacji opartej na kolejce
- $O(E + V * \log(V))$  w przypadku implementacji opartej na kopcu Fibonacciego

### 5.3.2. Algorytm Forda-Bellmana

Algorytm Forda-Bellmana [3] podobnie jak algorytm Dijkstry pozwala na znalezienie odległości pomiędzy dwoma wierzchołkami w grafie i również opiera się na metodzie **relaksacji krawędzi**. Jego ogromną zaletą w stosunku do algorytmu Dijkstry jest fakt, iż wagi w analizowanym grafie mogą być ujemne. Złożoność obliczeniowa tego algorytmu wynosi  $O(V^3)$ . W rozpatrywanym w niniejszej pracy grafie złożoność silnie zależna od liczby wierzchołków jest bardzo niekorzystna, a fakt możliwości rozpatrywania ujemnych wag nie jest istotny. W przypadku wykorzystania zapisu grafu opartego o krawędzie jak na przykład lista sąsiedztwa można uzyskać złożoność uzależnioną od krawędzi wynoszącą  $O(V * E)$ , co może być znacznie korzystniejsze dla niektórych przypadków.

```

Każdemu wierzchołkowi v znajdującemu się w zbiorze wszystkich wierzchołków ustaw:
{
    d[v] = MAX
    poprzednik[v] = null
}
d[start] = 0
wykonuj tyle razy ile jest wierzchołków:
{
    dla każdego wierzchołka u wykonuj:
    {
        dla każdego wierzchołka v wykonuj:
        {
            jeżeli d[v] > d[u] + w(u, v) to
            {
                d[v] = d[u] + w(u, v)
                poprzednik[v] = u
            }
        }
    }
}

```

Rys. 5.9 Algorytm Forda-Bellmana w pseudokodzie.

### 5.3.3. Algorytm A\*

Algorytm A\* [11] został stworzony przez Petera Harta w 1968 roku. Jest to ogólny heurystyczny algorytm szukający drogi w przestrzeni stanów. Potrafi między innymi wyszukać najkrótszą drogę między dwoma punktami na płaszczyźnie, ale nadaje się również do przeszukiwania drzew gier oraz rozwiązywania innych problemów. Idea działania algorytmu opiera się na przeszukiwaniu stanów sąsiednich w stosunku do stanów już znanych. Przeszukiwane są w określonej kolejności według wybranego kryterium. W przypadku niniejszej pracy stanem nazywa się wierzchołek grafu.

Algorytm tworzy dwa zbiory, w których umieszcza już odwiedzone wierzchołki. Jeden z tych zbiorów, nazywany **Closed** przechowuje wierzchołki już sprawdzone, które poza wyjątkami nie będą już odwiedzane. Drugi zbiór, nazywany **Opened** przechowuje wierzchołki, które zostaną wkrótce odwiedzane. Początkowo zawiera jedynie wierzchołek startowy. Ze zbioru wybierany jest wierzchołek o najlepszej ocenie. Sposób definiowania oceny może być różny w zależności od przeznaczenia algorytmu. W kolejnym kroku analizowane są wierzchołki sąsiednie do wybranego. Mogą one być dodane do zbioru **Opened** lub usunięte z **Closed**, jeśli się tam znajdowały. Po tych działaniach analizowany wierzchołek dodawany jest do zbioru już przeszukanych i rozpoczyna się kolejna iteracja algorytmu.

Ocena wierzchołka jest ważnym elementem implementacji konkretnej wersji algorytmu A\*, ocena ta nie zależy od samej idei algorytmu a od wykorzystania, dlatego należy opracować ją samodzielnie. W przypadku problemu wyszukiwania drogi na płaszczyźnie ustalenie oceny jest proste. Jako koszt drogi od startu (wierzchołek **S**) do aktualnego wierzchołka (wierzchołek **C**) można przyjąć wartość bezwzględną z różnicy wektorów określających te dwa punkty. Podobnie określony został koszt od wierzchołka do celu (wierzchołek **E**). Łączny koszt drogi jest sumą kosztu od startu i kosztu do celu.

$$S = \overrightarrow{(x_s, y_s)}$$

$$E = \overrightarrow{(x_e, y_e)}$$

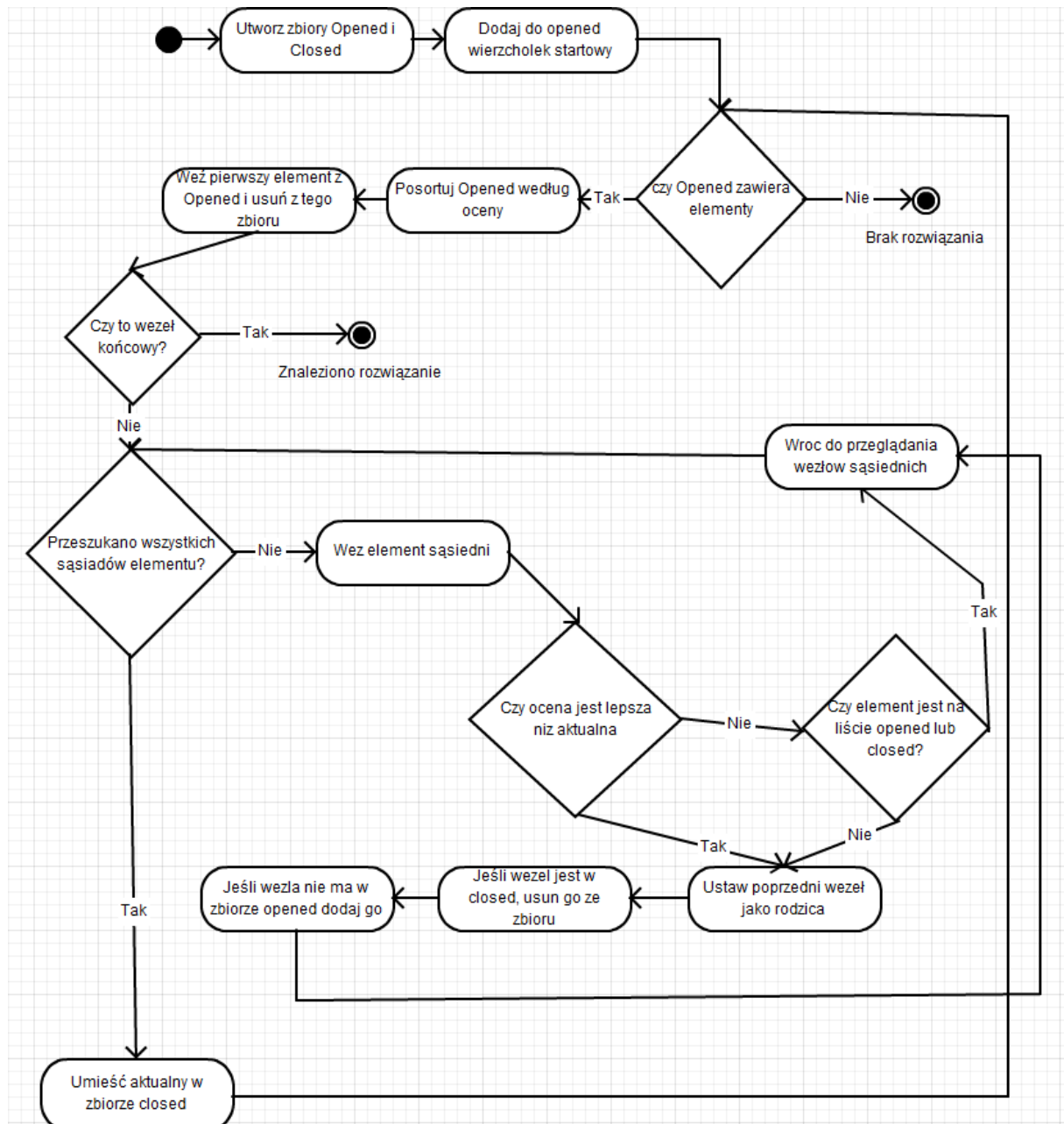
$$C = \overrightarrow{(x_c, y_c)}$$

$$\text{Koszt do celu} = |C - E| = \left| \overrightarrow{(x_c, y_c)} - \overrightarrow{(x_e, y_e)} \right|$$

$$\text{Koszt od startu} = |C - S| = \left| \overrightarrow{(x_c, y_c)} - \overrightarrow{(x_s, y_s)} \right|$$

$$\text{Łączny koszt} = \text{Koszt do celu} + \text{Koszt od startu}$$

Koszt ten może być wyliczany w inny sposób. Gdyby rozważać plansze, na której poruszanie się jest zależne od rodzaju terenu, co odpowiada wagom wierzchołków, wtedy również funkcję celu należałoby uzależnić od wag. Zła implementacja takiego rozwiązania wydłużyłaby jednak czas działania algorytmu, dlatego należałoby wtedy wagę od startu obliczać dodając do kosztu poprzedniego stanu koszt przejścia do aktualnego stanu. Koszt do celu powinien być obliczany bez uwzględnienia wag, ponieważ wagi na nieznaną jeszcze drogę również nie są znane.



Rys. 5.10 Diagram aktywności opisujący algorytm A\*.

W celu dokładnego przeanalizowania algorytmu została stworzona graficzna reprezentacja odwiedzonych pól. Na rysunkach (Rys. 5.11, Rys. 5.12) widać wynik obliczeń, którym jest odnaleziona najkrótsza ścieżka, a także wszystkie przeanalizowane przez algorytm pola. Najkrótsza ścieżka oznaczona jest białymi, a analizowane pola żółtymi punktami. Rysunek (Rys. 5.12) wyraźnie pokazuje, iż algorytm wykorzystuje kierunek na płaszczyźnie. Pola nieznajdujące się w kierunku celu są pomijane, za wyjątkiem pól sąsiednich do tych znajdujących się w wynikowej ścieżce. Na rysunku (Rys. 5.11) widać, że w przypadku znacznych przeszkód w drodze do celu algorytm przeszukuje wszystkie kierunki opierając się na odległości od analizowanego punktu do celu.





Rys. 5.11 Wynik działania algorytmu A\* wraz z analizowanymi polami.



Rys. 5.12 Wynik działania algorytmu A\* wraz z analizowanymi polami.



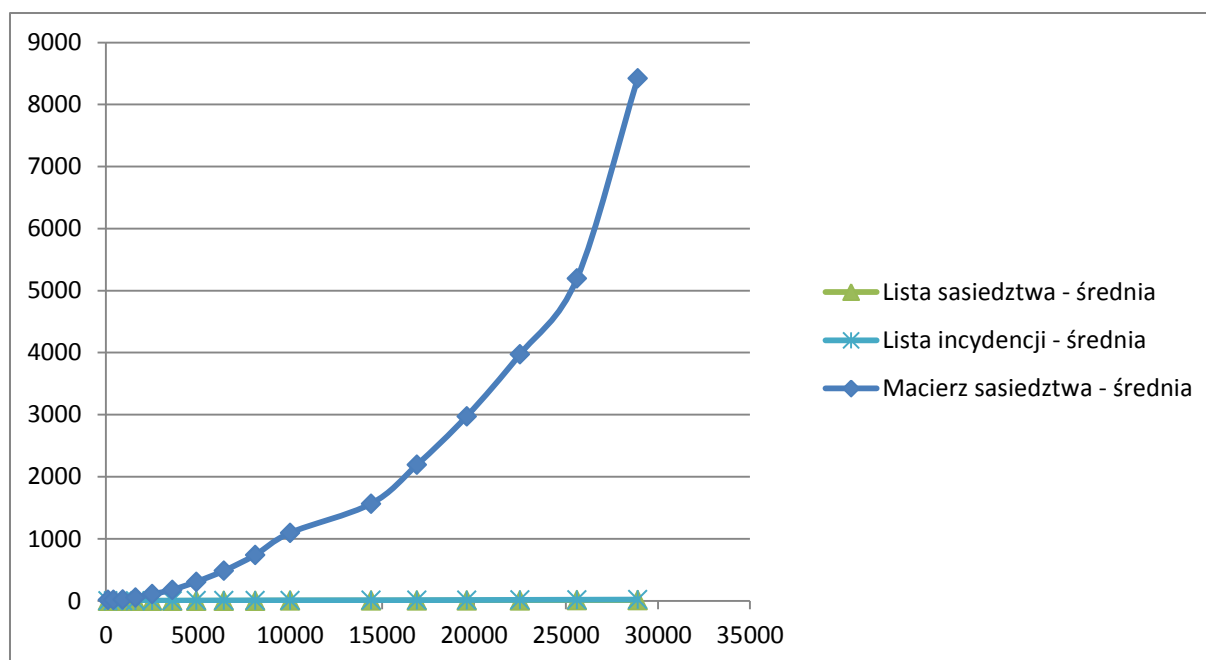
### 5.3.4. Porównanie algorytmów

Pierwsze porównanie dotyczyć będzie samych metod generowania grafów. Metody te będą ważne jedynie w przypadku algorytmów Dijkstry i Forda-Bellmana. Zatem, jeśli najskuteczniejszym algorytmem wyszukiwania drogi okaże się algorytm A\*, algorytmy te nie będą do niczego potrzebne. W celu zbadania tych algorytmów zostały wykonane pomiary czasu ich trwania. Pierwsze porównanie dotyczy zależności czasu ich wykonywania od wielkości generowanego grafu. Wielkość grafu zadawana jest, jako długość krawędzi kwadratowej płaszczyzny, zatem faktyczna liczba wierzchołków będzie wzrastać wykładniczo na osi rzędnych na wykresach. Ważny w tym przypadku jest fakt, iż generowanie takiego grafu należy wykonać raz, na samym początku rozgrywki. Następnie graf ten można modyfikować podczas rozgrywki. Skutkiem takiego założenia jest mały wpływ nieznacznych różnic prędkości na efekt końcowy. Należy jednak wyeliminować przypadki, gdy różnice prędkości będą zdecydowanie zbyt duże.

Pomiary zostały wykonane przy pomocy klasy **System.Diagnostics.Stopwatch**, a mierzoną jednostką czasu są milisekundy. Doświadczenie zostało powtórzone kilkakrotnie, a wynik jest średnią arytmetyczną wszystkich pomiarów. Graf był tworzony z planszy, na której losowo zapełniono 10% pól.

Długość boku planszy	Liczba wierzchołków grafu	Lista sąsiedztwa średni czas[ms]	Macierz sąsiedztwa średni czas[ms]	Lista incydencji średni czas[ms]
10	100	0,4	14,4	0,4
20	400	0	15,6	0
30	900	0	23	0
40	1600	0	54,2	1
50	2500	1	108,2	2
60	3600	1	178,6	2,8
70	4900	2	306,4	4
80	6400	2,2	483,6	5
90	8100	3	737,8	6,2
100	10000	4	1095,4	8
120	14400	5	1564,8	9,4
130	16900	6	2191,6	11
140	19600	7	2974,2	13
150	22500	8,2	3974	15,6
160	25600	9,2	5197	18
170	28900	10,4	-	20
200	40000	23,2	-	33,2
300	90000	44,8	-	85,8
400	160000	83	-	170,6
500	250000	139,2	-	291
600	360000	233,8	-	442,8
700	490000	274,2	-	495,6
800	640000	356,2	-	689
900	810000	442,8	-	829,2

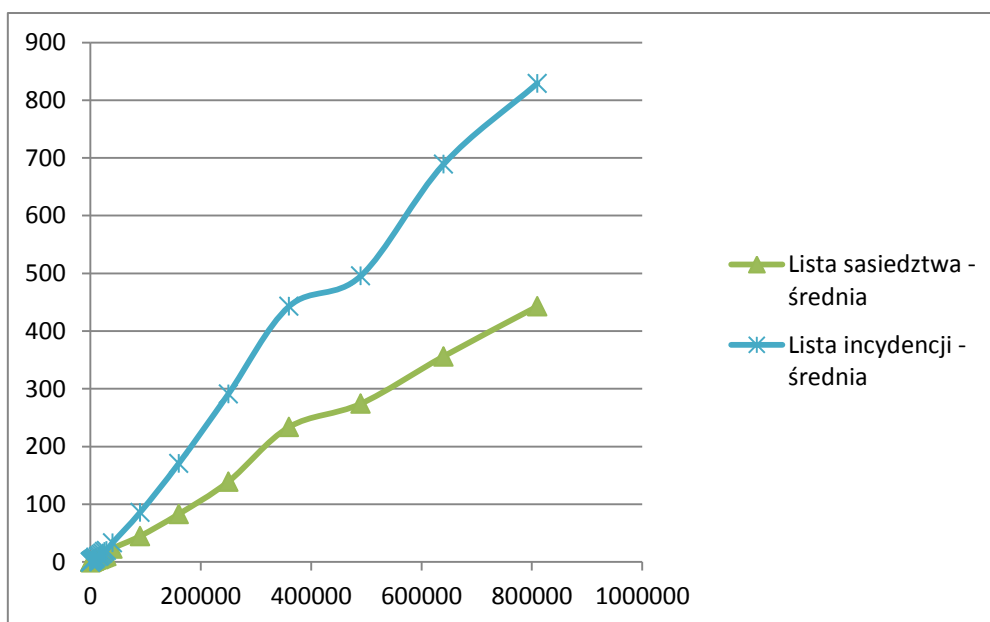
Rys. 5.13 Tabela [ms] czas tworzenia grafu w zależności od liczby wierzchołków.



Rys. 5.14 Wykres czasu [ms] tworzenia grafu w zależności od liczby wierzchołków.

Wykres (Rys. 5.14) wyraźnie pokazuje, że tworzenie macierzy sąsiedztwa trwa bardzo długo w porównaniu do pozostałych metod i co najważniejsze czas trwania rośnie wykładniczo. Jest to ogromną wadą tej metody, ponieważ graf potrzebny w aplikacji będzie miał znaczną liczbę wierzchołków. Maksymalna liczba wierzchołków obsługiwana w aktualnej wersji to 2 560 000 osiągana przy planszy o rozmiarach 1600:1600. Algorytm działa dość długo już dla 30 000 wierzchołków i można wywnioskować, iż czas generowania macierzy przy 2 milionach wierzchołków będzie zupełnie nie do przyjęcia. Dodatkowym problemem, jaki się pojawił jest zajmowanie dużej liczby zasobów. Algorytm alokuje pamięć o rozmiarze  $V^2$ . Okazało się, iż środowisko Unity3d nie pozwala na alokację zbyt dużej ilości pamięci. Problem pojawiał się epizodycznie już od alokacji tablicy o wymiarach 200x200. Problem nie polega na samym algorytmie, leży po stronie środowiska lub błędnie napisanego kodu, być może jest związany z obszernymi wyciekami pamięci. Mimo braku związku samego skutku z algorytmem należy ten algorytm odrzucić ze względu na liczne wady. Wszelkie zalety tej metody takie jak możliwość zapisu kierunku czy wagi, w tym ujemnej nie są na tyle ważne.

Wykres (Rys. 5.15) zestawia dwie pozostałe metody zapisu grafu. Jak widać różnica w czasie wykonywania nie jest znaczna i rośnie mniej więcej liniowo. Czas w przypadku obu algorytmów jest zdecydowanie akceptowalny nawet w przypadku miliona wierzchołków. Na pewno będzie również akceptowalny w przypadku maksymalnej liczby wierzchołków wynoszącej około 2,5 miliona wierzchołków. Różnica w czasie wykonywania między obydwoima algorytmami jest dwukrotna, ale rząd tej złożoności wydaje się być taki sam. Z tego powodu nie należy sugerować się jedynie tą cechą. Ważniejsza jest przydatność wygenerowanego grafu, a graf zapisany w postaci listy incydencji jest znacznie prostszy do przeszukiwania. Dodatkową ważną cechą jest jeszcze czas wyszukiwania drogi w tak zapisanym grafie, co zostanie omówione w dalszej części.



Rys. 5.15 Wykres czas [ms] tworzenia grafu w zależności od liczby wierzchołków.

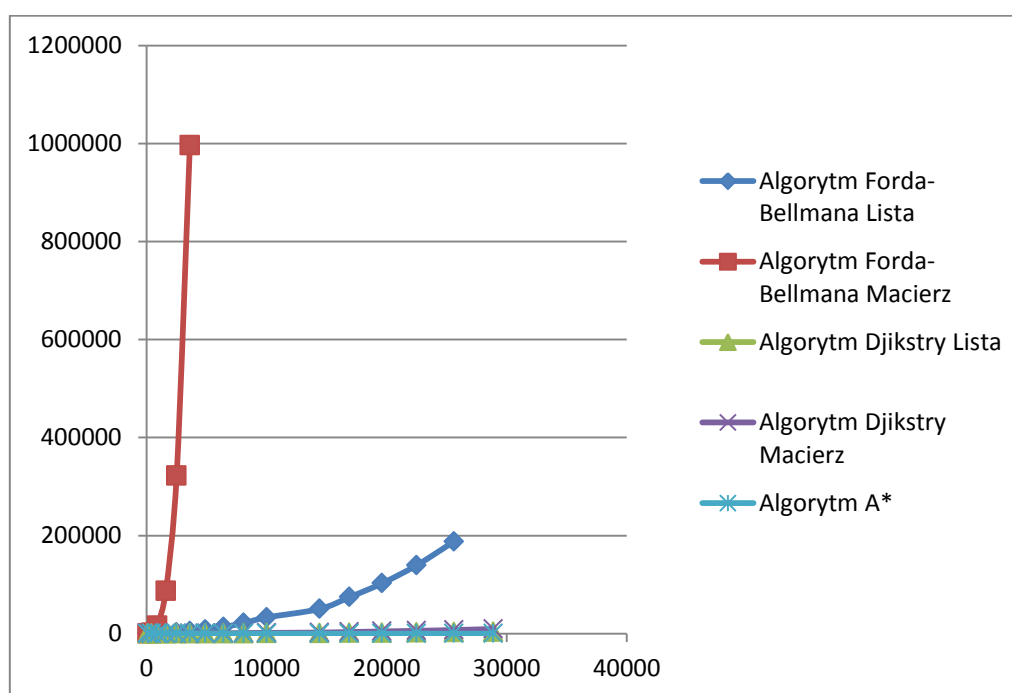
Po zbadaniu algorytmów generujących grafy zostały przetestowane algorytmy wyszukujące najkrótszą ścieżkę. Pomiar został wykonany przy losowym zapelnieniu 10% pól na planszy. Wyszukiwanie drogi polegało na znalezieniu drogi między dwoma punktami o współrzędnych S(2, 2) oraz E(50, 50), zatem odległość między punktami, liczona według zaproponowanej funkcji oceny wynosi 96. Tabela (Rys. 5.16) oraz wykres (Rys. 5.17) przedstawiają zależność czasu wykonywania się algorytmu od liczby wierzchołków. Najwolniej działa algorytm Forda-Bellmana, szczególnie wykorzystując macierz sąsiedztwa, jako reprezentację grafu. Na wykresie (Rys. 5.18) można dokładniej przyjrzeć się pozostałym wynikom. W przypadku algorytmu Dijkstry ponownie zapis grafu, jako macierz sąsiedztwa ma słabszą wydajność od listy incydencji. Zatem ostatecznie zapis w postaci macierzy sąsiedztwa można uznać za najmniej wydajny sposób w analizowanym przypadku.

Najwydajniejszym algorytmem wydaje się być A\*. Algorytm ten działa równie szybko bez względu na rozmiar planszy, co widać na wykresie (Rys. 5.19). Na kształt wykresu wpłynęły głównie błędy pomiaru i elementy losowe jak układ planszy. Czynnikiem, jaki wpływa na prędkość jego działania jest głównie odległość między badanymi punktami. Tabela (Rys. 5.20) oraz wykresy (Rys. 5.21, Rys. 5.22) prezentują zależność czasu wykonywania algorytmu od odległości na planszy między punktami. Wszystkie pomiary zostały wykonane na planszy o rozmiarach 300:300. Pomiary zostały powtórzone 100-krotnie, gdyż ogromny wpływ na wyniki ma losowo wygenerowana plansza, a przy krótkich czasach wykonywania błędy pomiaru będą znaczne.

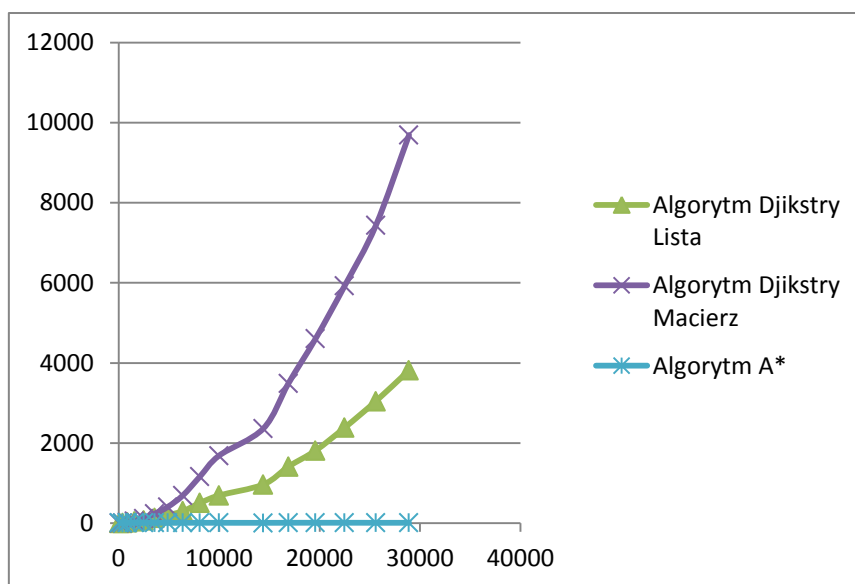
Wyniki pokazują, że odległość punktów na planszy ma wpływ na oba algorytmy. W przypadku algorytmu Dijkstry wpływ ten jest widoczny tylko, gdy rozwiązanie znajduje się blisko początku zbioru rozwiązań. Gdy rozwiązanie nie zostanie tak szybko znalezione algorytm szuka znacznie dłużej. Świadczy to o fakcie, że wyszukiwanie nie opiera się na żadnej heurystyce, tylko przeszukiwany jest cały graf. Na prędkość wykonywania w algorytmie A\* odległość wpływa przy wszystkich pomiarach równomiernie. Dzieje się tak, ponieważ wykorzystywana jest ocena kierunku, w którym podąża algorytm. Czas działania zależy silnie od odległości, nie zaś od wielkości planszy.

Długość boku planszy	Liczba wierzchołków grafu	Algorytm Forda-Bellmana Lista	Algorytm Forda-Bellmana Macierz	Algorytm Dijkstry Lista	Algorytm Dijkstry Macierz	Algorytm A*
10	100	5,8	22,8	8,4	2	9
20	400	59	1603,4	6,4	7,2	9,2
30	900	292,6	16247,6	19,4	18,4	9,6
40	1600	936,8	86932,2	41,2	53,2	9,8
50	2500	2155,4	322460,8	69,8	114,6	9,8
60	3600	4658,6	996555,4	135,8	239,8	9,4
70	4900	8155,4	-	202	408,6	9,8
80	6400	13674	-	300,4	696	9,6
90	8100	21793,2	-	506	1167,4	9,8
100	10000	33124,2	-	692,6	1687,2	9,8
120	14400	50383,8	-	966,4	2363,2	9,6
130	16900	74467,4	-	1412,4	3491,6	10,4
140	19600	102555,2	-	1813	4607	10,4
150	22500	139609,2	-	2385,8	5928,2	10,4
160	25600	187914,6	-	3045,8	7433,8	10,4
170	28900	-	-	3815,8	9689,4	10,6

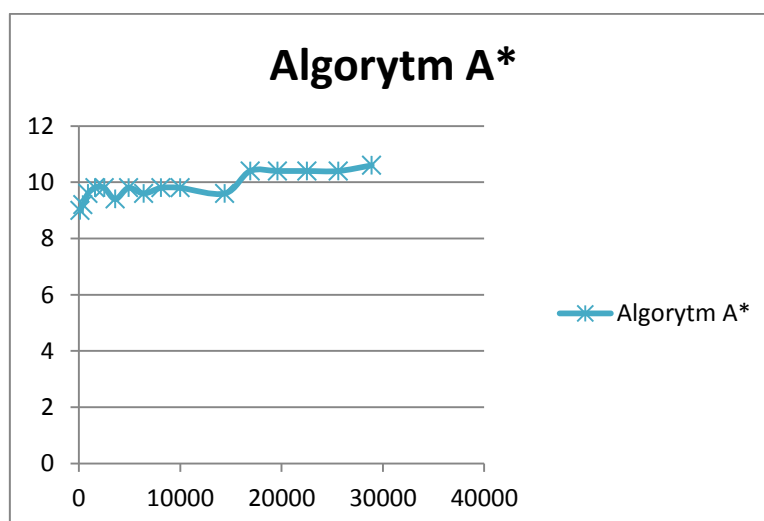
Rys. 5.16 Tabela czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.



Rys. 5.17 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.



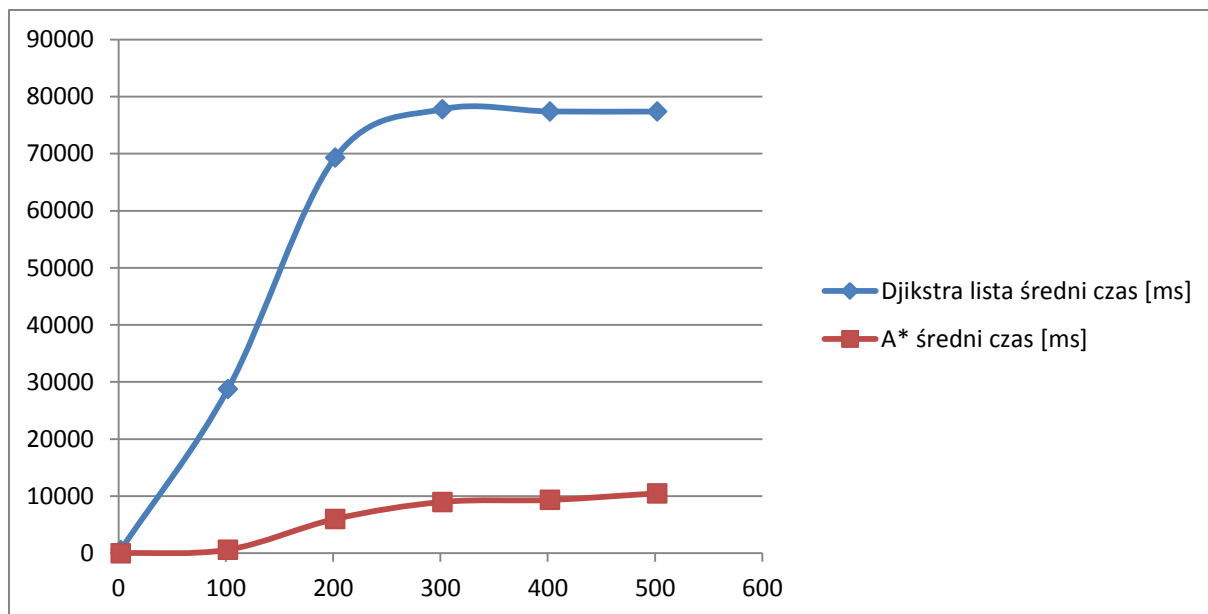
Rys. 5.18 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.



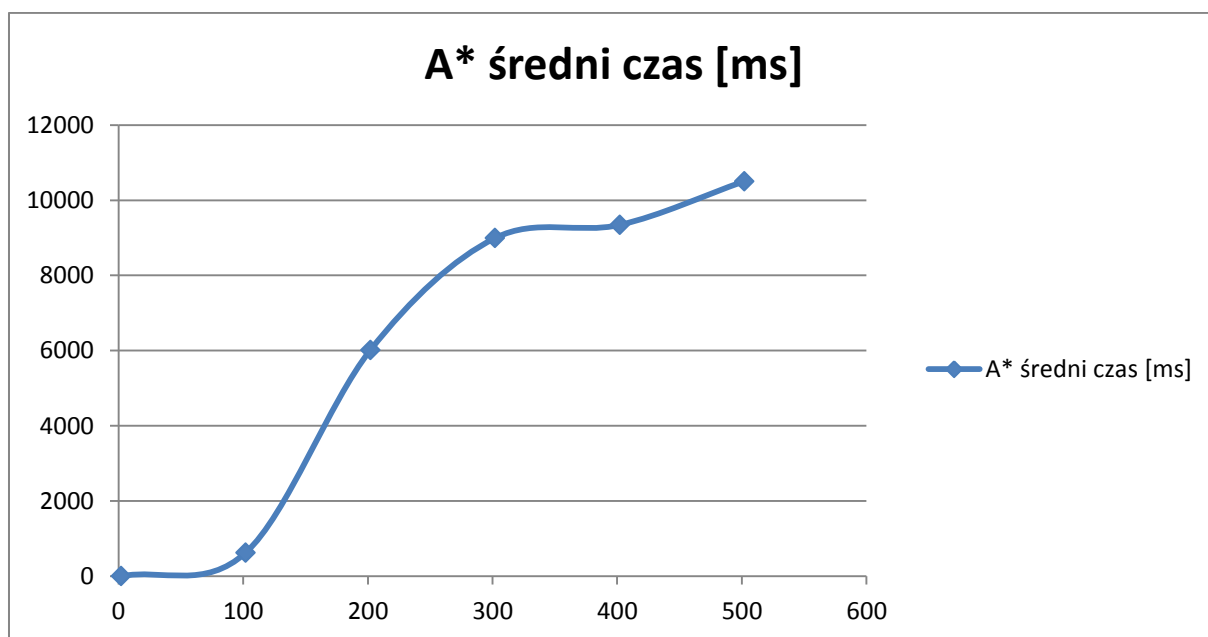
Rys. 5.19 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.

Punkty startowy	Punkty końcowy	Odległość	Dijkstra lista średni czas [ms]	A* średni czas [ms]
(0, 0)	(1, 1)	2	575	0
(0, 0)	(51, 51)	102	28774	626
(0, 0)	(101, 101)	202	69313	6014
(0, 0)	(151, 151)	302	77793	8993
(0, 0)	(201, 201)	402	77392	9343
(0, 0)	(251, 251)	502	77377	10505

Rys. 5.20 Tabela czas [ms] wyszukiwania ścieżki w zależności od odległości na planszy.



Rys. 5.21 Wykres czas [ms] wyszukiwania ścieżki w zależności od odległości na planszy.



Rys. 5.22 Wykres czas [ms] wyszukiwania ścieżki w zależności od odległości na planszy dla algorytmu A\*.

## 5.4. Modyfikacje algorytmu A\*

Przedstawiony **sposób zapisu** w oparciu o prostą siatkę sprawdza się idealnie w grach strategicznych, jednak inne gatunki gier wykorzystują odmienne metody [9] podziału płaszczyzny oraz przekształcania jej w strukturę grafową. Metody te również nadają się na dane wejściowe dla algorytmu A\*, pozwalają na określenie kierunku, a także posiadają wiele zalet ważnych w poszczególnych zagadnieniach różnych gatunków gier.

Najprostszym sposobem reprezentacji planszy jest **siatka** składająca się z kwadratów lub sześciątów. Metoda taka została wykorzystana w niniejszym projekcie, ponieważ idealnie nadaje się do gier strategicznych.

Zbliżoną do siatki metodą jest podział na **drzewa czwórkowe**, w którym płaszczyzna również jest podzielona na kwadratowe pola, jednak pola te są różnej wielkości. Jeśli duży kwadrat zawiera w sobie elementy terenu dzielony jest na cztery kwadraty, a następnie czynność powtarzana jest rekurencyjnie. Dzięki takiemu podejściu można otrzymać podział, którego przeszukiwanie będzie szybsze, ponieważ przy pustych przestrzeniach można ominąć bardzo wiele operacji. Metoda ta również sprawdza się dobrze w grach strategicznych, szczególnie przy dużych pustych obszarach.

Inną ciekawą metodą jest podział planszy ze względu na inne kryterium – **punkty widoczności**. Jest to szczególnie ważne w grach **FPS**, gdzie sztuczna inteligencja sterująca agentami stara się umieścić ich w dobrych taktycznie punktach. Punkty te wyszukiwane są w taki sposób, by widoczność na planszy z tych punktów była możliwie maksymalna. Często agenci wykorzystujący tę metodę nie przemieszczają się bezpośrednio do tych punktów, lecz jedynie zbliżają się, tak by być osłoniętymi przed innymi punktami widoczności. Miejsca te znajdują się zazwyczaj na wierzchołkach figur tworzonych przez przeszkody, a poruszanie się po płaszczyźnie nie sprowadza się jedynie do przemieszczania się między tymi punktami. W grach strategicznych użycie takiego zapisu planszy nie daje korzyści.

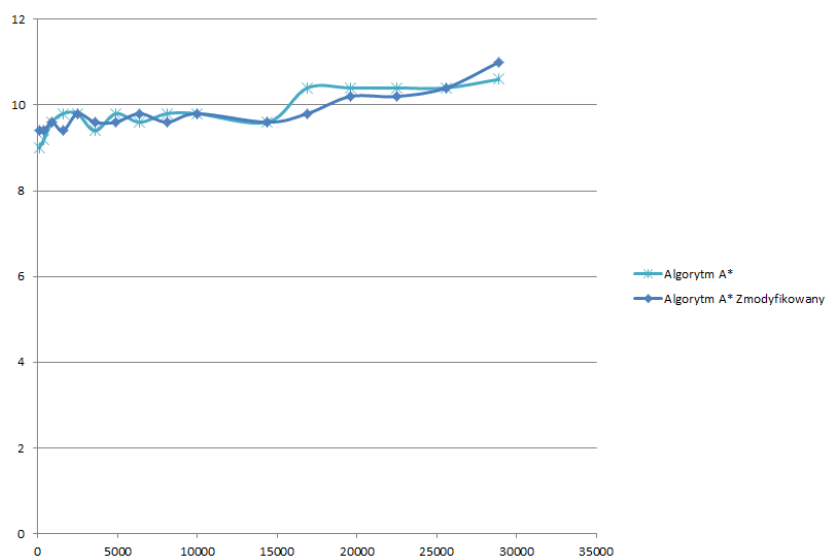
Bardzo ciekawą modyfikacją tego algorytmu jest podział płaszczyzny [9] na części reprezentujące na przykład wyspy czy pomieszczenia. Dzięki takiemu rozwiązaniu pomniejsza się przeszukiwany obszar. Algorytm działa w obrębie jednej części oraz w obrębie całego obszaru operując na częściach, a nie poszczególnych polach. Przykładowo, algorytm osobno przeszuka drogę między pokojami mając wiedzę na temat tego, które pokoje są połączone, a dopiero będąc wewnątrz konkretnego pokoju będzie szukał drogi do kolejnych drzwi. Podejście takie zostało użyte na przykład w grze *Dragon Age (2007)*. W grze tej został zastosowany trzy-poziomowy podział. Plansza jest reprezentowana poprzez siatkę, co jednak nie jest wydajnym sposobem. Z tego powodu siatka podzielona jest na większą, która może być przeszukiwana szybciej. Pola dużej siatki podzielone są na obszary, wyliczone na podstawie przeszkód znajdujących się w polu.

Na potrzeby różnych gier powstało wiele modyfikacji algorytmu A\*. Część z nich jest przeznaczona do konkretnych gatunków gier, w których występują określone zagadnienia. Przykładem może być modyfikacja minimalizująca liczbę skrętów drogi. Można to osiągnąć poprzez dodawanie dodatkowej wagi do drogi przy każdym skręcie. Wbrew temu, co może się wydawać nawet najmniejsza wartość dodana przy skręcie da sporą różnicę w kształcie drogi. Jeśli dodawana waga będzie stosunkowo duża droga może się znacznie zmienić eliminując skróty, co również w niektórych przypadkach może być pożądane. Za stosunkowo dużą wagę można uznać na przykład taką jak waga przejścia między polami lub większą. Czasami do wygładzania drogi stosuje się bardziej zaawansowane algorytmy jak **krzywe sklepane Catmulla-Roma**. Pozwalają one na wyznaczenie gładkiej krzywej przechodzącej przez wyszukane algorytmem A\* punkty. Dobrym rozwiązaniem jest częściowe obliczenie wartości tego algorytmu i umieszczenie ich w kodzie, co pozwoli na znacznie szybsze działanie takiego ulepszenia.

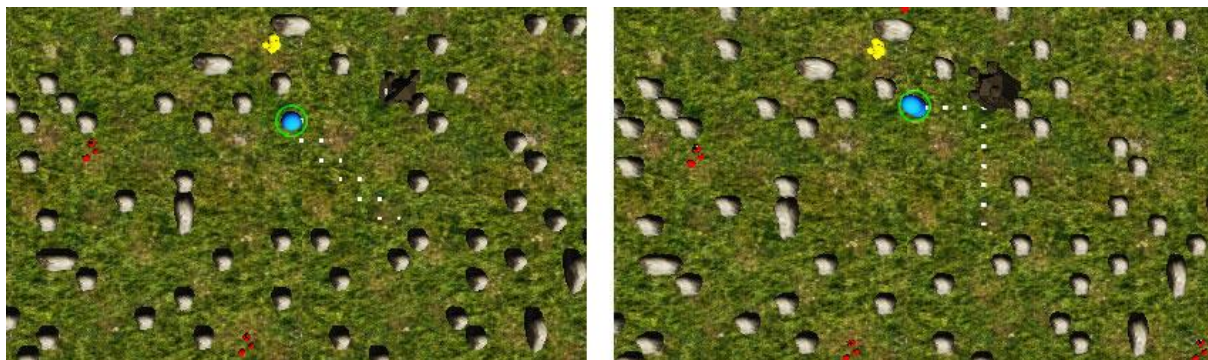
### 5.4.1. Proponowane modyfikacje

Ważnym czynnikiem wpływającym na prędkość wykonywania się algorytmu jest sposób jego implementacji. Algorytm opiera się o dwa zbiory danych **Opened** i **Closed**, które są przeszukiwane i sortowane, warto zadbać by te operacje były możliwie szybkie. Implementacja zawarta w niniejszej pracy opiera się o obiekt **IDictionary** dostępny w **.NET**. Kluczem obiektu jest liczba całkowita jednoznacznie identyfikująca pole. Liczba ta określana jest na podstawie współrzędnych  $x$  i  $y$  pola. Takie rozwiązanie działa znacznie wydajniej, niż implementacja w oparciu o wektor, choć prawdopodobnie wciąż da się zrobić to lepiej.

W programie został zaimplementowany również zmodyfikowany algorytm A\*. Modyfikacja polegała na dodaniu dodatkowej wagi w przypadku skrętu, co poskutkowało wybieraniem bardziej wygładzonych ścieżek, jest to cecha bardzo pożądana w niektórych gatunkach gier. Powstały dwie różne tego typu modyfikacje pierwsza z nich dodaje dodatkową wagę dla przejścia między wierzchołkami, jeśli kierunek się zmienia, druga dodaje wagę za brak zmiany kierunku. Dzięki temu powstały różne sposoby poruszania się po planszy widoczne na rysunku (Rys. 5.24). Sposób uzyskania takiego efektu jest bardzo prosty, należy w tym celu sprawdzić czy kolejne trzy wierzchołki mają równe wartości  $x$  oraz  $y$  określające ich położenie. Obserwacja pokazała, iż najlepsze wyniki daje waga typu zmiennoprzecinkowego tak, by kara za skręcanie była bardzo mała w stosunku do kosztu przejścia między polami. W przeciwnym wypadku może się zdarzyć, że znaleziona droga nie będzie optymalna. Dodatkowo zostało dodane sprawdzenie, czy droga pomiędzy punktami nie jest prostą drogą nie zawierającą przeszkód, co prawdopodobnie przyspieszy działanie algorytmu w specyficznych przypadkach.



Rys. 5.23 Wykres czas [ms] wyszukiwania ścieżki w zależności od liczby wierzchołków.

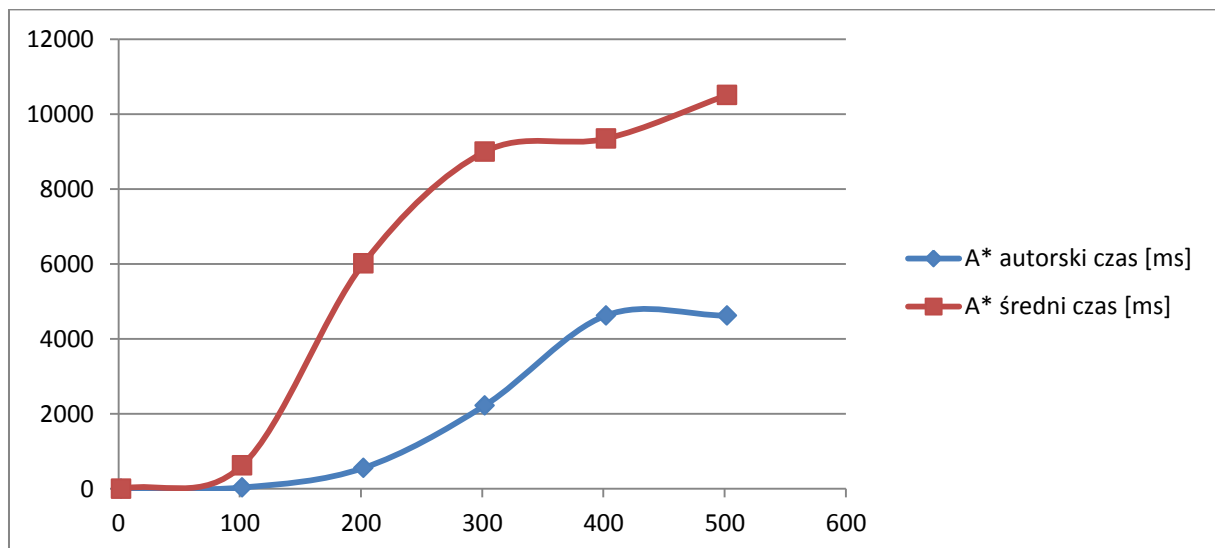


Rys. 5.24 Wynik działania modyfikacji algorytmu A\*.



Rysunek (Rys. 5.23) pokazuje, iż zmodyfikowana wersja algorytmu daje bardzo zbliżone wyniki. Nie jest to radykalna różnica w stosunku do poprzedniej wersji. Co oznacza, że nie udało się przyspieszyć algorytmu wydajnościowo, lub że dla zadanych warunków początkowych nie stanowi to różnicy. Być może w przypadku innej planszy różnica byłaby bardziej znacząca. Została dodana jednak kosmetyczna zmiana dotycząca liczby skrętów w ścieżce, zatem można zastosować ulepszony algorytm w finalnej wersji gry.

W aplikacji stworzonej na potrzeby pracy, co prawda nie została zaimplementowana pełna wersja metody drzew czwórkowych, za to została wprowadzona autorska metoda usprawniająca algorytm A\*. Metoda ta bazuje na podziale całej siatki na kwadraty nie zawierające przeszkód. Podział nie jest rekurencyjny, plansza dzielona jest na równe kwadraty o boku długości 5 pól, zatem taki kwadrat zawiera 25 pól. Po załadowaniu planszy generowana jest nowa tablica zawierająca wygenerowane kwadraty, ponieważ plansza się nie zmienia generacja wykonywana jest tylko raz, na początku gry, zatem czas trwania takiej nie jest problemem. Następnie w algorytmie A\* zamiast sprawdzać zawartość każdego pola sprawdzane jest najpierw czy cały kwadrat nie jest pusty od przeszkód. Jeśli tak, eliminowane są dodatkowe obliczenia, efekt działania modyfikacji pod względem wydajnościowym widać na rysunku (Rys. 5.25). Pomiary dostały wykonane z 50 krotnym powtórzeniem na wylosowanej planszy.



Rys. 5.25 Wykres czas [ms] wyszukiwania ścieżki w zależności od odległości na planszy dla modyfikacji algorytmu A\*.

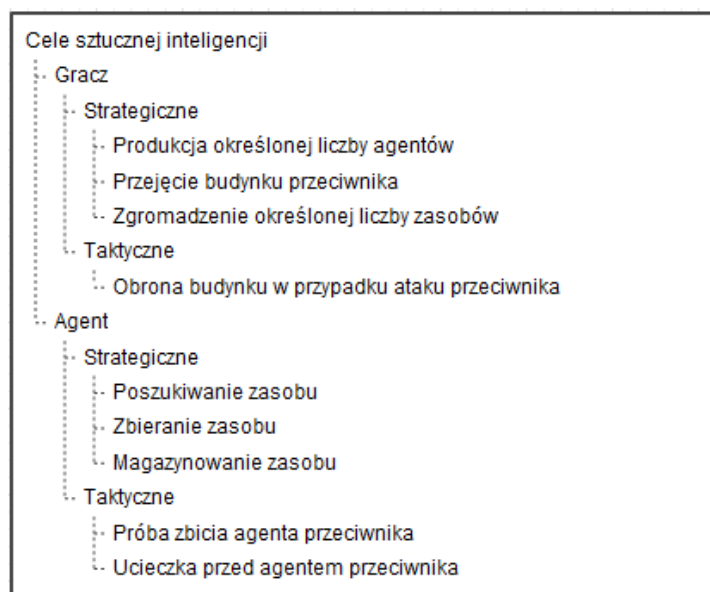
Skuteczność działania takiej modyfikacji silnie zależy od tego jak wygląda wygenerowana plansza, a w szczególności czy jest dużo pustych obszarów, których przeszukiwanie można przyspieszyć. Wielkość kwadratów powinna być zależna od specyfiki planszy i dopasowana do niej. Istnieje wersja algorytmu, która zakłada rekurencyjny podział planszy na takie obszary tworząc drzewo. Jednak w przypadku tak zdefiniowanej planszy, nałożyłoby to zbyt dużo dodatkowych obliczeń i zysk byłby niewielki. Dlatego proponuję rozwiązanie przystosowane do specyfiki planszy. Mechanizm można ulepszyć o wyliczanie optymalnej wartości dla podziału w oparciu o statystyczne dane. Wyliczanie takie mogłoby zająć trochę czasu, jednak wciąż uruchamiane byłoby tylko raz, więc wydajność nie jest problemem.

Niestety zdarzyło się, że algorytm dawał złe wyniki, to znaczy nie zauważał przeszkód. Jest to jednak wynik tylko i wyłącznie jakiegoś błędu w implementacji, ponieważ teoretyczna wersja algorytmu jest spójna i zdaje się być poprawna, a zysk otrzymany z takiej modyfikacji jest widoczny bardzo wyraźnie. Z powodu nie do końca poprawnego działania modyfikacja nie została udostępniona do wyboru dla użytkownika w opcjach aplikacji.

## 5.5. Logika rozmyta i automaty stanów skończonych

Sterowanie sztuczną inteligencją w grze przy pomocy automatów stanów jest stosunkowo prostym zadaniem, wymaga jednak zaplanowania takich automatów, ponieważ od ich budowy zależy zachowanie sztucznej inteligencji. W projektowanej grze sterowanie podzielone jest (Rys. 5.26) na kilka obszarów. Najważniejszy podział działań sztucznej inteligencji to podział według gracza. Każdy gracz posiada osobne sterowanie i nie może ingerować w sterowanie przeciwnika, dodatkowo gracz niebędący człowiekiem jest w pełni sterowany przez automaty stanów, zaś działania człowieka są wspomagane przez automaty stanów w taki sposób, by nie musiał panować nad wszystkimi agentami, upraszczając rozgrywkę manualnie, jednocześnie nie ułatwiając rozgrywki pod względem strategicznym.

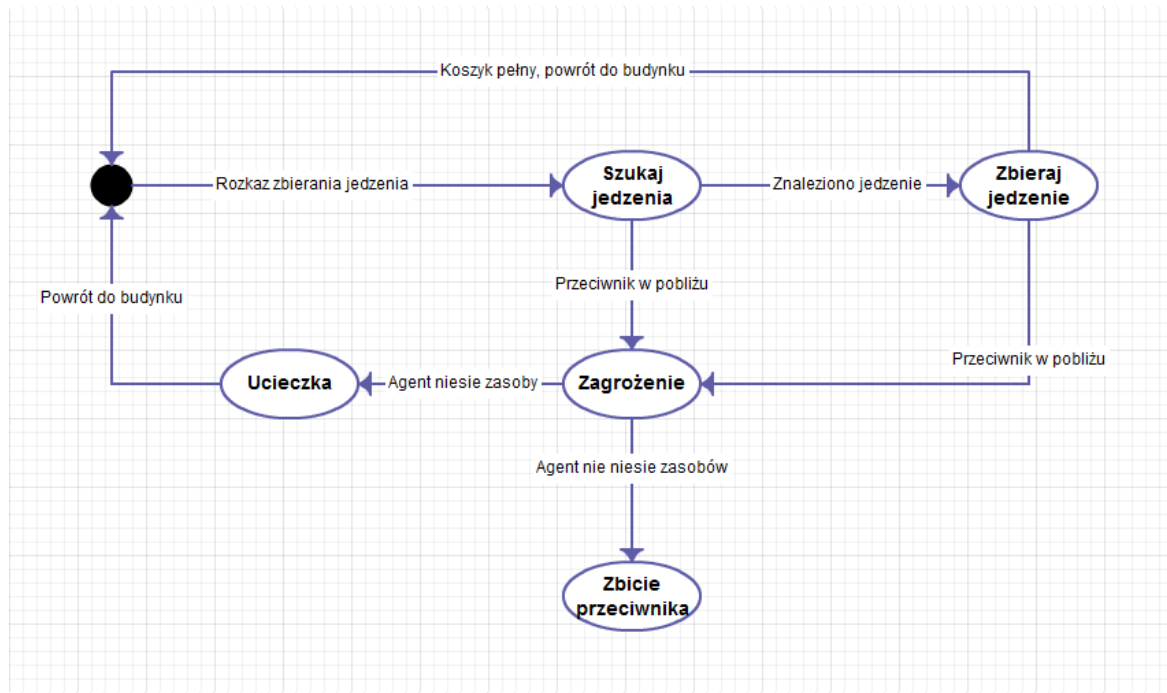
Kolejnym podziałem są osobne obiekty odpowiedzialne za sztuczną inteligencję agenta oraz całego gracza. W przypadku gracza będącego człowiekiem, występują jedynie obiekty sztucznej inteligencji agentów. Ostatnim podziałem jest podział na cele strategiczne oraz taktyczne. W przypadku sztucznej inteligencji całego gracza cele strategiczne to na przykład: wybór, jakie zasoby gromadzić, jak je wydawać, zaś cele taktyczne to przykładowo zebranie wszystkich agentów w pobliżu budynku w celu obrony. W przypadku sztucznej inteligencji sterującej agentem celami strategicznymi są poszukiwanie i zbiór zasobów, zaś celami taktycznymi ucieczka przed wrogimi agentami. Cele taktyczne są wymagane do osiągnięcia natychmiastowo, zatem ich pojawienie się chwilowo wstrzymuje wykonywanie celów strategicznych, a pośrednio prowadzi do ich osiągnięcia lub zabezpiecza aktualną pozycję w rozgrywce.



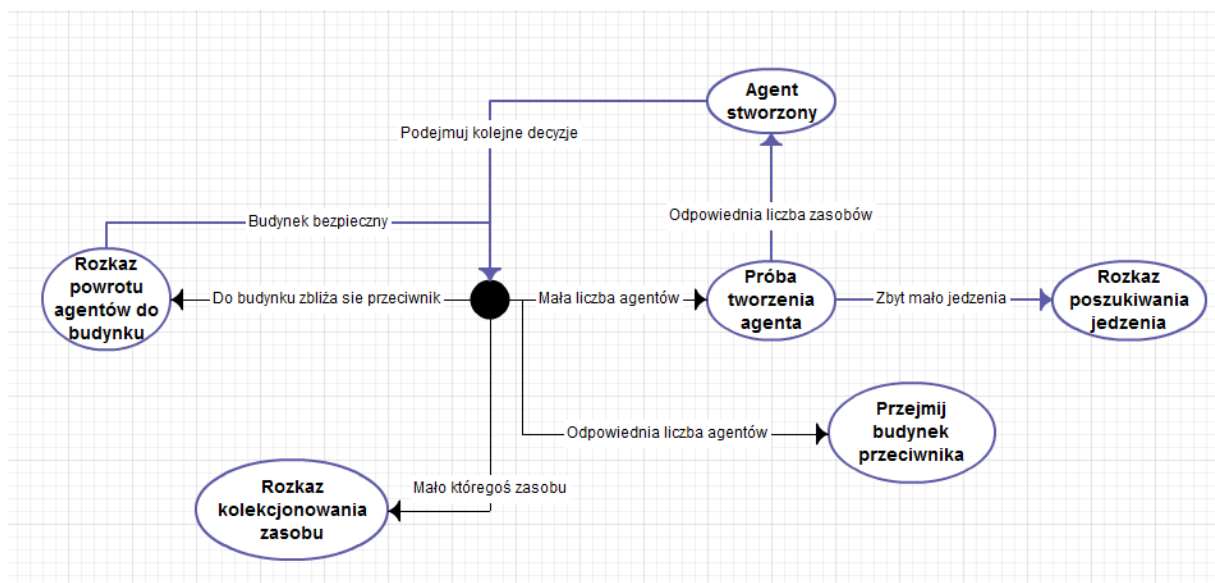
Rys. 5.26 Podział celów sztucznej inteligencji.

Rysunek (Rys. 5.27) pokazuje automat stanów odpowiedzialny za zbieranie zasobu przez pojedynczego agenta. W stanie początkowym agent nie robi nic, dopóki ogólna sztuczna inteligencja lub gracz będący człowiekiem nie rozkaże mu zbierać konkretnego zasobu. Podobne działanie wykonuje agent dla pozostałych zasobów. Na początku zasoby są szukane w najbliższej odległości od agenta, następnie są one zbierane i magazynowane. Czynność ta może zostać przerwana przez spotkanie z obcymi agentami. Wtedy agent próbuje uniknąć spotkania lub zbić agentów przeciwnika.

Rysunek (Rys. 5.28) pokazuje automat stanów gracza sterowanego w pełni przez sztuczną inteligencję. Widać, że na podstawie kilku wskaźników podejmowane są decyzje o aktualnych celach poszczególnych agentów. Schemat zakłada zarówno rozwój ekonomiczny, obronę jak i atak w odpowiednim momencie. Wskaźniki decydujące o podejmowanej akcji to stopień zagrożenia budynku, liczba agentów pozwalająca na osiągnięcie zwycięstwa oraz aktualna ilość zmagazynowanych zasobów.



Rys. 5.27 Automat stanów agenta odpowiedzialny za zbieranie zasobu.



Rys. 5.28 Automat stanów gracza odpowiedzialny strategię gry.

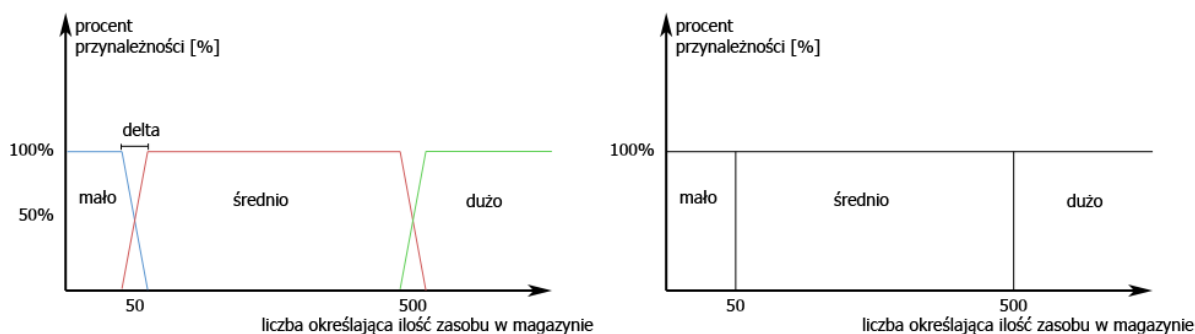
Automaty stanów wykorzystywane w programie używają wskaźników, które definiowane są na podstawie logiki rozmytej. Na początek opisany zostanie mechanizm używający zwykłej logiki. Przykładowym wskaźnikiem jest stan zasobu w magazynie, który może być niski, średni lub wysoki. Stan niski nie pozwala na użycie go i sygnalizuje, iż natychmiastowo trzeba zebrać więcej. Stan średni to stan, który pozwala na wykorzystanie zasobu. Stan wysoki to taki, który sygnalizuje nadmiar i można zaprzestać magazynowanie tego zasobu. Aktualny stan zasobu w programie jest reprezentowany oczywiście za pomocą liczby. Na podstawie tej liczby przypisywany jest stan. Przykładowo w przypadku zasobu *jedzenie* stan niski zawiera się w przedziale  $<0,50)$ , stan średni w przedziale  $<50, 500)$ , a stan wysoki w przedziale  $<500, \infty)$ . Dla pozostałych zasobów zakresy te są inne, ponieważ zapotrzebowanie zasobu również jest inne.

Kolejnym wskaźnikiem wymagającym użycia logiki rozmytej jest liczba agentów, gdzie sprawa jest bardziej skomplikowana. Mała liczba agentów wymagająca natychmiastowego zwiększenia zawiera się w przedziale  $(0, x>$ . Liczbie  $x$  przypisana jest większa z liczb 10 i liczby agentów przeciwnika. Zatem wraz z przyrostem liczby agentów przeciwnika wzrasta  $x$ , dzięki temu automat zawsze będzie się starał mieć więcej agentów niż przeciwnik lub przynajmniej 10 agentów. Natomiast stan dużej liczby agentów zachodzi wtedy, gdy ich liczba przekroczy sumę liczby 10 i liczby agentów przeciwnika. Jest to liczba dobra do ataku na budynek przeciwnika.

```
public poziom GoldLevel
{
    get
    {
        if (gold < 50) _goldLevel = poziom.Malo;
        else if (gold < 500) _goldLevel = poziom.Srednio;
        else _goldLevel = poziom.Duzo;
        return _goldLevel;
    }
}
```

Rys. 5.29 Opis stanu zasobu w magazynie z wykorzystaniem standardowej logiki.

Opisany powyżej (Rys. 5.28) podział wskaźników jest z punktu widzenia gracza sztuczny. Podczas rozgrywki będzie można zauważyć, że zmiana poziomu zasobu w magazynie o 1, diametralnie zmienia zachowanie sztucznej inteligencji. Z tego powodu należy skorygować mechanizm o logikę rozmytą. Korekcja ta polega na płynnym przejściu między stanem niskim, średnim i wysokim, a także wprowadzenia stopnia przynależności do danego stanu, suma wszystkich stopni przynależności wynosi zawsze 100%. Wizualizacje różnic między standardową logiką a logiką rozmytą widać na rysunku (Rys. 5.30). Niestety wymaga to stworzenia dodatkowych funkcji, które wydają się dość zawiłe. Dla tego celu została stworzona klasa FuzzyLogic (Rys. 5.31), Posiada ona funkcję przeliczającą stopień przynależności wartości dla danych progów oraz wartości delta określającej płynność przejścia. Dzięki takiej funkcji w prosty sposób (Rys. 5.32) będzie można wielokrotnie użyć logiki rozmytej dla różnych wartości i w oparciu o bardziej zróżnicowane, a jednocześnie ustandaryzowane wskaźniki tworzyć logikę automatów stanów.



Rys. 5.30 Wykresy ukazujące różnice między logiką rozmytą a standardową logiką.

```

public enum poziom { Malo, Srednio, Duzo };
public class FuzzyLogic
{
    public int[] ProcentPrzynaleznosci;
    public int delta;

    public FuzzyLogic(int delta)
    {
        ProcentPrzynaleznosci = new int[(int)poziom.Duzo + 1];
        this.delta = delta;
    }
    private int CalcPerc(int val, int threshold)
    {
        return (Math.Abs(val - threshold + delta / 2) * 100 / delta);
    }
    public void CalcObj(int threshold1, int threshold2, int val)
    {
        if (val < threshold1 + delta / 2)
        {
            if (val < threshold1 - delta / 2)
                ProcentPrzynaleznosci[(int)poziom.Malo] = 100;
            else
                ProcentPrzynaleznosci[(int)poziom.Malo] = 100 - CalcPerc(val, threshold1);
        }
        if (val < threshold2 + delta / 2 && val >= threshold1 - delta / 2)
        {
            if (val < threshold2 - delta / 2 && val > threshold1 + delta / 2)
                ProcentPrzynaleznosci[(int)poziom.Srednio] = 100;
            else if (val >= threshold2 - delta / 2)
                ProcentPrzynaleznosci[(int)poziom.Srednio] = 100 - CalcPerc(val, threshold2);
            else if (val <= threshold1 + delta / 2)
                ProcentPrzynaleznosci[(int)poziom.Srednio] = CalcPerc(val, threshold1);
        }
        if (val >= threshold2 - delta / 2)
        {
            if (val > threshold2 + delta / 2)
                ProcentPrzynaleznosci[(int)poziom.Duzo] = 100;
            else
                ProcentPrzynaleznosci[(int)poziom.Duzo] = CalcPerc(val, threshold2);
        }
    }
}

```

Rys. 5.31 Klasa wspomagająca użycie logiki rozmytej.

```

public FuzzyLogic FoodLevelFuzzy
{
    get
    {
        FuzzyLogic ret = new FuzzyLogic(20);
        int val = food;
        ret.CalcObj(50, 500, val);
        return ret;
    }
}

```

Rys. 5.32 Przykład wykorzystania klasy FuzzyLogic.

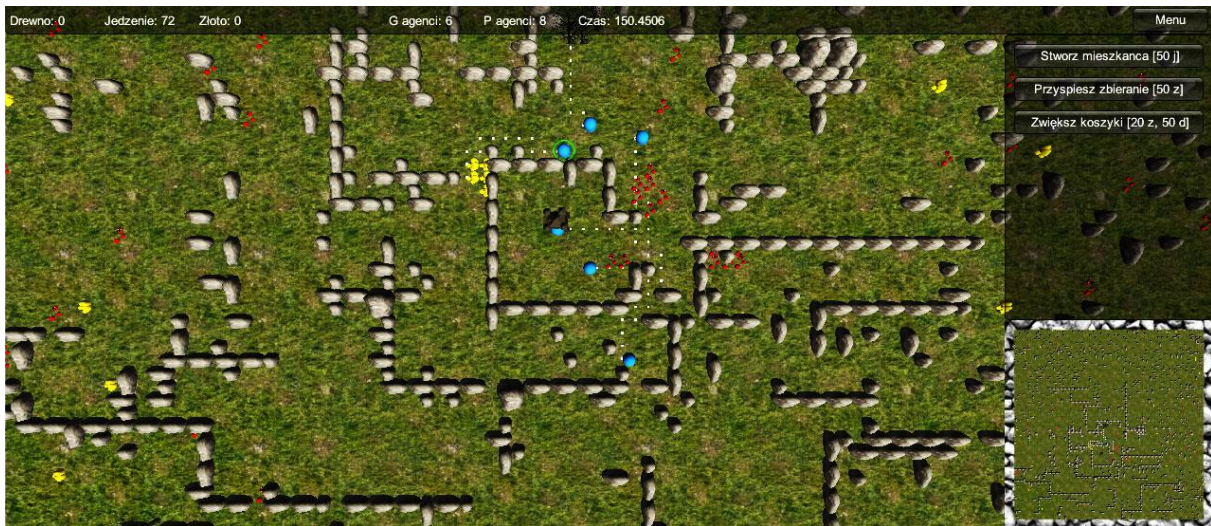
Dzięki tym wszystkim operacjom agencji występujący w grze posiadają własny cel, pamiętają wcześniejszy stan środowiska oraz potrafią dokonać decyzji na temat najkrótszej ścieżki do celu, co sprawia, że są agentami najbardziej rozbudowanego typu. Jest to jednak tylko przynależność teoretyczna, bo choć posiadają te cechy to posiadają je w małym stopniu i nie potrafią ich wykorzystać wystarczająco inteligentnie.



## 6. Podsumowanie

### 6.1. Efekty implementacji

Tworząc niniejszą pracę magisterską udało się zaimplementować prostą grę strategiczną wraz ze sztuczną inteligencją. Końcowy efekt implementacji, czyli samą rozgrywkę widać na rysunkach (Rys. 6.1, Rys. 6.2). Pierwszy z nich przedstawia agentów gracza sterowanego przez człowieka, drugi zaś agentów gracza sterowanego przez sztuczną inteligencję. Gra nadaje się do prostej rozgrywki, zatem nie jest to tylko badawczy projekt, ale faktycznie działająca aplikacja.



Rys. 6.1 Rozgrywka.



Rys. 6.2 Rozgrywka.

W celu oceny jakości wykorzystanych mechanizmów sztucznej inteligencji poproszono 5 osób o wypełnienie ankiety. Przed jej wypełnieniem należało dziesięciokrotnie zagrać w grę, przy czym pierwsza rozgrywka służyła opanowaniu zasad gry. Ankieta polegała na przydzieleniu oceny w skali od 0 do 10 poszczególnym elementom sztucznej inteligencji oraz odpowiedzi na pytania.

Pytania znajdujące się w ankiecie brzmiały następująco:

- A. Która z modyfikacji algorytmu A\* najlepiej nadaje się do gry?  
(Nr. 1 – Kara za skręcanie. Nr. 2 – Kara za nie skręcanie.)
- B. Ocena skomplikowania rozgrywki.
- C. Ocena działania sztucznej inteligencji przeciwnika – automat stanów wojownik.
- D. Ocena działania sztucznej inteligencji przeciwnika – automat stanów budowniczy.
- E. Ocena działania wydajnościowego gry.
- F. Ile razy z 10 gier udało Ci się wygrać?

A	B	C	D	E	F
Nr. 2	6	5	5	0	5
Nr. 1	8	6	8	4	2
Nr. 2	7	7	7	1	3
Nr. 2	5	3	8	3	7
Nr. 2	9	8	8	2	0

Rys. 6.3 Wyniki ankiety.

Wyniki ankiety pokazują na pierwszy rzut oka, iż wszystkim spodobała się modyfikacja algorytmu A\* nagradzająca chodzenie po skosie. Faktycznie w tym przypadku, gdy nie było możliwości bezpośredniego chodzenia po planszy na ukos ta modyfikacja nadawała racjonalności wyszukanej drodze.

Gracze jak widać dość dobrze ocenili projekt rozgrywki pod względem skomplikowania. Spora część z nich miała problem z opanowaniem tempa rozgrywki, przez co ponieśli sporo porażek. Bardzo słabo została oceniona wydajność gry, powodem był fakt, iż przy przekroczeniu 50 agentów gra działała bardzo wolno i bardzo się zawieszała. Wcześniejsze testy nie przewidywały tak długich rozgrywek, w których liczba agentów na planszy przekraczała 50. **Rozgrywka taka trwa ponad 30 minut**, a każda gra testowa wynosiła maksymalnie 10 minut.

Gracze lepiej ocenili bardziej skomplikowany automat stanów (budowniczy) odpowiedzialny za sztuczną inteligencję przeciwnika. Automat ten nastawiony był na dłuższą grę i z czasem coraz trudniej było z nim wygrać. Automat pierwszy (wojownik) już na początku starał się zakończyć rozgrywkę własnym zwycięstwem, ale łatwo było go oszukać poprzez zgromadzenie zasobów do obrony, by postawić się w lepszej sytuacji. Drugi z automatów był na taką sytuację przygotowany.

Podsumowując wydaje się, że już proste algorytmy symulujące inteligencję znacznie zwiększają jakość rozgrywki i zostają docenione. Gracze zauważyli jednak braki w implementacji, ponieważ brak wydajności związany był głównie ze sposobem implementacji, nad czym należy popracować.



## 6.2. Wady i zalety aplikacji

Zaimplementowana aplikacja ma oczywiście prócz zalet ogromną ilość wad. Ponieważ gry wymagają dużego nakładu pracy liczba wad i możliwości rozbudowy zdecydowanie przeważają.

Zdecydowanymi zaletami są:

- Dobrze działający algorytm A\*.
- Możliwość prostej rozbudowy o nowe automaty stanów.
- Uporządkowany kod źródłowy.
- Mechanizmy umożliwiające zapisywanie przetwarzanych danych.
- Możliwość prostej rozbudowy graficznej.
- Możliwość prowadzenia rozgrywki na zadowalającym poziomie.

Główne wady aplikacji to:

- Niedopracowany mechanizm reprezentacji planszy gry ze strony implementacyjnej.
- Proste automaty stanów odpowiedzialne za sztuczną inteligencję.
- Aplikacja generuje duże obciążenie, nie jest zoptymalizowana pod względem liczby rysowanych obiektów oraz akcji wykonywanych podczas rysowania każdej klatki animacji.
- Uboga oprawa graficzna.
- Mało rozbudowana rozgrywka.
- Mało możliwości taktycznych.
- Nieintuicyjny interfejs zawierający opcje, które były potrzebne przy tworzeniu gry.
- Mało intuicyjna obsługa kliknięcia i zaznaczenia obiektu rozgrywki.

## 6.3. Możliwość rozwoju

W temacie gier komputerowych możliwości rozwoju są nieograniczone. Choć można wymienić bardzo wiele pomysłów na uatrakcyjnienie gry, przede wszystkim należy skupić się na wyeliminowaniu wad. Pierwszymi rzeczami, jakie należałoby poprawić to optymalizacja wydajności aplikacji, uporządkowanie interfejsu użytkownika oraz rozbudowa automatów stanów w taki sposób, by obsługiwały podstawowe taktyki jak na przykład nękanie przeciwnika. Kolejne mogłoby być usprawnienie mechanizmu reprezentacji planszy zarówno pod względem wydajnościowym jak i w celu uporządkowania kodu źródłowego. Dodatkowo można by stworzyć edytor plansz, do czego zostały przygotowane mechanizmy zapisu oraz wczytywania planszy z pliku tekstowego.

Po wykonaniu takich podstawowych usprawnień można by urozmaicić rozgrywkę o dodatkowe zasoby, rozróżnienie typów agentów (mieszkańcy, wojownicy) lub stworzyć możliwość tworzenia budynków. Wraz z tymi wszystkimi nowościami należałoby urozmaicić sztuczną inteligencję przeciwnika w taki sposób, by wykorzystywał te funkcjonalności w swojej taktyce i strategii.

Środowisko Unity3d pozwala na stosunkowo łatwe użycie grafiki dwuwymiarowej, trójwymiarowych modeli, a nawet dźwięków, co jest również dobrą drogą do rozbudowy aplikacji pod względem odbioru audio-wizualnego. Najważniejszy jednak jest rozwój rozgrywki oraz sztucznej inteligencji na tyle, by potrafiła ona wykorzystać nowe elementy. W aktualnej wersji aplikacji zdecydowanie brakuje podsumowania rozgrywki. Podsumowanie takie mogłoby zawierać tabelę informującą o sumie zebranych zasobów, wyprodukowanych agentów czy zdobytych ulepszeń.

Wszystkie z zaproponowanych możliwości rozwoju aplikacji wymagałyby sporo czasu, ale mogłyby znacznie urozmaicić rozgrywkę. I choć rozwój można prowadzić bez końca, to warto wprowadzić część takich usprawnień.

## 6.4. Podsumowanie użytych metod sztucznej inteligencji

Wykonane pomiary wyraźnie pokazały, iż do rozwiązania problemu wyszukiwania najkrótszej ścieżki najlepiej nadaje się algorytm  $A^*$ , który jednak wymaga usprawniających modyfikacji. Algorytmy grafowe wypadły nieporównywanie gorzej zarówno pod względem czasu działania jak i pamięci potrzebnej do zapisu grafu. Istnieją jednak w komputerowych grach strategicznych problemy, które można rozwiązywać przy ich użyciu, należy tylko zwrócić uwagę jak obszerny będzie analizowany graf oraz czy oczekiwane wyniki powinny być dokładne czy tylko przybliżone jak w przypadku algorytmu  $A^*$ . Zastosowanie tych algorytmów jest uzasadnione choćby faktem, iż są to najpopularniejsze algorytmy tego typu w dodatku powszechnie używane przy rozwiązywaniu faktycznych problemów na przykład w routerach.

Jeśli chodzi o algorytmy grafowe przy problemie wyszukiwania najkrótszej ścieżki zdecydowanie lepiej spisał się algorytm Dijkstry. Algorytmy grafowe były do zaakceptowania jednak tylko przy małych planszach, więc jeśli już konieczne byłoby użycie ich, nie mogłoby to być wykorzystane w grze strategicznej, w której plansza jest rozmiaru 1600x1600 wierzchołków.

Sam sposób zapisu planszy w postaci grafu zdecydowanie nie powinien być wykonany z użyciem macierzy sąsiedztwa, macierzy incydencji, gdyż zajmują one zbyt wiele pamięci. Nie powinien być zapisany również w postaci listy sąsiedztwa, gdyż zapis ten jest niepraktyczny ze względu na operowanie na nim. Małe grafy służące do celów innych, niż zapis planszy najlepiej zapisywać właśnie przy pomocy macierzy sąsiedztwa. Szczególnie ma to uzasadnienie, gdy graf jest grafem decyzyjnym i będzie często odczytywany. Macierz sąsiedztwa, choć jest długo generowana umożliwia natychmiastowy odczyt.

Poza faktem, iż algorytm  $A^*$  wypadł najlepiej ważne jest, iż modyfikowanie go może poprawić czas jego działania. Do tego celu służą modyfikacje, takie jak sprawdzenie, czy jego użycie jest konieczne, czy inny sposób zapisu planszy jak na przykład zapis przy użyciu drzew czwórkowych lub zaproponowany uogólniony zapis pustych przestrzeni na planszy.

Poza modyfikowaniem czasu działania algorytmu  $A^*$  można modyfikować zwracaną ścieżkę, by dostosować ją do wymagań. Modyfikacje takie najczęściej polegają na zwiększeniu, lub zmniejszeniu liczby skrętów oraz wygładzeniu ścieżki. Tego typu działania mają uwiarygodnić zastosowane algorytmy w oczach gracza, by wszystko wyglądało naturalnie.

Zarówno algorytmy grafowe jak i proponowana modyfikacja algorytmu  $A^*$  wymagają dodatkowych obliczeń uruchamianych przed ich działaniem. Ważne jest, iż takie dodatkowe obliczenia nie są wadą nawet, jeśli trwają po kilkanaście sekund. Powodem jest to, że wykonują się jednokrotnie, a użytkownik widząc, iż gra wciąż się ładuje nie uzna tego za problem wydajności. W przypadku, gdy obliczenia da się zapisać do pliku i odtworzyć podczas rozgrywki czas ich przygotowania nie jest ważny.

Implementacja różnych automatów stanów pokazała, iż z punktu widzenia użytkownika jest to kluczowy element, który diametralnie zmienia rozgrywkę. Z tego powodu elementu tego nie można ignorować skupiając się wyłącznie na wydajności. A wydajność użycia samych automatów stanów można ulepszyć poprzez użycie systemu zdarzeń zamiast sprawdzania wartości w każdym kolejnym kroku.

## 6.5. Zakończenie

Podczas tworzenia niniejszej pracy udało się stworzyć działającą grę komputerową i choć jej grafika pozostawia wiele do życzenia, to z powodzeniem można w nią zagrać. Udało się również zaimplementować oraz porównać niektóre metody sztucznej inteligencji. Początkowo zakładano porównanie większej ich liczby, jednak nie wszystkie nadawały się do implementowanej gry. Nie znaleziono odpowiedniego zastosowania dla drzew gier i algorytmów przeszukiwania tych drzew. Przykładowym zastosowaniem może być najprostsza gra w kółko i krzyżyk jednak w stworzonej grze nie było podobnych problemów decyzyjnych.

Najważniejszą rzeczą, jaka się udała jest przegląd oraz opisanie wielu technik stosowanych w grach strategicznych, choć nie wszystkie zostały użyte czy zbadane. Przegląd ten pozwolił mi zorientować się w temacie sztucznej inteligencji w grach. Wiedza zdobyta podczas analizy tematu na pewno okaże się przydatna przy przyszłych tego typu projektach, które mam zamiar tworzyć, a temat bardzo mnie interesuje.

Pisanie gier strategicznych jest bardzo czasochłonne, a możliwości dalszego rozwoju gry są ogromne. Należy również zauważyć, że testowanie takiej gry również wymaga czasu, szczególnie, jeśli chodzi o nieprzewidywalne działanie, gdy automat stanu obciążony jest błędem. Tego typu błędy spowodowały, iż ilość pracy włożonej w grę wzrastała. Dodatkowo wyszukiwanie błędów w programie poprzez testowanie sprowadza się do zagrania w grę, co jest czasochłonne, a rozgrywka w nieukończonych grze wcale nie jest taka przyjemna jak może się wydawać.

## 7. Bibliografia

- [1.] AI Game Programmers Guild. Data dostępu 09.2013 – 01.2014.  
<http://www.gameai.com/>
- [2.] Alfred Aho, John Hopcroft, Jeffrey Ullman. Algorytmy i struktury danych. 2003.
- [3.] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein.  
Wprowadzenie do algorytmów. 2001.
- [4.] Ryan Creighton. Unity 3D Game Development by Example Beginner's Guide. 2007.
- [5.] Will Goldstone. Unity Game Development Essentials. 2011.
- [6.] Andrzej Kisielewicz. Sztuczna inteligencja i logika. Podsumowanie przedsięwzięcia naukowego. 2011.
- [7.] Jeff Orkin. Three States and a Plan: The AI of Fear. 2006.  
[http://web.media.mit.edu/~jorkin/gdc2006\\_orkin\\_jeff\\_fear.pdf](http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf)
- [8.] Stuart Russel, Peter Norvig. Artificial Intelligence A Modern Approach. 2002.
- [9.] Nathan Sturtevant. Memory-Efficient Abstractions for Path finding. 2007.  
<http://webdocs.cs.ualberta.ca/~nathanst/papers/mmabstraction.pdf>
- [10.] Jakub Swacha. Sztuczna inteligencja w grach komputerowych.  
Data dostępu 09.2013 – 01.2014.  
<http://klubinformatyka.pl/artykul.php?a=12&s=1>
- [11.] Dante Treglia, Mark DeLoura.  
Perełki programowania gier. Vademecum profesjonalisty Tom 1. 2000.
- [12.] TF3DM – Darmowe modele 3D. Data dostępu 09.2013 – 01.2014.  
<http://tf3dm.com/>
- [13.] Unity Technologies. Game Development Tool, 2011.  
Data dostępu 09.2013 – 01.2014. <http://unity3d.com/>
- [14.] Unity Technologies. Unity Manual. Data dostępu 09.2013 – 01.2014.  
<http://unity3d.com/support/documentation/Manual/index.html>
- [15.] Unity Technologies. Unity Script Reference. Data dostępu 09.2013 – 01.2014.  
<http://unity3d.com/support/documentation/ScriptReference/index.html>
- [16.] Krzysztof Wardziński. Przegląd algorytmów sztucznej inteligencji stosowanych w grach komputerowych. Data dostępu 09.2013 – 01.2014.  
<http://www.hc.amu.edu.pl/numery/5/wardzinski.pdf>

## 8. Załącznik: Instrukcja edycji kodu źródłowego gry

### 8.1 Potrzebne aplikacje

Możliwe, że w przyszłości ktoś chciałby rozwijać stworzoną grę. W tym celu niezbędna jest instalacja programów **Unity3d** i **Visual Studio**. Program Unity3d można pobrać z oficjalnej strony (<http://unity3d.com/unity/download>), aplikacja jest darmowa, ale istnieje również jej płatna wersja, posiadająca więcej funkcjonalności. Na oficjalnej stronie internetowej dostępna jest również pełna dokumentacja środowiska oraz udostępnianego API (<http://unity3d.com/learn/documentation>).

### 8.2 Budowanie projektu

Po instalacji obu programów można otworzyć projekt w aplikacji Unity3d. W tym celu należy wybrać z menu **File/Open Project** i otworzyć folder **game\sources\empiresstrategy**. Środowisko pozwoli otworzyć akurat ten folder, ponieważ zawiera on pliki odpowiednie dla projektu Unity3d. Tworzona aplikacja składa się z obiektów charakterystycznych dla środowiska (prefaby, skrypty, grafika, modele 3d, itd), a także bibliotek **.NET** stworzonych w **Visual Studio**.

Aby edytować wykorzystywane biblioteki należy otworzyć za pomocą Visual Studio plik **game\sources\empiresstrategy\source\empiresstrategy\source.sln**. Przed kompilacją należy upewnić się, iż projekt odwołuje się właściwie do plików. W ustawieniach projektu w zakładce **Reference Paths** powinno być odwołanie do ścieżki **game\sources\empiresstrategy\Assets\libraries**. Do tej samej ścieżki powinna wskazywać właściwość **Output Path** z zakładki **Build**. Po edycji i skompilowaniu biblioteki należy umieścić w odpowiednim katalogu **game\sources\empiresstrategy\Assets\libraries**. Następnie można zbudować pełną aplikację przy pomocy środowiska Unity3d. Środowisko pozwala również na testowanie gry i edycje kodu w trakcie jej działania. Unity3d pozwala na budowanie projektu w wersjach nie tylko przystosowanych do systemu Windows. Możliwa jest budowa, aplikacji webowej czy działającej na systemie Android. Stworzone biblioteki **.NET** mogą jednak sprawić, iż te wersje nie będą działać poprawnie.

Struktura plików oraz kodu źródłowego została opisana w rozdziale 4.2. Struktura programu.

### 8.3 Rozwój mechanizmów sztucznej inteligencji

Rozwój aplikacji pod względem algorytmicznym mógłby polegać na wprowadzeniu modyfikacji algorytmu A\*, tak by był on wydajniejszy lub bardziej realistyczny. Innym pomysłem na modyfikację mogłoby być wprowadzenie własnego systemu do zarządzania stanami obiektów oraz akcjami wykonywanymi w danych stanach. Mogłoby to uatrakcyjnić rozgrywkę i uwiarygodnić wykorzystywaną sztuczną inteligencję. Możliwe jest również wprowadzenie innych mechanizmów, jak drzewa gry, należałoby jednak najpierw to przemyśleć pod kątem wydajności oraz powodu, dla którego miałyby one działać. Inne pomysły na rozwój aplikacji zostały przedstawione w rozdziale 6.3. Możliwość rozwoju.